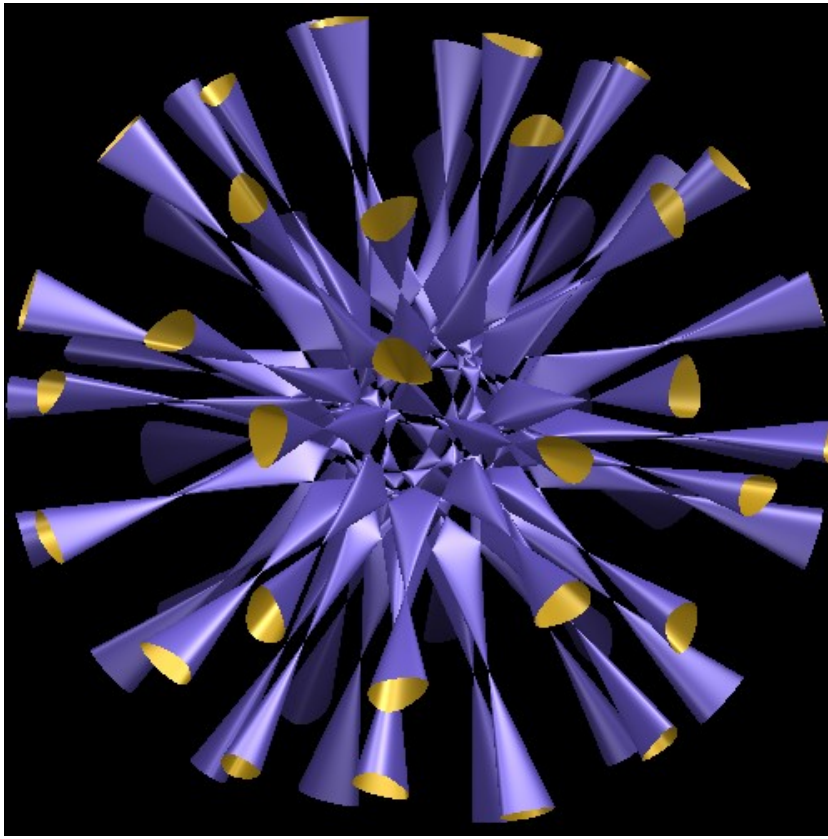


はじめての Maxima(改訂 α_4 版)

東芝インフォメーションシステムズ株式会社

横田博史

平成 20 年 6 月 3 日 (火)



MATLAB は The MathWorks の登録商標です.

Maple は Waterloo Maple Inc. の登録商標です.

Mathematica は Wolfram Research Inc. の登録商標です.

VMWare は WMWare Inc. の登録商標です.

はじめての Maxima(改訂 α 版)©(2008) 横田 博史著

この文書の内容の誤りなどによって起こった損害に対し, 工学社,KNOPPIX/Math-Project のメンバー, 及び, 著者の私は一切の責任を負いません.

この文書の二次配布を行う場合は, 著者に連絡した上でお願いします.

連絡先:ponpoko@cap.bekkoane.ne.jp

まえがき

数式処理システム Maxima は 1960 年代に開発された MACSYMA を Common Lisp 上に移植したソフトウェアです. Mathematica や Maple 等と比較して, 古色蒼然とした面も否定出来ませんが, 見掛け以上に強力なソフトウェアであり, GPL の下で配布される数式処理システムの中では非常に魅力的な汎用の数式処理ソフトウェアの一つです.

この本は Maxima の入門編, 簡易マニュアル編, 応用編と Octave による数値行列処理, そして, Maxima が利用可能な環境構築と Maxima のインストールに分かれています. 更に, Maxima を理解する上で最低限必要と思われる LISP の事と, 数学上の概念に関しても非常に簡単な説明も加えています. 簡易マニュアルに関しては, Schelter 氏が記述した Maxima-5.6 に附属のマニュアルを参考にして, Maxima-5.15.0 に対応する様に修正を加えたものです.

Maxima の動作環境としては, KNOPPIX/Math 等の UNIX 環境を前提としていますが, Windows 環境でも外部アプリケーションの起動を除けば特に問題はないでしょう. 猶, この文書はあくまでも α_3 版です. 書籍の「はじめての Maxima」と比較して, 5.15.0 に対応させる為の修正, 調べ直し, 新たに判った事や理解した事を色々と追加しています.

そして, 「はじめての Maxima」の改訂版が出版されるかどうかは, 既刊の本の売行きに依存する為に, 全く将来は不明瞭であり, 仮に出版されるにしろ, この文書のままという事は, 本文書が α 版である事からも無いでしょう. 確かな事は, 正規の改訂版の原稿を目指し, α 版から β 版へと続く更新作業が今後月一回か二回程度のペースで行われる事でしょうか.

何か人質にしている様で申し分けない事ですが, 何れにせよ, この文書をより良くする為の時間は十分にある訳で, 皆様のご協力を請う次第であります.

平成 20 年 6 月 賑やかな勲武と宇佐子に囲まれて.

狸穴主人 横田博史

目次

第 1 章 この本の趣向について	1
1.1 想定読者について	2
1.2 数式処理って何?	5
第 2 章 ちょっとした計算例	9
2.1 ユーザーインターフェイス	10
2.2 入力	13
2.3 オンラインマニュアル	15
2.4 演算子	17
2.5 式の操作	21
2.6 式の微分・積分	25
2.7 方程式の解	27
2.8 行列	29
2.9 FORTRAN や $\text{\TeX}$ への出力	32
2.10 グラフ表示	34
2.10.1 gnuplot による Klein の壺	38
2.10.2 openmath による Klein の壺	39
2.10.3 Geomview による Klein の壺	39
2.10.4 番外編	41
2.11 ファイル	41
第 3 章 LISP について	45
3.1 背景	46
3.2 数値, 文字列	47
3.2.1 LISP の代表的な数値関数	50
3.3 リスト	52
3.4 t と nil	54
3.5 配列	55

3.6	ハッシュ表	56
3.7	割当と評価	56
3.8	構造体	58
3.9	写像関数	59
3.10	lambda 式	59
3.11	関数の定義	60
3.12	制御文	60
3.13	属性リスト	61
3.14	入出力	62
第 4 章	数学の色々な事	67
4.1	集合について	68
4.2	同値関係について	69
4.3	群について	72
4.3.1	群の例	73
4.4	環について	76
4.4.1	イデアル	77
4.4.2	環の例	78
4.5	体について	80
4.6	準同型写像について	81
4.7	数式の表現	82
4.8	順序について	84
4.8.1	色々な順序	86
4.9	多項式の表現	88
4.10	Gröbner 基底の紹介	89
4.11	集合論の話題から	91
4.11.1	色々な数	91
4.11.2	濃度	92
4.11.3	自然数	95
4.11.4	超限順序数	97
4.11.5	数学の基礎を巡る論争	99
4.11.6	Hilbert 計画	104
4.12	命題と述語	106
4.12.1	小史	106
4.12.2	命題と述語について	109

4.12.3	Frege の概念記法	117
4.12.4	Russell の型理論	148
4.12.5	Hilbert による形式化	153
4.12.6	述語論理	155
4.12.7	標準形について	156
4.12.8	公理的集合論	158
4.13	帰納的関数	160
4.14	λ 計算	161
4.15	Gödel とその後	162
4.16	そして計算機	163
第 5 章	Maxima の処理原理について	165
5.1	Maxima の基礎概念	166
5.1.1	Maxima の原子	166
5.1.2	有理数と複素数の表現	169
5.1.3	Maxima の関数と演算子	170
5.1.4	属性	172
5.1.5	Maxima の式	173
5.1.6	Maxima の式の内部表現	173
5.1.7	Maxima の部分式と項	174
5.1.8	大域変数	174
5.1.9	述語	175
5.1.10	文脈	176
5.1.11	規則	176
5.1.12	式の自動簡易化	177
5.1.13	まとめ	178
5.2	順序	179
5.2.1	Maxima の変数順序	179
5.2.2	項式項に対する順序	180
5.2.3	局所的な順序の変更	182
5.2.4	順序に関連する関数	183
5.2.5	関数を含めた順序	184
5.3	演算子	186
5.3.1	Maxima の演算子について	186
5.3.2	演算子の束縛力	188

5.3.3	演算子の型	190
5.3.4	演算子の属性を宣言する関数	191
5.3.5	演算子属性の削除	196
5.3.6	算術演算子	197
5.3.7	論理演算子	200
5.3.8	割当の演算子	201
5.3.9	その他の演算子	202
5.3.10	演算子に関連する関数	204
5.4	属性	206
5.4.1	put 関数による一般的な属性指定	207
5.4.2	declare 関数 について	208
5.4.3	declare 関数で付与可能な属性	209
5.4.4	属性の追加	210
5.4.5	属性の表現関数	211
5.4.6	declare 関数に用意された属性	213
5.4.7	declare 関数以外の関数による関数属性の付加	221
5.4.8	属性の表示	226
5.4.9	remove 関数による属性の削除	228
5.5	述語	231
5.5.1	Maxima の述語について	231
5.5.2	述語の判断	233
5.5.3	述語を評価する関数	237
5.5.4	Maxima の真理関数	238
5.5.5	引数が一つの真理関数	243
5.5.6	その他の関数	245
5.6	文脈	246
5.6.1	文脈の概要	246
5.6.2	文脈に登録可能な述語	247
5.6.3	述語の登録	248
5.6.4	文脈内部での属性と述語の表現	251
5.6.5	文脈の階層	257
5.6.6	文脈の指定に関連する大域変数	259
5.6.7	変数の正值性に関連する大域変数	260
5.7	規則と式の並びについて	262
5.7.1	規則の概要	262

5.7.2	述語と変数の指定	264
5.7.3	並びの指定に関連する関数	267
5.7.4	defrule 関数による規則	269
5.7.5	defrule 関数で定義した規則の適用に関連する大域変数	274
5.7.6	tellsimp 関数と tellsimpafter 関数による規則の定義	274
5.7.7	let 関数による規則	276
5.7.8	letsimp 関数による規則の適用	278
5.7.9	letsimp 関数 に関連する大域変数	280
5.7.10	規則の削除	280
5.8	式の評価	283
5.8.1	式の自動簡易化	283
5.8.2	ev 関数	284
5.8.3	評価に関連する関数	294
5.8.4	関数や演算子に影響を与える大域変数の表示	295
5.9	LISP に関する関数	297
5.9.1	Maxima と LISP	297
第 6 章	Maxima の対象とその操作	301
6.1	数値	302
6.1.1	Maxima で扱える数値について	302
6.1.2	数値に関連する大域変数	303
6.1.3	Maxima の定数	304
6.1.4	数値に関連する関数	305
6.2	多項式	311
6.2.1	多項式の一般表現	311
6.2.2	多項式の CRE 表現	313
6.2.3	一般表現と CRE 表現への変換を行う関数	315
6.2.4	有理式の CRE 表現	316
6.2.5	係数体について	321
6.2.6	tellrat 関数	323
6.2.7	Horner 則	327
6.2.8	多項式に関する関数	328
6.2.9	その他の関数	346
6.3	級数の扱い	348
6.3.1	Maxima に於ける級数の表現	348

6.3.2	Taylor 級数に関連する函数	350
6.4	式について	352
6.4.1	変数や文字列の内部表現	352
6.4.2	二項演算の内部表現	353
6.4.3	割当の演算子の内部表現	354
6.4.4	Maxima の函数の内部表現	355
6.4.5	配列とリストの内部表現	356
6.4.6	Maxima の制御文の内部表現	356
6.4.7	表示式と内部表現	357
6.4.8	変数と変数項	360
6.4.9	部分式に分解する函数	363
6.4.10	部分式を扱う函数	367
6.4.11	総和と積	370
6.4.12	式の符号を判定する函数	377
6.4.13	式の複製を行う函数	378
6.4.14	二項演算式の検出を行う函数	379
6.5	代入操作	380
6.5.1	通常の代入函数	380
6.5.2	substpart 函数と substinpart 函数	383
6.6	式の展開と簡易化	386
6.6.1	自動展開を行う大域変数	386
6.6.2	指数函数の展開に関連する函数	389
6.6.3	式の展開に関連する函数	390
6.6.4	sum 函数の簡易化に関連する函数	393
6.6.5	簡易化を行う函数	394
6.6.6	簡易化に関する補助的函数	394
6.6.7	共通の項で纏める函数	397
6.7	リスト	398
6.7.1	Maxima のリスト	398
6.7.2	リストの生成を行う函数	398
6.7.3	リスト処理に関連する大域変数	400
6.7.4	リスト処理に関連する主な函数	401
6.7.5	map 函数族	407
6.7.6	map 函数族に関連する大域変数	408
6.7.7	map 函数いろいろ	409

6.8	集合について	413
6.8.1	概要	413
6.8.2	集合の生成に関連する関数	414
6.8.3	集合演算の関数	417
6.8.4	集合に関連する関数	418
6.8.5	集合操作の関数	421
6.8.6	集合に関連する真理値関数	425
6.9	配列	427
6.9.1	Maxima の配列について	427
6.9.2	配列操作に関連する関数	431
6.10	行列	433
6.10.1	行列について	433
6.10.2	行列を生成する関数	435
6.10.3	行列に関連する関数	442
6.10.4	eigen パッケージ	451
6.11	文字列	456
6.11.1	Maxima の文字列	456
6.11.2	ストリーム処理に関連する関数	459
6.11.3	stringproc パッケージの真理関数	464
6.11.4	文字列変換の関数	466
6.11.5	文字列操作の関数	468
6.11.6	真理関数を利用する文字列操作の関数	473
6.11.7	関連する大域変数	478
第 7 章	プログラム	479
7.1	Maxima でプログラム	480
7.1.1	block 文	480
7.1.2	if 文	481
7.1.3	do 文による反復処理	482
7.1.4	構造体	487
7.2	Maxima で関数の定義	489
7.2.1	関数定義	489
7.2.2	マクロの定義	493
7.2.3	apply 関数	496
7.2.4	関数定義に関連する関数	497

7.3	最適化	499
7.3.1	式の最適化	499
7.3.2	LISP の関数に変換する関数	500
7.3.3	コンパイルを行う関数	502
7.3.4	関数の変換やコンパイルに関連する大域変数	503
7.3.5	変数の型宣言	507
7.3.6	declare_translated 関数	512
7.4	データ入出力について	513
7.4.1	データの入出力	513
7.4.2	Maxima のファイル検出方法	513
7.4.3	ファイル処理に関連する関数	516
第 8 章	Maxima の関数	525
8.1	数論に関連する関数	526
8.1.1	階乗	526
8.1.2	剰余	527
8.1.3	Bell 数	528
8.1.4	Bernoulli 数	528
8.1.5	beta 関数	532
8.1.6	二項係数	532
8.1.7	euler 関数	533
8.1.8	Fibonacci 数	534
8.1.9	gamma 関数	535
8.1.10	li 関数	536
8.1.11	Möbius の関数 μ	537
8.1.12	numfactor 関数	537
8.1.13	ψ 関数	538
8.1.14	ζ 関数	538
8.1.15	連分数に関連する関数	539
8.1.16	二次体に関連する関数	541
8.1.17	ifactor パッケージに含まれる関数	542
8.1.18	ifactor パッケージに含まれる大域変数	546
8.1.19	numth パッケージ	549
8.1.20	Kronecker の δ と Stirling 数	551
8.2	三角関数	553

8.2.1	三角函数一覧	553
8.2.2	三角函数に関連する函数	557
8.2.3	atrig1 パッケージ	558
8.3	指数函数と対数函数	559
8.3.1	指数函数と対数函数の概要	559
8.3.2	対数函数に関連する函数	561
8.4	代数方程式	567
8.4.1	Maxima の方程式	567
8.4.2	1 変数多項式方程式の場合	571
8.4.3	区間内の根の個数	572
8.4.4	多項式方程式の場合	573
8.4.5	一般的な方程式	577
8.4.6	漸化式の場合	581
8.5	極限	582
8.5.1	極限について	582
8.6	微分	586
8.6.1	微分に係する函数	586
8.6.2	vect パッケージ	589
8.7	積分	595
8.7.1	記号積分について	595
8.7.2	変数の変換について	598
8.7.3	有理式の記号積分	600
8.7.4	記号積分の結果検証	601
8.7.5	数値積分について	610
8.7.6	antid パッケージ	613
8.8	常微分方程式	615
8.8.1	常微分方程式の扱い	615
8.9	特殊函数	620
8.9.1	Airy 函数	620
8.9.2	Bessel 函数	621
8.9.3	hypgeo パッケージ	622
8.10	楕円函数	623
8.10.1	Jacobi の楕円函数	623
8.10.2	第一種楕円積分函数	625
8.11	dynamics パッケージ	627

8.11.1	dynamics.mac に含まれる関数	627
8.11.2	complex_dynamics.lisp ファイルに含まれる関数	629
8.11.3	plotdf 関数	630
8.11.4	幾つかの例題	632
第 9 章	Maxima のシステム関連の関数	639
9.1	ディレクトリに関連する大域変数	640
9.2	システムの初期化	641
9.2.1	maxima-init.mac ファイル	641
9.2.2	rest 関数や kill 関数による初期化	642
9.3	処理の中断	645
9.3.1	中断の制御文字	645
9.3.2	break 関数による中断	645
9.4	ラベルの参照	646
9.5	結果の表示	651
9.5.1	表示に関連する大域変数	653
9.5.2	式の表示を行う関数	656
9.5.3	後戻し表示を行う関数	658
9.5.4	古風な表示を行う関数	659
9.5.5	エラー表示	661
9.6	ヘルプに関連する関数	663
9.7	システムの状態を調べる	668
9.7.1	大域変数 infolists	668
9.7.2	status 関数と sstatus 関数	671
9.7.3	room 関数	673
9.8	時間に関連する関数	673
9.8.1	処理時間に関連する関数	673
9.8.2	システムの時間を返す関数	675
9.8.3	timer 関数, untimer 関数と timer_info 関数	676
9.9	式の変換を行う関数	676
9.9.1	T _E X の書式に変換	677
9.9.2	FORTTRAN の書式に変換	679
9.10	便利な関数	681
9.11	外部プログラムの起動	682
9.12	Maxima の終了	682

9.13	虫取りに関連する関数	684
9.13.1	trace 関数と untrace 関数	684
9.13.2	bug_report 関数と build_info 関数	687
9.13.3	関連する大域変数	688
9.13.4	LISP の trace 関数との併用	689
9.13.5	システムの検証計算を行う関数	690
第 10 章	Maxima でグラフ表示	693
10.1	Maxima のグラフ表示	694
10.1.1	はじめに	694
10.1.2	表示アプリケーションについて	694
10.1.3	フロントエンド	695
10.1.4	plot2d 関数と plot3d 関数の動作	696
10.1.5	plot2d 関数	698
10.1.6	plot3d 関数	701
10.2	大域変数 plot_options	702
10.2.1	gnuplot に関連した関数	716
10.3	その他の描画関数	717
10.3.1	openplot_curves	717
10.3.2	contour_plot	720
10.3.3	Postscript に関連する関数	720
10.4	plot_option 以外の描画に関連する大域変数	722
10.5	gnuplot による描画について	723
10.5.1	maxout.gnuplot の内容について	723
10.5.2	set 命令	725
10.5.3	plot 命令による曲線の表示	725
10.5.4	splot 命令による曲面の表示	728
10.5.5	pm3d	729
10.5.6	等高線の階調の変更	734
10.5.7	等高線の調整	738
10.5.8	視点の変更	743
10.5.9	cbrange と cblabel	746
10.5.10	陰線処理	748
10.5.11	曲線と曲面の細かさの指定	754
10.5.12	描画の領域設定	756

10.5.13 ラベル表示と注釈に関連する事項	758
10.5.14 gnuplot の式	764
10.5.15 電卓としての gnuplot	769
10.5.16 プログラム言語としての gnuplot	771
10.6 plot2d 関数と plot3d 関数の活用事例	775
10.6.1 gnuplot_preamble の使い方	775
10.6.2 Maxima のバッチ処理	777
10.7 draw パッケージ	783
10.7.1 draw パッケージの概要	783
10.7.2 対象と属性について	783
10.7.3 draw 関数と draw2d 関数, draw3d 関数との関係	787
10.7.4 draw 関数	788
10.7.5 大域的な属性について	792
10.7.6 グラフの枠に関連する属性	795
10.7.7 対象について	797
10.7.8 環境の違いを調整する大域変数	808
第 11 章 積分関数の動きを観察しよう	809
11.1 積分関数ツアー募集要項	810
11.2 Maxima のソースファイル	810
11.3 integrate 関数	810
11.4 integrate_use_rootsof を用いた積分	817
第 12 章 Maxima の簡単な改造	823
12.1 使い勝手の向上を目指して	823
12.2 describe 関数は何処にある?	823
12.3 describe 関数の動作	824
12.4 関数 ponpoko の仕様	825
12.5 大域変数の作り方	826
12.6 判別について	827
12.7 外部アプリケーションの立ち上げ方	828
12.8 完成	829
12.8.1 MS-Windows 環境の場合	831
12.9 大域変数の変更について	832

第 13 章 結び目の Alexander 多項式	835
13.1 結び目の概要	835
13.2 結び目の射影図	838
13.3 結び目群の Wirtinger 表示	839
13.4 群環と Fox の微分作用素	844
13.5 Maxima で遊ぶ Fox の微分子	845
13.6 Alexander 多項式	854
13.7 Alexander 多項式の意味	859
13.8 Alexander 多項式で結び目を分類しよう	861
13.9 結び目の連結和と Alexander 多項式	863
13.10 結び目の鏡像と Alexander 多項式	865
13.11 おまけ：スケイン多項式	867
第 14 章 surf を使う話	871
14.1 代数曲面	871
14.2 surf の概要	871
14.3 Maxima から surf を使う方法	872
14.4 surf の設定の取込み方法	872
14.5 多項式の処理	875
14.6 簡単な例	880
14.7 Barth Diec	883
14.8 Steiner のローマ曲面ギャラリー	884
14.9 SINGULAR を使ってみよう	887
14.9.1 SINGULAR 初歩	887
14.9.2 SINGULAR で surf を使ってみよう	890
14.10 Maxima で終結式を使ってみよう	892
14.11 デモファイルでも書いてみよう	898
14.12 例題ファイルもついでに...	902
第 15 章 Octave で遊ぶ話	905
15.1 MATLAB クローン色々	905
15.2 Octave の基本	906
15.3 入力	906
15.4 数学定数	907
15.4.1 Octave の演算	909

15.4.2 Octave の行列	915
15.5 ベクトルに対する並びの照合	922
15.6 便利な行列の定義方法	928
15.7 多項式の扱い	931
15.8 安直な関数の定義	932
15.9 外部アプリケーションの起動命令	934
15.10 グラフ表示機能	936
15.11 Octave で file を利用する話	940
15.11.1 load 命令によるデータファイルの処理	940
15.11.2 ファイルの Open と Close	946
15.11.3 データの読み込み	947
15.11.4 ファイルの更新とデータの追加の例	954
15.12 Maxima との簡単なインターフェイス作製	956
第 16 章 KNOPPIX/Math	959
16.1 KNOPPIX/Math について	959
16.2 KNOPPIX/Math に含まれている数式処理	960
16.3 可視化ツール	962
16.4 MATLAB もどき	963
16.5 対話的幾何学ソフト KSEG	964
16.6 Maxima の例題の実行	968
第 17 章 Maxima のインストール	969
17.1 道標	969
17.2 VirtualBox で KNOPPIX を利用する場合	969
17.2.1 VirtualBox の概要	969
17.2.2 仮想計算機の設定	970
17.2.3 仮想計算機の起動について	980
17.3 VMware Player で KNOPPIX を利用する場合	983
17.3.1 VMware Player のインストール	983
17.3.2 仮想計算機の生成	983
17.3.3 vmx ファイルの作成	984
17.3.4 VMware Player で仮想計算機の起動	985
17.4 MS-Windows 環境への Maxima のインストール	987
17.4.1 インストール環境について	987

17.4.2	Maxima の入手	988
17.4.3	Maxima のインストール	992
17.4.4	初期化ファイル	998
17.4.5	環境変数 Path の設定	999
17.5	我慢強い方向け	1000
17.5.1	Common Lisp の選択	1000
17.5.2	コンパイルの手順	1001
17.5.3	Maxima の初期設定	1002
第 18 章	最後に	1003

第1章 この本の趣向について

Maxima と聞いて,それが一体何を意味するのか全く判らない方,知ってはいるがやはり何なのか見当のつかない方,そして,この本を書店で見つけてひやかし半分でこの本を眺めている貴方向けに,ちょっとした性格判断と,その傾向と対策について.

1.1 想定読者について

はじめに FAQ の形式で Maxima がどのようなもので、この本がどのような癖を持っているか簡単に説明しておきましょう。

Maximaって何？ 美味しいもの？

Maxima は汎用の数式処理システムです。つまり、数式そのものを計算機に入力すれば、計算機が貴方の代わりに計算を行うだけでなく、曲線や曲面さえも綺麗にグラフ表示してくれる美味しいシステムです。

猶、Maxima は 1960 年代の MIT の Project MAC で開発された MACSYMA (MAC's SYmbolic MANipulation system) の DOE (エネルギー省) 版を Texas 大学の Schelter 氏が Common Lisp: The language 第一版 (cltl1 と略記) に対応した gcl に移植したものです。

当初は Schelter 氏が Maxima の管理を行っていましたが、Schelter 氏の死後はメイリングリストを中心に Maxima の保守・管理と開発が進められています。詳細は、sourceforge の Maxima のサイト (<http://maxima.sourceforge.net/>) を参照して下さい。

Maxima は幾らで買えますか？

Maxima は GPL に基く無課金で利用可能なソフトウェアです。Maxima の自由な改変や配布を妨げないのであれば、その利用に制限はありません。

Maxima の入手は、<http://maxima.sourceforge.net/download.shtml> から入手可能です。更に、Maxima を収録した本も幾つかありますが、殆どは MS-Windows 版のインストールのみを収録している程度です。

猶、一枚の CD/DVD で起動可能な Linux 環境の KNOPPIX/Math や KNOPPIX Edu には Maxima が使える状態で最初から収録されています。この本には KNOPPIX/Math が収録されているので、KNOPPIX が使える環境であればどうぞ。

Maxima が動作可能な環境は？

何らかの Common Lisp が動作する環境であればまず動作可能です。Common Lisp は様々な計算機環境に移植されており、世間一般の標準的な OS、例えば、Solaris、Linux、MacOSX、MS-Windows であれば動作可能です。猶、この本では Common Lisp の中でも多くの環境に移植されている CLISP で利用する事を前提にしていますが、処理系の違いが問題になる様な高度な事は行っていません。

どの様な読者層を想定していますか？

この本では以下の読者を想定しています：

- 我慢強い方 (ソースの修正,configure,make は当たり前)
- 軟派な方 (楽をしたい. 興味があるので一寸使ってみたい)
- MS-Windows 派 (MS-Windows 上で全てを済ませたい)

先ず, 我慢強い方は Linux 環境で使う事を想定しています. Linux にはツールやアプリケーション等のパッケージを纏めた様々なディストリビューションと呼ばれるものがあります. これらは大きく分けて RedHat ,Debian ,Slackware の3種類があり, パッケージ管理の手法 (極端にはパッケージの修飾子) で殆ど区別出来ます. RedHat の場合は.rpm,Debian の場合は.deb,Slackware の場合は.tar.gz となります.

但し, この本ではディストリビューションは特に問題としません. 実際, 自力でインストールする場合でも, 以前と比較して非常に容易に行える様になっています.

軟派な方にはKNOPPIX/Math という素晴らしい玩具箱を用意しました.KNOPPIXの詳細に関しては産総研のサイト (<http://unit.aist.go.jp/itri/knoppix/>) を参照して下さい. KNOPPIXは一枚のCDやDVDから立ち上げ可能なLinuxです. 更に,KNOPPIX/Mathは数学関係のソフトウェアを集積したKNOPPIXの分派です. 手軽にLinux上で動作する数学ソフトを利用したい方にはおすすめです. KNOPPIX/Mathの情報は<http://www.knoppix-math.org/>から入手出来ます. 猶,KNOPPIX/Mathを使わなくても, このCD-ROMには様々な数学ソフトに関する文書があるので, 無視しないで眺めて下さい.

最後のMS-Windows派の方にはMS-Windows版のインストールの話と修正したプログラムの読込に関する話があります. インストーラーを入手してインストールすると一番安易に使える一方で, 自分で色々と工夫をしはじめると, 実は一番苦勞しなければならない方です.

でも, 福音があります.VMware Player や VirtualBox を使えばMS-Windows環境上でもLinuxを動かす事が出来ます. そうすれば両方の楽しい所が堪能出来ます. この際にVMware Player や VitrualBoxを使ってKNOPPIXで遊ぶのは如何でしょうか? 楽しいですよ.

そして,Maximaは数学のソフトウェアです. ですから, 数学の知識があるに越した事はありません. この本ではささやかですが,Maximaの考え方に必要な数学の事を書いています.Maximaも車と同じ道具です. 使い方だけを知ってさえいれば, 利用に困る事は無いでしょうが, その原理を知っていれば応用も効きますし, 何よりも, より楽し

く使えます。個人的には、群、環、イデアルと聞いて、心ときめかせられる高校生、大学生、一般社会人が増えれば良いなと思っています。

私は Mac ユーザーです !!

Mac OS X を UNIX として使える様に環境設定していれば、FINK を用いたインストールが出来る様です。又、最近の Intel Mac では KNOPPIX の立ち上げも可能との事です。それ以外の古典的 Mac OS の利用者の方々は Mac 版の Linux (Debian, OpenSuSE や Vine 等) をインストールすれば普通に使えます。

猶、この本では Mac ユーザーは我慢強い方に分類しています。日常的に我慢を強いられる事が多いでしょうから…。

初心者なのでよくわかりません

先ず、この本は小説とは違い頭から読む必要はありません。この本には Maxima を使う上での必要となる予備知識や雑多な事を色々と詰め込んでいるので厚くなっています。そこから、貴方が必要と思えるものだけを読めば良いのです。但し、Maxima がどのようなものであるか想像がつかないのであれば、この章と §2 の「ちょっとした計算例」を最初に読んで、雰囲気把握し、それから、§5.1 には Maxima の基本的な考え方を纏めているので、こちらを眺めて必要に応じてマニュアル部分等を読むのは如何でしょうか。

Maxima の利用方法で、より初心者向けに纏まった話が必要な場合は、KNOPPIX/Math の CD-ROM/DVD-ROM の文献を参照して下さい。この文献は、Math-ja フォルダの中にある PDF フォルダに収納されている maxima-note.pdf という PDF 文書です。これは中川義行氏による「Maxima 入門ノート」です。私の本が世間一般で言われる初心者向けでない理由の一つが、この「Maxima 入門ノート」があるお陰です。

一応、この本では KNOPPIX/Math を推奨環境としています。猶、Maxima を MS-Windows 環境にインストールして使いたい方で、何をどうすれば良いのか判らないという方には MS-Windows 環境へのインストールの仕方を §17.4 に書いています。

そして、些細な事でも Maxima を使って色々と試してみましょう。兎に角、日常的に利用し、試行錯誤したり遊んだりする事が一番の上達の方法です。

1.2 数式処理って何?

さて前節の FAQ では,

- Maxima は汎用の数式処理である
- Common Lisp が動作する環境があれば Maxima を動かせるかも

と申しました. とは言え数式処理? Common Lisp? となる方も多い事でしょう. ここでは, もう少し詳しく解説致しましょう.

先ず, 数式処理システムは数式自体の直接処理を目的としたソフトウェアです. C や BASIC でも数式を扱えるじゃないかと思う方も多いと思いますが, 標準的なライブラリを用いて C や BASIC で扱う数式は, その式から得られる数値を扱っているのであって, 数式それ自体を扱っている訳ではありません.

具体的に考えましょう. 先ず, $1+1$ は C や BASIC でも簡単に処理出来ます. 128 の因数分解はどうですか? プログラムを組めばなんとかなりますね. 例えば, "エラトステネスのふるい" と呼ばれるアルゴリズムを使って処理が出来ます.

次に, x が 1 の時に $x^2 + 2x + 1$ を計算する事も簡単ですね. それでは, $x^2 + 2x + 1$ の因数分解は如何ですか? 更に, $(x+1)(x-1)^{10} + (4x-2)^4$ なんて式を展開する事は? 更には, $2\cos^2(\theta) + \cos(2\theta)$ を簡単な式にするとか, sin や cos といった初等函数の微分や積分等は?

この場合, 式を単なる文字列として見るのではなく, 変数は何で, どのような性質を持った演算や函数が用いられているかをシステムは理解していなければなりません.

この様に数式に含まれる記号をそのまま扱い, 与えられた式の様々な計算を行う事を数式処理は目的としています. この処理では代数学の考えが色々用いられており, そんな事もあって数式処理は記号代数処理とも呼ばれています.

数式処理自体はシステムに公式集を持たせて与えられた式を機械的に公式にあてはめて行けば, それで出来てしまう様に見えます. 実際, 公式にあてはめる事で数式の直接処理を行うものもあります.

最初に初歩的な例として, Prolog を用いた簡単な微分の例を見てみましょう. 先ず, Prolog に x による微分の初歩を教えてやります.

```
dx(x,1).
dx(Y,0) :- atomic(Y), Y \== x.
dx(Y+Z,DY+DZ) :- dx(Y,DY), dx(Z,DZ).
dx(Y-Z,DY-DZ) :- dx(Y,DY), dx(Z,DZ).
dx(Y*Z,DY*Z+Y*DZ) :- dx(Y,DY), dx(Z,DZ).
```

```
dx(Y^N,N*Y^N1*DY):- integer(N), N1 is N-1, dx(Y,DY).
```

Prolog の様な言語では, 論理式を入れて行く方式になります. 例えば, $dx(x,1)$ の意味は, x による x の微分が 1 であるという意味で, 以降, $dx(Y,0)$ は自由変数 Y がアトムで, Y が x と等しくなければ 0 であると述べています. 以降, 和と差の場合の微分, 積の場合と定数の冪の場合の微分について述べています.

SWI-Prolog でこのプログラムを実行した時の例を以下に示しておきましょう.

```
?- consult(diff).
```

```
% diff compiled 0.00 sec, 2,560 bytes
```

```
Yes
```

```
?- dx(x^2+2*x*y+y^2,Y).
```

```
Y = 2*x^1*1+ ((0*x+2*1)*y+2*x*0)+2*y^1*0
```

```
Yes
```

この例では, 最初に `consult` 命令でファイル `diff.pl` の読込を行い, それから, $dx(x^2+2*x*y+y^2,Y)$ で $x^2 + 2xy + y^2$ の x による微分を計算させています. 但し, このプログラムでは, $1*x$ の処理, 0 倍の処理や式の展開やまとめといった処理が欠落しています. 即ち, この処理では, 計算機は x が何で y が何であると言った事を全く知らずに機械的な処理だけを実行しています. この方法でも計算の補佐役としては十分実用的ですね. 但し, 計算結果を鵜呑みにする場合, この単純に公式に当て嵌めるだけの手法は大きな危険性を持っています. 例えば, $1/x^2$ を区間 $[-1, 1]$ で定積分する事を考えましょう. 一般的な公式集では, $f(x)$ の原始函数が $F(x)$ の場合の区間 $[a, b]$ での函数 $f(x)$ の定積分は次の式で与えられます:

$$\int_a^b f(x)dx = F(b) - F(a)$$

ここで $n \in \mathbb{N}^+$ ($\mathbb{N}^+$ は自然数の集合から 0 を抜いたもの) の場合, $1/x^n$ の不定積分は一般的に以下の公式で与えられますね.

—— $1/x^n$ の不定積分の公式 ——

$$\int \frac{1}{x^n} dx = -\frac{1}{(n-1)x^{n-1}} + c$$

さて、これらの公式を $1/x^2$ の積分に適用すると、以下の結果が得られます。

$$\frac{1}{(2-1)(1)^{(2-1)}} - \frac{1}{(2-1)(-1)^{(2-1)}} = 2$$

簡単な事です…しかし、残念な事に、この計算は間違いです。何故なら $x=0$ でこの函数は無限大に飛んでしまうので、この場合は公式が成立しない為です。この事を確認してみましょう。先ず、 ϵ を十分小さな正の実数として、 $[\epsilon, 1]$ の区間で上の式を積分してみましょう。この場合、 $\frac{1}{\epsilon} - 1$ となります。ここで ϵ が 0 に近づくとどうなるでしょうか？この値は一方的に増大し、その上限が無い事も判ります。何故なら、正の実数 M で $\frac{1}{\epsilon} - 1$ が抑えられると仮定しましょう。ここで ϵ を $\frac{1}{M+1}$ よりも小さい値、例えば $\epsilon = \frac{1}{2(M+1)}$ にしてみましょう。すると $\frac{1}{\epsilon} - 1 = 2M + 1$ となるので、間違い無く M よりも大きくなります。これは矛盾なので、この様な M が存在しない、即ち、この函数には上限が無い事が判ります。この様に上限が無い時の事を無限大と呼びます。更に、 $1/x^2$ の値は常に正で、Y 軸に対して対称になっている事から、 $\int_{-1}^1 \frac{1}{x^2} dx$ の値が 2 になる筈が無い事が判るでしょう。この函数 $1/x^2$ の $x=0$ の様に、函数の値が無限大になる点の事を函数の極と呼びます。この例の様に定積分を行う領域に被積分函数の極が存在する場合は問題が生じます。

そこで、数式の処理をちゃんと実行する為には、与えられた式の計算で一般的な公式が利用可能な範囲に収まっているかどうかを検証する必要があります。その事を忘れると、当たり前ですが先程の結果の様に間違った結果が返されてしまいます。従って、より本格的な数式処理は単に公式に当て嵌めるだけではなく、与えられた式に含まれる函数等の特徴も把握するシステムであるべきです。

では、この様な数式処理はどの様な計算機言語で記述されているのでしょうか。現在は C や C++ で記述する事が多くなっていますが、古いものになると LISP で記述された物が多くあります。先程は Prolog の例を出しましたが、LISP は 1980 年代の人工知能 (AI) が流行った時に、Prolog と共に脚光を浴びた言語でもあります。この LISP という計算機言語は、FORTRAN とほぼ同時期に生れた非常に古い言語ですが、その一方で非常に柔軟な言語です。但し、規格化の進んだ FORTRAN と比べて非常に方言の多い言語で、Common Lisp はその LISP 言語の方言の一つです¹。

LISP の特徴は色々ありますが、FORTRAN が数値を扱う事を主とした言語であるのに対して、LISP はリストと呼ばれる記号や数値で構成されるデータ形式が容易に扱える特徴があります。更に、LISP はプログラム言語的には函数型と呼ばれるものになり、定義した函数を組合せてプログラムを組む様式になります。因に、C や FORTRAN は

¹Common Lisp の他に Scheme があります。

手続型と呼ばれるプログラム言語になります。猶、プログラム言語には他にも色々ありますが、先程の Prolog は論理型と呼ばれる言語になり、こちらは、プログラミングと処理の間に区別が無い独特の処理言語です。

この本で解説する Maxima は Common Lisp で記述された汎用の数式処理システムです。この Maxima は微妙な所で LISP が顔を覗かせており、更に、パッケージを記述する場合は Maxima の処理言語ではなく、Common Lisp で記述される事が多くあります。又、汎用の数式処理と言う事は、様々な問題に対処可能である事を意味します。実は、数式処理には色々あって、非常に専門的対象に限定した処理を行うものや、Maxima の様に幅広い対象に適用可能なものがあります。但し、汎用のものは、専用のもものと比較して、浅く広く処理を行う傾向があり、専用のもの方が効率が良かったり、汎用のものでは不十分な処理しか出来ない事もあります。この点は、普通の道具と同様で、利用者がその特性を見極める必要があります。但し、気楽に使う為には、最適な環境を提供してくれるでしょう。

次の章で Maxima について実例を交えて簡単な説明を行います。

—— コラム：数学ソフト色々 ——

本格的な数式処理で有名なものに AXIOM [59], *Mathematica* [64], Maple [65], MuPAD [67] や REDUCE [69], それに、TI のグラフ電卓でも使われている DERIVE [60], ワープロと合体した mathcad [63] やカルキング J [73] 等が挙げられます。これらは高校生や大学の教養程度の数式の処理が可能なものです。

これに対して、数学の特定分野に特化したシステムもあります。これらは GPL や BSD license に似たライセンス形態を持つものが殆どです。

ツールボックスで Maple が使える MATLAB [74] は数式処理とはまた別の種族で、数値行列データを効率的に扱う事を目的とした言語仕様となっています。この MATLAB は (小中規模の) 数値行列の処理で広く使われており、標準的なソフトウェアになっています。その為に、Octave, Scilab, RLab 等の MATLAB 風のシステムが多くあります。

統計処理が必要な場合は S 言語の系統 (S-PLUS [79] や R [78]) が広く用いられています。

第2章 ちょっとした計算例

この章では Maxima で何が出来るかを具体的に実例を使って紹介しましょう。
猶, ここでの実例で紹介した関数で気になったものは Maxima のオンラインマニュアルや, この本に収録した簡易マニュアルを参照して下さい。

2.1 ユーザーインターフェイス

まずは Maxima で一寸した計算をしてみましょう. 現在, Maxima には様々なフロントエンドがありますが, KNOPPIX や Windows 版には昔ながらの maxima と GUI を用いた wxMaxima や xmaxima が使えます.

どれを使っても構いませんが, 現時点では wxMaxima が一番敷居が低いでしょう. 何故なら, MS-Windows 版の Maxima では wxMaxima を標準のフロントエンドとしているからです. その為, MS-Windows 版を使っている方には wxMaxima=Maxima と思っている方も多いかもしれませんが, wxMaxima は Maxima とは別個のアプリケーションです. 実際, MS-Windows 以外の環境で利用する為には, 別途, wxMaxima を導入する必要があります. 勿論, KNOPPIX/Math にはちゃんと収録されているので気楽に使えます.

この wxMaxima を使うと, 図 2.1 に示す様に数式が美しくレンダリングされて出力されます. 更に, 大小の左括弧を入力すると同時に右側の括弧も補完する機能もあるので慣れると便利でしょう.

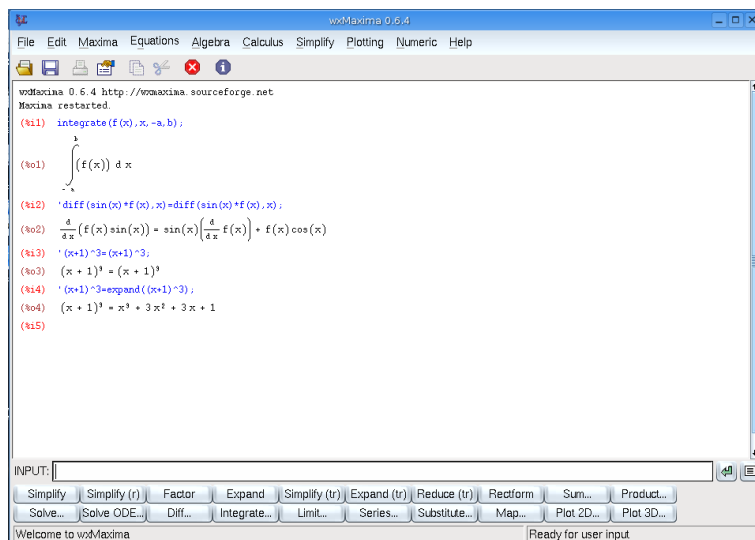


図 2.1: wxmaxima

ところで, wxMaxima は Maxima と通信を行う事で wxMaxima が受け取った利用者からの入力を Maxima に処理させて, その結果を wxMaxima が受け取る方式を採用しています. ここで PC のファイアーウォールの設定によっては, この通信が遮断され

る事がある事に注意して下さい. この場合, フロントエンドの wxMaxima が立ち上がるだけで, 処理を行う Maxima と遣り取りが出来ない為に, 何等の処理が出来なくなります. もしも, wxMaxima を立ち上げても変なメッセージしか出てこないとか, ファイヤーウォールソフトから何らかの警告が出た場合, ファイヤーウォールの設定を変更して, wxMaxima-Maxima 間の通信が行える様にして下さい.

次の xmaxima は Maxima のソースファイルに標準で附属しているフロントエンドです.

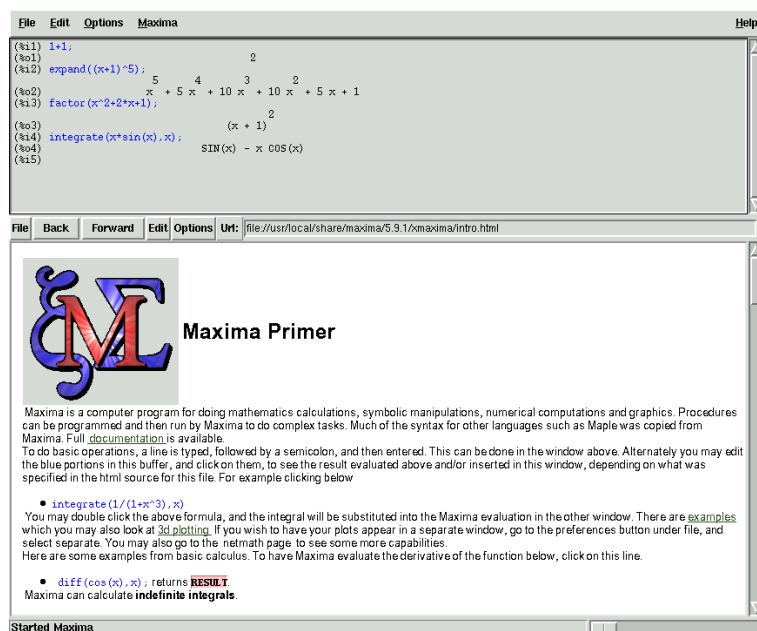


図 2.2: xmaxima

MS-Windows 版では wxMaxima の前に標準的なフロントエンドとして収録されていましたが, 現在では, wxMaxima がその立場に立っています.

xmaxima のフロントエンドは図 2.2 に示す様に, 上下の副ウィンドウに分割されており, 通常の入力はウィンドウの上段で行います. 又, グラフは plot_format が openmath の場合, 上側副ウィンドウに表示されるので, 1995 年頃の Maple V の様な雰囲気になります. 下側副ウィンドウには入門用の例題が表示されています. maxima と xmaxima の両方共, 数式の表示は ASCII 文字を用いたもので, TrueType 等のフォントで美しく数式が表示される商用のものと比較すると明らかに劣ります.

その他には TeXmacs や imaxima があり, これらも数式を商用のものと劣らない程, 美しく表示します.

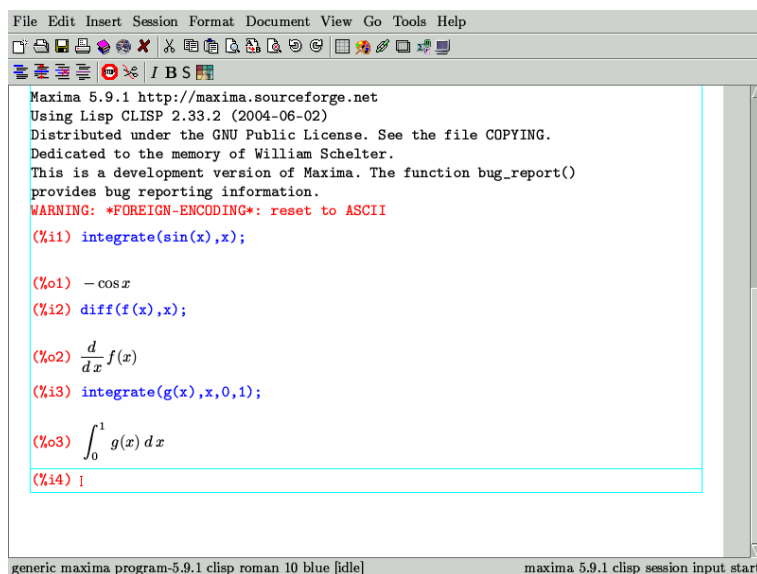


図 2.3: TeXmacs

ここで TeXmacs は Maxima 専用のフロントエンドではなく, 他の, gnuplot, Macaulay2, Octave, PARI/GP のフロントエンドとしても使えます. Maxima のフロントエンドとして利用する為には, Insert メニュー → Session → maxima で Maxima を選択します.

因に, ここで解説したフロントエンドは全て KNOPPIX/Math に衆力された Maxima で利用可能です.

猶, この本では基本的に wxMaxima 等のフロントエンドを用いない素の Maxima の入力と出力を表示します. この理由ですが, どの様な環境でも利用可能なので, 検証を行う上で優れている点に加え, こちらの方が慣れてしまえば手軽な為です.

2.2 入力

さて,Maxima を立ち上げると入力行のプロンプト (`%i1`) が出ています. 式はこのプロンプトの後に続けて入力します.

ここで入力行末尾には必ずセミコロン; か \$ といった入力行の末尾を示す文字を付けなければなりません. 何故なら, これらの行末の文字が入力される迄,Maxima は式の入力が継続していると判断して入力の完了を待ちます. この時はプロンプトが現われずに,Enter キーを押しても改行されるだけになるので容易に分ります.

では簡単な数値計算を幾つか実行させてみましょう.

```
(%i1) 1+2;
(%o1)                                     3
(%i2) 2/3+3/5
;
                                     19
(%o2)                                  --
                                     15

(%i3) display2d:false;
(%o3) false
(%i4) 2/3+3/5;
(%o4) 19/15
(%i5) (%o1)+(%o2)*15;
(%o5)                                     22
(%i6) quit();
```

Maxima で和+, 差-, 積*, 商/といった基本的な演算記号 (演算子と呼びます) は C や FORTRAN と同様です. この例では最初に $1+2$ を Maxima で計算させています. この時, 入力式は C の数式と同様に `1+2;` と入力します. Maxima は入力式が数値であれば直ちに計算を行い, その結果を返します

入力行のプロンプトには (`%i` 番号) といった入力のラベルが付いています. この入力に対応する出力行の先頭には (`%o` 番号) という出力のラベルが付きます. 数式の入力はこの入力のラベルに続けて行い, 結果は出力のラベルに続けて出力されます. Maxima では入力値とその結果は個々のラベルに保存されます. この例では, 入力側のラベル (`%i1`) の内容は $1+2$ で, それに対応する出力側のラベル (`%o1`) の内容は 3 になります. 次は分数の計算例です. ここでは $2/3+3/5$ の計算をさせていますが, この例ではセミコロンを行末に付けなかった為, Maxima は入力待ちになっています. 即ち, (`%i2`) 行

に $2/3+3/5$ と入力しても何も表示されていません。そこでセミコロン;`;`を入力すると直ちに計算結果を返し、その結果は出力ラベル (`%o2`) の行に表示されています。この様に、Maxima は、入力行に;`;`や`$`を入力する迄、式が入力中であると判断し、計算処理を行いません。そして、;`;`が行末であれば、その入力行の処理結果を表示し、`$`が行末であれば、その入力行の処理結果を表示しません。

Maxima は wxMaxima や TeXmacs を利用している場合、綺麗にレンダリングされた式が返されるのですが、通常の Maxima や xmaxima では計算結果を ASCII 形式で数式らしく表示しようとします。その為、ウィンドウが小さな場合、式が少しでも複雑なものや、長くなると意味不明になる事もあります。更に、計算結果を他のプログラムに書き写そうとしても、この 2D 表記ではそのまま使えません。この事を避ける為、Maxima の大域変数 `display2d` に `false` を割当てて、この表示を止めさせる事が出来ます。ここでの例で大域変数 `display2d` に `false` を割当てる為に演算子:`:`を用いています。ここで演算子とは Maxima の式を繋ぎ合わせる働きをする記号で、その様にして出来た Maxima の式を Maxima は解釈します。さて、この割り当ての演算子は、他の数式処理や C では演算子:`:=`を用いたりしますが、Maxima では変数に値を割当てる場合は演算子:`:`を用います。因に Maxima で演算子:`:=`は関数定義で用いる演算子、演算子:`=`は等号の演算子となります。この点は C や FORTRAN と異なるので注意しましょう。

さて、大域変数 `display2d` を `false` に変更すると `2/3+3/5;` の結果は `19/15` と表示され、一行に収まる様に表示されていますね。ここで、初期状態の 2D 表示に戻したければ `display2d:true;` と入力すれば良いのです。

この様な大域変数を Maxima は幾つか持っています。大域変数には一つの函数動作のみに影響を与えるものから複数の函数に影響を与えるもの等と色々あります。

次の `(%o1)+(%o2)*15` の意味は、`(%o1)` に表示された計算結果に `(%o2)` で示された計算結果に対して 15 倍したものの和を計算させる事を意味します。Maxima では計算結果を入出力の値を左側のラベルに対応させて保存しているので、一々、変数に割当てなくても値は保存され、履歴を用いた計算が行えるのです。但し、この履歴は Maxima を普通に終了すると消えてしまいます。

ここで Maxima の履歴を参照する主な大域変数や函数に、`%` と `%th` があります。`%` は直前の計算結果が割当てられており、`%;` と入力するだけで、直前の計算結果が参照出来ます。次の函数 `%th` は整数を引数に持ち、`%th(n)` で `n` 回前の計算結果が参照出来ます。猶、`%th(1)` は `%` と同じです。

この履歴はバッチ処理でも利用可能です。因に、バッチ処理というものは、処理する内容を予めファイル等に記述しておき、それをアプリケーションで逐次実行させる処理方法で、この目的で用いるファイルの事をバッチファイルと呼びます。このバッチ処理を上手く使うと定型的な処理が非常に楽に行えるのです。ここで、ラベルを直接 (`%o10`)

の様に直接指定して利用する場合, その処理以前の利用状況によってラベルの位置が色々と変化するので注意が必要になります. 実際, 初期化の方法が計算機の環境毎に異なっていれば, それだけでラベルの位置が異なるかもしれません. その上, ファイルが大きくなればなる程, 確認が面倒になって混乱を招き易いので, 相対的にラベルを指定する方が, 通常の利用でも問題が少なく安全でしょう.

次に, `kill(labels);` と入力すれば, ラベルに割当てた値を全て破棄します. この関数はバッチファイルの実行や, メモリが足りなくなった場合で用います. この処理を行うと履歴が全て消されてしまうので, %等で以前の履歴が参照出来なくなります. 更に, `kill(labels)` の実行によってラベルの番号は全て 1 に戻されます. その為, `kill(labels)` を頭に入れて, その後に実際の処理を行うバッチファイルであれば, %o1+%o2*15 の様にラベル番号を直接指定する処理でも問題はありません.

最後に Maxima の終了は `quit();` で行います. この quit 関数は引数が必要ありませんが, `quit;` と入力しても終了しない事に注意しましょう. 終了する場合には必ず quit の後の小括弧 () を忘れてはいけません. Maxima では, 引数を必要としない関数に対し, () を省略する事は出来ません.

2.3 オンラインマニュアル

それでは Maxima をまた立ち上げましょう. 今度は `help;` と入力してみてください.

```
(%i1) help;
(%o1)          type describe(topic) or example(topic);
(%i2)
```

この様に describe 関数や example 関数を使えと出てきます. 最初の describe 関数は texinfo を用いたオンラインマニュアルを表示する関数で, 次の example 関数は Maxima の例題ファイル (*.dem) を実行する関数です. 主要な関数や変数にはオンラインマニュアルの項目があります. 猶, 例題は重要な関数であれば大抵存在しますが, 存在しない事もあります.

describe 関数は一寸した字引代りにも使えます. 例えば, 関数や大域変数の名前が分からなくても, その関数や大域変数に関連する事項を指定すれば, 関連する項目を Maxima が一覧として表示してくれます. 例えば, 方程式を解きたい場合, キーワードを solve として `describe(solve);` と入力してみましょう.

```
(%i36) describe(solve);
```

```

0: desolve : (Maxima.info) definitions for differential equations.
1: fast_linsolve : definitions for affine.
2: funcsolve : definitions for equations.
3: globalsolve : definitions for equations.
4: linsolve : definitions for equations.
5: linsolve_params : definitions for equations.
6: linsolvewarn : definitions for equations.
7: solve : definitions for equations.
8: solve_inconsistent_error : definitions for equations.
9: solvedecomposes : definitions for equations.
10: solveexplicit : definitions for equations.
11: solvefactors : definitions for equations.
12: solvenullwarn : definitions for equations.
13: solveradcan : definitions for equations.
14: solvetrigwarn : definitions for equations.
15: zsolve : definitions for equations.
enter space-separated numbers, all or none:

```

この状態から 7 を入力すると番号 7: の solve 関数のオンラインマニュアルが表示されます。猶、Maxima の describe 関数は単純に info ファイルを表示するだけなので、使い勝手はあまり良いものではありません。その上、マニュアルの内容が詳細になれば、仮想端末のバッファに収まりきらない事がある為、頭から読む事が出来ない事もあります。オンラインマニュアルを使って色々と調べながら Maxima を利用するのであれば、wxmaxima, xmaxima や TeXmacs, 或いは GNU Emacs をフロントエンドに利用した方が良いでしょう。

describe 関数と同じ働きをする特殊な記号 (演算子) に "?" があります。[?] 事項 は [describe(事項)] と同じです。ここで "?" の後には必ず空白 (space か tab) を入れましょう。空白を入れなければ、Maxima の裏で動いている LISP によって調べたい事項が LISP の関数と看做されます。但し、この場合は引数が無い為にエラーにはならず、単純に false が返却されるだけです。

関数や大域変数の綴を忘れた場合、apropos 関数を使いましょう。この関数は与えられた文字列から適合する関数名や大域変数等のリストを返す関数です。例えば、積分関数が int か integrate か判らなくなった場合、[apropos(int)] と入力すると、Maxima 内部の関数や大域変数で int が頭に付く関数や大域変数のリストを返します。

```
(%i11) apropos(int);
```

```
(%o11) [int, intanalysis, inte, integer, integerp, integer_partitions,
integfactor, integrate, integrate_use_rootsof, integrationconstant,
integration_constant_counter, interaction, interpolate, interpolate_subr,
intersect, intersection, intfaclim, intfactor, intopois, intosum, intp,
intpolabs, intpolerror, intpolrel, int_partitions]
```

この apropos 関数は関数と大域変数等を一つのリストに纏めて返す為、場合によっては不必要に長いリストが返却される事もあります。この int の場合でも、整数 (integer) に関連するもの等と、積分 (integrate) 以外のものが色々と含まれているので、返却されたリストが長くなっている事が判りますね。

この様に apropos 関数と describe 関数を使えば、Maxima をより便利に使う事が出来ます。

2.4 演算子

Maxima で扱う数式は通常の数学で用いられる式と殆ど同じものが使えます。和、差、積、商といった四則演算 (+, -, *, /) は FORTRAN 等のプログラム言語で用いられる数式と同じ使い方です。猶、このような記号の事を演算子と呼びます。Maxima にはこれらの他にも色々な演算子があります。

その中で、Dot 積と呼ばれる積演算子があります。例えば、 a と b の Dot 積を $a . b$ で表記します。この Dot 積は通常 $a . b$ と $b . a$ は等しくなりません。これは通常の積 $*$ と比較すると随分、奇妙な演算です。実際、 $2 * 3$ と $3 * 2$ は同じ 6 になりますね。普通の生活で用いられる積は演算子の左右に置かれた被演算子を交換する事が可能です。この性質を可換性と呼びます。この Dot 積の様に可換でない積は具体的にはどのようなものでしょう？思い出して頂きたいのですが、高校生で習った行列の積の性質は如何でしたか？行列の積は非可換にならない積演算の中では最も有名なものでしょう。

この本では $*$ の様に可換な積を可換積と呼び、可換でない積演算子の事を非可換積と呼ぶ事にします。

Maxima には、これらの二つの積演算子に対し各々に対応する二つの冪演算子があります。可換積に対応する冪が $\wedge$ 演算子で、非可換積に対応する冪が $\cdot$ です。例えば、 a^3 は $a * a * a$ を意味し、 $a^{\cdot 3}$ は $a . a . a$ を意味します。猶、演算子 $\wedge$ は FORTRAN の冪乗の演算子 $**$ と同じ意味ですが、Maxima の演算子 $**$ もまた可換積の冪です。

さて、今迄解説した演算子は左右に引数、則ち、被演算子が置かれる演算子でした。Maxima の演算子には被演算子が前に置かれる演算子もあります。例えば、階乗 $!$ は $3!$ の様に数学の表記方法と同じ書き方になります。この階乗と似たものに演算子 $!!$ があり

ます. こちらは $n!!$ で n より 2 を引いた整数の値の積, 即ち, $n(n-2)\cdots(n-2k)\cdots$ の計算を行います.

```
(%i14) 2*3+5/(2*(4+1));
                                     13
(%o14)
                                     --
                                     2
(%i15) (x+1+y^2)/(x*(x+y^2+1))-1/x;
(%o15)
                                     0
(%i16) 11!
;
(%o16)
39916800
(%i17) 11!!!;
(%o17)
10395
(%i18) factor(%o17);
                                     3
(%o18)
3  5 7 11
(%i3) stardisp:true$
(%i20) factor(11!!!);
                                     3
(%o21)
3  *5*7*11
```

この例では最初に整数の演算, 次に多項式の演算を行っています. 最後に階乗!と!!の違いを示しています. ここで factor 関数は整数や式の因子分解を行います. ここでは 11!!! の計算結果 %o17 の結果を因数分解しています. この結果から 11!!! は $3 \cdot 5 \cdot 7 \cdot 9 \cdot 11$ と 11 以下の奇数全ての積となっている事が分ります. ここで, Maxima で可換積記号の表示は, 通常, 空行に置換えられて表示されています. この可換積演算子*の表示は大域変数 stardisp の設定で制御しています. 初期値の false で可換演算子の表示を行わず, true の場合に可換積演算子*を表示します.

Maxima の演算子には幾つかの種類があり, それは被演算子の取り方で分類出来ます. 一つは微分演算子の様に引数の前に置かれる演算子, この演算子を前置表現の演算子と呼びます. 次は積の様に二つの引数の間に置かれる演算子で, この演算子の事を内挿表現の演算子と呼びます. そして階乗の様に引数の後の置かれる演算子で, 後置表現の演算子と呼びます. 更に引数を持たない演算子もあります.

Maxima の面白い点は, 利用者が必要に応じて演算子の定義が行える事です. 演算子の定義手順は, 演算子が前置表現, 内挿表現, 後置表現, 或いは無引数かどうかを宣言し,

それから演算子本体を Maxima の関数として定義する方法, 或いは, 最初に関数を先に定義して, その関数が演算子であると定義する方法があります. ここで Maxima の簡単な関数の定義方法は:=を用いる方法です.

では, 実際に和の演算子+の様な nary 型の内挿表現演算子 mike を実際に定義してみましょう.

```
(%i1) nary("mike");
(%o1)                                     "mike"
(%i2) x mike y:=x*y*sin(sqrt(x^2+y^2));
                                     2      2
(%o2)          x mike y := x y sin(sqrt(x  + y ))
(%i3) 10 mike 5;
                                     3/2
(%o3)          50 sin(5      )
```

この例では `nary("mike")` で mike が nary 型の内挿表現の演算子であると宣言しています. ここで重要な事ですが, nary 関数を用いる前に関数を生成しておく必要はありません. 上の例に示した様に演算子としての宣言を行った上で, 演算子の本体を定義しても構いません. 但し, 演算子本体の定義は予め演算子として宣言するかどうかで, その書式が異なります. この例では nary 演算子としての属性を与えてしまったので, 通常の関数の書式ではなく, 内挿表現の演算子の書式で定義しなければなりません, そこで, 関数定義の演算子:=の左辺を `x mike y` とし, 右辺に処理の内容を記述しています. 最後に `10 mike 5` で演算子 mike による計算を実行しています.

参考迄に関数を定義し, それから関数に演算子の属性を与える方法で演算子 mike を定義した例を示しておきます.

```
(%i1) mike(x,y):=x*y*sin(sqrt(x^2+y^2));
                                     2      2
(%o1)          mike(x, y) := x y sin(sqrt(x  + y ))
(%i2) nary("mike");
(%o2)          mike
(%i3) 10 mike 5;
                                     3/2
(%o3)          50 sin(5      )
```

演算子には nary 型の内挿表現の演算子に加え, infix 型の内挿表現の演算子, 前置 (prefix), 後置 (postfix), matchfix 型と引数無し (nofix) の演算子があります. 被演算子の個

数は, 前置表現と後置表現の演算子は一つ, 内挿表現の演算子は二つ, matchfix 型は任意, 引数無しの場合は零個になります.

では, 前置表現と後置表現の演算子を実際に定義してみましょう.

```
(%i7) prefix("tama");
(%o7)
"tama"
(%i8) tama x:=2^x;
(%o8)

$$\text{tama } x := 2^x$$

(%i9) tama 3;
(%o9)
8
(%i10) postfix("pochi");
(%o10)
"pochi"
(%i11) x poch := x/10;
(%o11)

$$x \text{ poch} := \frac{x}{10}$$

(%i12) 2 poch;
(%o12)

$$\frac{1}{5}$$

```

次に $2 \cdot 3^5 / 4 - 1$ と数式を記述すれば Maxima では $((2 * (3^5)) / 5) - 1$ が入力されたと解釈されます. これを表現する為に, 二つの Maxima の演算子が一つの被演算子異を挟む場合, どちらの演算を優先させるかを数値で表現しています.

少し考えてみましょう. 数学の計算で, 数式 $2^3 \times 4 + 1$ は $((2^3) \times 4) + 1$ と解釈し, $2(3 \times (4 + 1))$ とは解釈しません. 通常の数式では冪が最も強く, その次に商, 積, 差と最後に和になります. Maxima では, この演算子が被演算子を引き付ける力の事を束縛力と呼び, 200 以下の数値でこの力の大きさを表現します. そして, 二つの演算子が一つの被演算子を共有する場合, この束縛力の大きな演算子が優先される仕組みになっています. ここで演算子の宣言で束縛力の大きさを設定しなければ, 初期値として 180 が設定されます. この束縛力は左右別々に設定可能です. 因に, 和+は左右の束縛力が 100, 積*は左束縛力が 120 で右束縛力が未設定, 冪^は, 左が 140 で右が 139 となっています.

では, 具体的な例で束縛力を確認してみる事にしましょう.

```
(%i29) prefix("test:")$
```

```
(%i30) test:2+3-test:(2+3);
(%o30)                - test: 5 + test: 2 + 3
(%i31) prefix("test:",90)$
(%i32) test:2+3-test:(2+3);
(%o32)                test: 5 - test: 5
(%i33) %-test:(5-test:5);
(%o33)                0
```

この例では前置表現の演算子 `test:` を定義しています。この演算子の場合、被演算子は左側に配置する為、左束縛力のみが設定可能です。最初の定義では左束縛力を設定していない為、初期値として 180 が自動的に設定されます。この束縛力は和や差と比べて格段に大きい為、`test:2+3` は `(test:2)+3` として Maxima は評価します。

演算子の束縛力の変更は再度、演算子を宣言する関数で束縛力を設定し直せばよいので、今度は `prefix` 関数を使って束縛力を和や差よりも小さな 90 に設定します。すると、`test:2+3-test:(2+3)` で最初の `test:` の直後の 2 は和の演算子 `+` に引き寄せられるので、`2+3` が優先して処理されます。そして、次の差の演算子 `-` よりも演算子 `test:` の方が束縛力が今度は弱いので、`2+3` は差の演算子に引き寄せられてしまい、結局、`test:2+3-test:(2+3)` は `test:((2+3)-test:(2+3))` として解釈されます。

この様な形で自分の演算子が簡単に定義出来ます。又、演算子の定義に加えて Maxima の規則を上手く使えば、より高度な処理が容易に行えます。

2.5 式の操作

Maxima では C や FORTRAN の数式が殆どそのまま使えます。

```
(%i10) (1+2+3*x)^2*(1+y)/(z-1);
                                2
                                (3 x + 3) (y + 1)
(%o10) -----
                                z - 1
(%i11) a:sin(x)*cos(y)-x^2+2-(1+u^2);
                                2      2
(%o11)      sin(x) cos(y) - x  - u  + 1
(%i12) a^2;
                                2      2      2
(%o12)      (sin(x) cos(y) - x  - u  + 1)
```

先ず, 式を括る場合は小括弧 () を用います. ところで, 変数に値や式を割当てる場合は C や FORTRAN とは異なり演算子: を用います. 多くの数式処理では演算子:= が割当に利用されていますが, この演算子を Maxima では関数の定義に用います. 又, 演算子= は等しいかどうかを検出する比較の演算子で, C の == と同じ意味を持ちます. この辺は他の多くの数式処理と勝手が違うので注意が必要です.

上の例で示す様に Maxima では入力式をある程度は自動的に簡易化しますが, 式の展開は通常行いません. この様な展開や簡易化はそれらに関連した大域変数を調整する事で, 或る程度の自動化が行えます. 但し, 本格的な式の展開や簡易化では, expand 関数や ev 関数等の専用の関数や文脈といった Maxima の仕組みを利用して行う必要があります.

先ず, 式の展開は expand 関数で出来ます.

```
(%i13) (x+1)*(x-1)+(y+1)^2;
                2
(%o13)      (y + 1) + (x - 1) (x + 1)
(%i14) expand((x+1)*(x-1)+(y+1)^2);
                2          2
(%o14)      y  + 2 y + x
```

これで簡単な式になる事も多くありますが, 一方で, 下手に式を展開した為に項が沢山出過ぎて, 見通しが悪くなる事もあります. その様な場合には, factor 関数や ratsimp 関数等の式を纏める関数を使います.

```
(%i8) 2*x^2+4*x^3+x^4-2*x^5-x^6;
                6      5      4      3      2
(%o8)      - x  - 2 x  + x  + 4 x  + 2 x
(%i9) factor(%);
                2      2      2
(%o9)      - x  (x + 1) (x - 2)
(%i10) 1/(x+1)+x^2/(x^2-1);
                2
                x      1
(%o10)      ----- + -----
                2      x + 1
                x  - 1
(%i11) ratsimp(%);
```

```

              2
            x  + x - 1
(%o11) -----
              2
            x  - 1

```

手計算で一番苦勞する事が式を簡単にする事です. 数式処理でも或る程度は自動的に行えますが, より簡易な式が必要になれば, Maxima に色々な指示を与える必要があります.

一般的に式を簡単にする方法の幾つかを簡単に説明しましょう. 先ず, 括弧で括られた式を展開する方法. これは `expand` 函数を用います. これで式が括弧を持たない平坦な式になります. 展開する事で式が簡単になる事もありますが, かえって項が増えて見通しが悪くなる事もあります. しかし, 式を簡単にする函数の多くが `expand` 函数を使って式を展開する事が前提になっています. 次に, 式を共通の因子で纏めて簡単にする操作, 即ち, 因子分解を Maxima では `factor` 函数で行います. 更に, 与えられた式が有理数係数の多項式や函数式の分母を持つ場合は, まだ物足りません. この場合は, `ratsimp` 函数を用います. 更に, 式に三角函数や指数函数が含まれている場合, `trigsimp` 函数を使えば $\cos^2(x) + \sin^2(x) = 1$ 等の三角函数や指数函数の代表的な公式を使って式が簡易化されます. この様に, 様々な函数を適宜組合せて式を簡単にして行きます.

ところで, $\sqrt{x^2}$ の様な式では, 変数 x の値によって x や $-x$ の様に, より簡単な式にする事が可能です. 変数 x が 0 以上であるとか 0 以下であるといった情報は `assume` 函数を使って Maxima に教えてやる事が出来ます.

では Maxima の二つの式 `sqrt(x^2)` と `sqrt(y^2)` に対し, `assume` 函数を使って変数の条件を教えた場合にどんな事が生じるかを観察してみましょう. ここでは変数 x に対してのみ $x > 0$ を仮定します.

```

(%i1) assume(x>0);
(%o1) [x > 0]
(%i2) sqrt(x^2);
(%o2) x
(%i3) sqrt(y^2);
(%o3) abs(y)

```

この例で示す様に, 変数 x に条件が入った事で, Maxima の式 `sqrt(x^2)` が x に自動的に簡易化されましたが, `sqrt(y^2)` の変数 y には何等の条件も無いので `abs(y)` が返却されていますね.

Maxima では、この様な仮定のあつまりの事を文脈 (context) と呼びます。これは Mathematica にある文脈と同等のものです。一般の式の簡略化は実は文脈上で実行されています。ここで初期状態の文脈は initial という名前の文脈です。Maxima では文脈を複数用いる事も可能で、利用者が自分専用の文脈を生成して、そこで式等処理する事も出来ます。

式の簡易化では文脈だけではなく簡易化を行う関数の動作を制御する様々な Maxima の大域変数を調整する必要がある事もあります。この様な処理に長じているのが ev 関数です。

この関数は非常に機能が高いので、詳細は §5.8.2 を参照して下さい。以下に簡単ですが、ev 関数の代表的な使い方を幾つか示しておきましょう。

```
(%i9) A:sin(3*x)-cos(3*y)*z;
(%o9)
      sin(3 x) - cos(3 y) z
(%i10) ev(A,trigexpand=true,trigexpandtimes=true,eval,z=2);
      3      2      3      2
(%o10) - 2 (cos (y) - 3 cos(y) sin (y)) - sin (x) + 3 cos (x) sin(x)
(%i11) B:sin(2*x)-cos(3*y)*(z+1)^3;
      3
(%o11) sin(2 x) - cos(3 y) (z + 1)
(%i12) B,trigexpand=true,trigexpandtimes=true,eval,z=2;
      3      2
(%o12) 2 cos(x) sin(x) - 27 (cos (y) - 3 cos(y) sin (y))
```

この例では、ev 関数を用いて式 A を三角関数の倍角公式を使って展開し、変数 z に 2 を代入する例と、ev 関数を表に出さずに同様の処理を行う例です。ev 関数は ev(< 与式>,< 条件₁>,...,< 条件_n>) の書式になり、与式に対して後続く条件を適用して行きます。Maxima の最上層 (プロンプトが (%i 番号) となっている場所) では、ev 関数を省略して中身だけを記述しても構いません。

更に、変数の属性を与えたり、定義した関数の性質を付加する必要がある事もあります。Maxima には対象を定めて規則を定義し、その規則を式に対して適用する事も出来ます。

```
(%i19) matchdeclare(_a,true);
(%o19)
      done
(%i20) let(tama(_a)^2,tama(2*_a)+1);
```

```

                2
(%o20)          tama (_a) --> tama(2 _a) + 1
(%i22) letsimp(expand((1+tama(x))^2));
(%o22)          tama(2 x) + 2 tama(x) + 2

```

この例では, `tama` という名前の実体を持たない関数 (要するに未定義の関数) に対して, $\text{tama}(x)^2$ を $\text{tama}(2x)+1$ で置換えさせる規則を定義し, `letsimp` 関数を使って実際の式に適用させたものです. この様な規則を定める関数には他に `defrule` 関数があります. この規則の詳細に関しては §5.7 を参照して下さい.

2.6 式の微分・積分

式の微分は `diff` 関数で行います. 微分は積分と比較して非常に機械的な操作になるので, その結果は信頼出来ます, 以下に簡単な式の微分の例を示しておきましょう.

```

(%i5) diff(x*y*sin(sqrt(x^2+y^2)),x);
                2          2      2
                2      2      x y cos(sqrt(y  + x ))
(%o5)          y sin(sqrt(y  + x )) + -----
                                   2      2
                                   sqrt(y  + x )

(%i6) diff(x*y*sin(sqrt(x^2+y^2)),x,n);
          n
          d
          --- (x y sin(sqrt(y  + x )))
          dx

(%i7) 'diff(x*y*sin(sqrt(x^2+y^2)),x,n);
          n
          d
          --- (x y sin(sqrt(y  + x )))
          dx

```

この例では函数 $xy \sin(\sqrt{x^2 + y^2})$ の一階微分と n 階微分を計算しています. 但し, n 階微分は n の値が不明な為, そのままの形で返却されています. 最後の例では演算子 $'$ を項の頭に付けています. この記号が付けられた項の評価を Maxima は行いません. この様に演算子 $'$ が付けられた Maxima の式の項の事を Maxima では名詞型と呼びます. 猶, 項の先頭に演算子 $'$ を付けると評価しない方式は, 数式処理の Maple や *Mathematica* でも同様です.

通常の不定積分は `integrate` 函数を使います. この `integrate` 函数で物足りない場合, Risch の積分アルゴリズムに基いた `risch` 函数があります. 但し, Risch の積分手法は Maxima にはまだ完全に実装されていません.

```
(%i18) integrate((sqrt(x^2+1)),x);
```

$$\frac{\operatorname{asinh}(x)}{2} + \frac{x \sqrt{x^2 + 1}}{2}$$

```
(%o18)
```

```
(%i19) diff((%o18),x);
```

$$\frac{\sqrt{x^2 + 1}}{2} + \frac{x}{2 \sqrt{x^2 + 1}} + \frac{1}{2 \sqrt{x^2 + 1}}$$

```
(%o19)
```

```
(%i20) ratsimp(%);
```

$$\sqrt{x^2 + 1}$$

```
(%o20)
```

```
(%i21) integrate((sqrt(x^2+1)),x,1,2);
```

$$\frac{\operatorname{asinh}(2) + 2 \sqrt{5}}{2} - \frac{\operatorname{asinh}(1) + \sqrt{2}}{2}$$

```
(%o21)
```

```
(%i22) defint((sqrt(x^2+1)),x,1,2);
```

$$\frac{\operatorname{asinh}(2) + 2 \sqrt{5}}{2} - \frac{\operatorname{asinh}(1) + \sqrt{2}}{2}$$

```
(%o22)
```

積分の場合は微分と比べて注意が必要になります. これは積分が非常に難しい問題だからです. その為, 最低でも積分した結果は, 微分して積分する前と一致するかどうか

を確認する事を強く薦めます. 更に, 積分する前と後の函数のグラフを描いて視覚的にも確認する事も薦めます. これは本来なら連続な函数が積分すると何故か非連続な函数になる事があるからです. この点については後の章でも解説します.

2.7 方程式の解

Maxima で方程式は $x^2+2*x+1=0$ の様に二項演算子=で繋がれた式で表現します. 更に, Maxima では連立方程式も扱えます. この場合, [eq1, eq2] の様に方程式で構成された平坦なリストを用いて連立方程式を表現します.

代数方程式の厳密解を求める場合, solve 函数 が使えます.

```
(%i25) solve(x^2+2*x+1,x);
(%o25) [x = - 1]
(%i26) solve([x^2+2*x*y+y^2+1,y+2*x-1],[x,y]);
(%o26) [[x = 1 - %i, y = 2 %i - 1], [x = %i + 1, y = - 2 %i - 1]]
```

猶, 実数値のみで十分な場合は, realroots 函数が使えます.

```
(%i3) realroots(x^8+x^3+x-5);
(%o3) [x = -  $\frac{43873333}{33554432}$ , x =  $\frac{37579631}{33554432}$ ]
```

連立方程式を解く場合, linsolve 函数や algsys 函数が使えます. 特に, algsys 函数は厳密解が計算可能なら厳密解を計算し, 困難であれば数値近似解を求める函数です.

```
(%i31) eq1:[x^2+y*x-1,x*y^2-x+y];
(%o31) [x y + x2 - 1, x y2 + y - x]
(%i32) ans:algsys(eq1,[x,y]);
```

```

                                sqrt(sqrt(2) + 2)
(%o32) [[x = sqrt(sqrt(2) + 2), y = - -----],
                                sqrt(2)
                                sqrt(sqrt(2) + 2)
[x = - sqrt(sqrt(2) + 2), y = -----],
                                sqrt(2)
                                sqrt(2 - sqrt(2))
[x = - sqrt(2 - sqrt(2)), y = - -----],
                                sqrt(2)
                                sqrt(2 - sqrt(2))
[x = sqrt(2 - sqrt(2)), y = -----]]
                                sqrt(2)

(%i33) eq1,ans[1]$
(%i34) trigsimp(%);
(%o34) [0, 0]

```

この例では連立方程式 $x^2 + yx - 1 = 0, xy^2 - x + y = 0$ を `algsys` 関数を用いて解き、4組の解の中の最初の組を使って検算を実行しています。ここでの `eq1,ans[1]` は `ev(eq1,ans[1])` の別の表記方法で、連立方程式 `eq1` の各式に解のリスト `ans` の最初の成分の値を変数 `x,y` に代入して、式の評価を行っています。

Maxima では常微分方程式も扱えます。例えば、バネの微分方程式 $\frac{d^2x(t)}{dt^2} + Kx(t) = 0$ を解いてみましょう。

```

(%i48) ode2('diff(x(t),t,2)+K*x(t)=0,x(t),t);
Is K positive, negative, or zero?

pos;
(%o48) x(t) = %k1 sin(t sqrt(K)) + %k2 cos(t sqrt(K))
(%i49) ic2(%o36,t=0,x(t)=0,'diff(x(t),t)=10);
10 sin(t sqrt(K))
(%o49) x(t) = ----- + x(0) cos(t sqrt(K))
sqrt(K)

```

ここでは常微分方程式を解く為に、`ode2` 関数 を用いています。この例の様に `ode2` 関数は必要に応じて利用者に式の正負を尋ねてきます。`ode2` 関数の出力で、`%k1` と `%k2` がありますが、これらは `ode2` 関数が定めた定数です。初期条件を与えて解く事も出来ま

す. 二階の常微分方程式では初期条件は `ic2` 関数や `bc2` 関数 で与えます. この例では $x(0)=0, x'(0)=10$ としています.

2.8 行列

Maxima で扱える行列の大きさや行列の成分に制限は特にありません. 数値行列でも多項式の行列でも扱えます. 但し, 行列の演算は MATLAB の様な数値行列処理ソフトと比較して速いものではありません. その為, 数値行列の計算で多倍長浮動小数点数 (`bigfloat` 型の行列) で計算するのが必要がなければ, MATLAB の様な数値行列ソフトの利用を強く薦めます. 猶, この本では §15 で GNU の MATLAB と言える Octave の利用に触れているので, Octave に興味があれば参照して下さい.

Maxima で行列の定義には幾つかの方法があります. 一般的な方法は `matrix` 関数を用いて定義する方法です.

```
(%i13) A:matrix([1,2,3],[4,3,2],[5,1,3]);
          [ 1  2  3 ]
          [      ]
(%o13)     [ 4  3  2 ]
          [      ]
          [ 5  1  3 ]

(%i14) B:matrix([10,9,8],[1,2,3],[2,4,6]);
          [ 10  9  8 ]
          [      ]
(%o14)     [ 1  2  3 ]
          [      ]
          [ 2  4  6 ]
```

同じ大きさの行列の和や差は演算子 `+` と `-` で行えます. 但し, 可換積 `*` とその冪 `^` は, 行列に対しては, 通常の行列の積や冪ではなく, 各成分同士の可換積や成分単位の冪となるので注意しましょう.

(%i15) A+B;

```
[ 11  11  11 ]
[          ]
[  5   5   5 ]
[          ]
[  7   5   9 ]
```

(%o15)

(%i16) A*B;

```
[ 10  18  24 ]
[          ]
[  4   6   6 ]
[          ]
[ 10   4  18 ]
```

(%o16)

(%i17) A^2;

```
[  1   4   9 ]
[          ]
[ 16   9   4 ]
[          ]
[ 25   1   9 ]
```

(%o17)

行列の積は非可換です. そこで Maxima では行列の積に非可換演算子, その冪に非可換演算子の冪を用います.

(%i18) A . B;

```
[ 18  25  32 ]
[          ]
[ 47  50  53 ]
[          ]
[ 57  59  61 ]
```

(%o18)

(%i19) A^^2;

```
[ 24  11  16 ]
[          ]
[ 26  19  24 ]
[          ]
[ 24  16  26 ]
```

(%o19)

(%i20) A . A^^(-1);

```
(%o20)      [ 1  0  0 ]
             [      ]
             [ 0  1  0 ]
             [      ]
             [ 0  0  1 ]
```

逆行列の計算では `invert` 関数 を用いたり, 例の様に演算子`^^` を使って計算する事も可能です.

```
(%i21) invert(A);

             [  7   1   1 ]
             [ - --  --  - ]
             [ 30  10   6 ]
             [      ]
             [  1   2   1 ]
(%o21)      [ --  -  - - ]
             [ 15   5   3 ]
             [      ]
             [ 11   3   1 ]
             [ --  - --  - ]
             [ 30   10  6 ]
```

```
(%i22) A^^(-1);

             [  7   1   1 ]
             [ - --  --  - ]
             [ 30  10   6 ]
             [      ]
             [  1   2   1 ]
(%o22)      [ --  -  - - ]
             [ 15   5   3 ]
             [      ]
             [ 11   3   1 ]
             [ --  - --  - ]
             [ 30   10  6 ]
```

Maxima で行列の固有値の計算は eigen パッケージを用います.

```
(%i5) load(eigen);
(%o5) /usr/local/share/maxima/5.9.2/share/matrix/eigen.mac
(%i6) A:matrix([1,2,0],[2,1,1],[0,1,1]);
      [ 1  2  0 ]
      [      ]
(%o6) [ 2  1  1 ]
      [      ]
      [ 0  1  1 ]

(%i7) eigenvalues(A);
(%o7) [[1 - sqrt(5), sqrt(5) + 1, 1], [1, 1, 1]]
(%i8) eigenvectors(A);
      sqrt(5)  1
(%o8) [[[1 - sqrt(5), sqrt(5) + 1, 1], [1, 1, 1]], [1, - ----, -],
      2      2
      sqrt(5)  1
      [1, ----, -], [1, 0, - 2]]
      2      2

(%i9) charpoly(A,t);
      2
(%o9) ((1 - t) - 1) (1 - t) - 4 (1 - t)
```

ここでの例では最初に load 関数で eigen パッケージの読込を実行します. 固有値の計算は eigenvalues 関数を用い, 固有ベクトルの計算は eigenvectors 関数が使えます. 猶, 特性多項式の計算は charpoly 関数で行えます.

2.9 FORTRAN や T_EX への出力

Maxima は計算した結果を FORTRAN の書式や T_EX のソースに変換する事が出来ます. 先ず, FORTRAN の書式に式を変換する関数は fortran 関数です. 次に TeX への変換を行う関数は tex 関数です. これらの関数は変換すべき式のみを引数に取ります.

```
(%i12) expr1:expand((x+2*y)^5);
      5      4      2 3      3 2      4      5
(%o12) 32 y  + 80 x y  + 80 x y  + 40 x y  + 10 x y  + x
```

```
(%i13) fortran(expr1);
      32*y**5+80*x*y**4+80*x**2*y**3+40*x**3*y**2+10*x**4*y+x**5
(%o13)                                     done
(%i14) tex(expr1);
$$$32\,y^5+80\,x\,y^4+80\,x^2\,y^3+40\,x^3\,y^2+10\,x^4\,y+x^5$$$
(%o14)                                     false
```

最初の fortran 関数の結果から判る様に, 冪は**で置換えます. この様に, そのままでは FORTRAN で扱えない演算や表記が与式に含まれていれば, それらを FORTRAN で利用可能な書式に変換します. 但し, この置換が無理な項があれば, その項をそのまま含めて出力します.

tex 関数は与式を T_EX のソースコードに変換し,\$で括ったものを返却します. その為, 実際に利用する場合, 両端の\$を削除して使います. 以下にその結果を示します.

tex 関数で変換した結果を利用

$$32y^5 + 80xy^4 + 80x^2y^3 + 40x^3y^2 + 10x^4y + x^5$$

一寸した工夫になりますが, 多項式を FORTRAN の書式に変換する時に horner 関数を使って, 与式を Horner 則で式を纏めると, 式に含まれる積の回数を減らす事が出来るので数値計算の高速化で大きな効果が得られます.

```
(%i27) horner(expr1);
                                     2      3      4      5
(%o27)      y (y (y (y (32 y + 80 x) + 80 x ) + 40 x ) + 10 x ) + x
(%i28) fortran(%);
      y*(y*(y*(y*(32*y+80*x)+80*x**2)+40*x**3)+10*x**4)+x**5
(%o28)                                     done
(%i29) tex(%o27);
$$$y\,\left(y\,\left(y\,\left(y\,\left(32\,y+80\,x\right)+80\,x^2\right)+40\,x^3\right)+10\,x^4\right)+x^5\leqno{\tt (\%o27)}$$$
(%o29)                                     (\%o27)
```

では, Octave を使ってその効果を確認しましょう.

```
octave:20> x=10.1;y=12.034;
octave:21> a1=time();32*y**5+80*x*y**4+80*x**2*y**3+40*x**3*y**2+
10*x**4*y+x**5;time()-a1
```

```
ans = 0.00014305
octave:22> a1=time();y*(y*(y*(y*(32*y+80*x)+80*x**2)+40*x**3)+10*x**4)+
x**5;time()-a1
ans = 0.00013208
octave:23>
```

この例では time 関数を使って計算に要した時間を計測する事で計算速度の違いを調べています. ここで用いた式は, 行列でもない5次の多項式の計算ですが, Horner 則で変形した式の方が数値計算が速くなっている事が判ります.

FORTRAN 関数は Maxima で処理した結果を FORTRAN のソースに変換する関数ですが, Maxima には利用者が定義した関数を LISP の関数に変換する translate 関数, 更に, translate 関数で変換した関数を LISP コンパイラを用いてコンパイルする compile 関数もあります. 詳細は §7.3 を参照して下さい.

2.10 グラフ表示

Maxima は幾つかのグラフ表示の命令を持っており, 二次元グラフや三次元グラフの表示が可能です. これらの関数の中で最も多機能なものとしては, plot2d 関数と plot3d 関数, そして, draw パッケージに含まれる draw2d や draw3d 関数が挙げられます. ここで plot2d 関数と plot3d 関数は Maxima に古くからある描画関数で, 様々な外部の描画アプリケーションが使えます. それに対し, draw パッケージ自体が Maxima の新しいパッケージで, このパッケージに収録された関数は gnuplot(4.2 以上) のみに対応しています.

ここでは幅広い環境で, 直感的に使える plot2d 関数と plot3d 関数について簡単に述べる事にします.

先ず, plot2d 関数, plot3d 関数は与えられた式から数値データを生成してファイルに書き出し, それを描画用の外部アプリケーションに引き渡すといった処理を行う関数です. 共に Maxima-5.9.0 迄は外部の描画用アプリケーションとして openmath を標準で用いていました. その為, openmath と maxima の親和性は高く, xmaxima の入力副ウィンドウ内にグラフを表示させる事が可能です. この様子を図 2.4 に示しておきます. この openmath は Maxima のソースファイルにも附属しており, 別途入手する必要がありません.

現在は, gnuplot が標準の描画用アプリケーションとなっており, 5.10.0 以降は gnuplot との親和性も格段に向上しています.

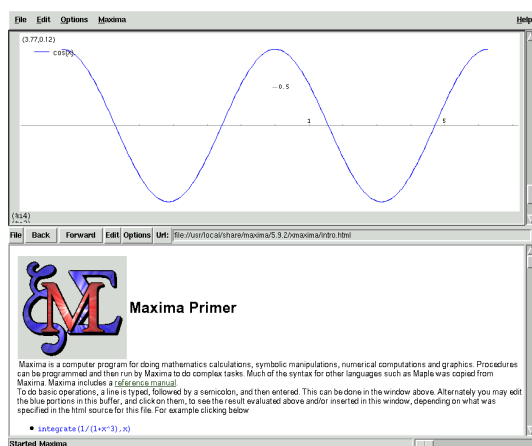


図 2.4: openmath+xmaxima による余弦関数のグラフ

さて、二次元グラフ表示は、`plot2d` 関数を用いて、通常の $y = f(x)$ の関数表示とパラメータ表示の両方で関数が描けます。

例えば、`plot2d(sin(x), [x, -2*%pi, 2*%pi]);` と入力すると、xmaxima 上で openmath を利用する時以外は、別ウィンドウが開かれて、そこにグラフが表示されます。

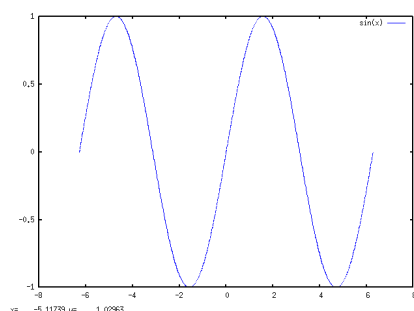


図 2.5: gnuplot による正弦関数の表示

パラメータ表示を行う事も可能です。`[cos(t), sin(t) cos(t)]` のグラフを描いてみましょう。但し、曲線の点数が少ないと綺麗な絵が描けない事があります。この場合、`set_plot_option` 関数で、`nticks` を適切な値に変更します。この図では、100 にしてみましょう。

```
(%i35) set_plot_option([nticks,100]);
```

```
(%o35) [[x, - 1.755559702014E+305, 1.755559702014E+305],
[y, - 1.755559702014E+305, 1.755559702014E+305], [t, - 3, 3],
[grid, 30, 30], [view_direction, 1, 1, 1], [colour_z, false],
[transform_xy, false], [run_viewer, true], [plot_format, gnuplot],
[gnuplot_term, default], [gnuplot_out_file, false], [nticks, 100],
[adapt_depth, 10], [gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]], [gnuplot_curve_styles,
[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,
with lines 6, with lines 7]], [gnuplot_default_term_command, ],
[gnuplot_dumb_term_command, set term dumb 79 22],
[gnuplot_ps_term_command, set size 1.5, 1.5;
set term postscript eps enhanced color solid 24]]
(%i36) plot2d([parametric,cos(t),cos(t)*sin(t),[t,-5,5]], [x,-3,3]);
```

この設定で `plot2d([parametric,cos(t),cos(t)*sin(t),[t,-5,5]], [x,-3,3]);` を実行した結果が図 2.6 になります。

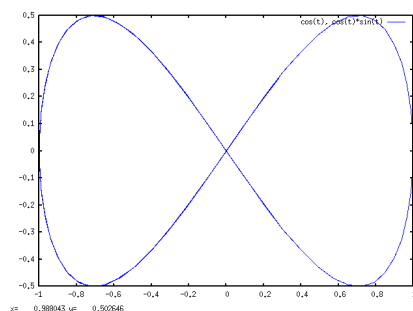


図 2.6: gnuplot によるリサージュ曲線の表示

三次元グラフ表示は、の場合は `plot3d` 函数を用います。ここでは、マニュアルにもあるクラインの壺を代表的な外部アプリケーションの `gnuplot`, `openmath` と `Geomview` で描いてみましょう。入力するのは以下の式です。

Klein の壺

```
plot3d(
[5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0)-10.0,
-5*sin(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0),
5*(-sin(x/2)*cos(y)+cos(x/2)*sin(2*y))],
[x,-%pi,%pi],[y,-%pi,%pi],[grid,40,40]);
```

Klein の壺の絵は壺の首が自分自身に交差している様に見える図 2.7 をご覧になった

方が多いでしょう.

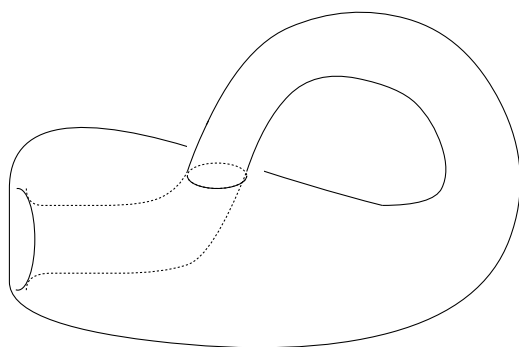


図 2.7: Klein の壺

正確に言えば, Klein の壺の自己交差の近傍は時間が違う個所に存在するので, 4 次元空間では自己交差無しで描けます.

この Klein の壺は 2 個の Möbius の輪をその境界で貼合せる事でも得られます. この曲面は向付けが出来ない曲面, 即ち, 裏表が無い曲面の有名な例です. 特に, Klein の壺の様に, 境界を持たない曲面を三次元空間内部で表現しようとする時, 自分自身が交わる個所がどうしても出てきます. 上の表記は Möbius の輪を二つ貼り合せた状態が判り易いもので, ドーナツ表面 (トーラス $(S^1 \times S^1)$ と呼びます) を捻じった状態になっています.

2.10.1 gnuplot による Klein の壺

gnuplot は Maxima で利用可能な外部表示アプリケーションの中では、最も高機能で、良好な画が表示可能なものの一つです。更に、gnuplot はグラフ表示のアプリケーションとして利用可能なだけでなく、一寸したグラフ計算機のように利用する事さえも可能な程です。

gnuplot は MS-Windows 版の Maxima にも標準で附属しており、openmath と同様に即座に利用する事が可能です。この gnuplot を用いる場合、MS-Windows 版とそれ以外の環境では多少使い勝手が異なります。先ず、MS-Windows 版では、曲面は面が張られた状態で表示され、マウスで図形の拡大や縮小が行えますが、その他の環境ではワイヤーフレームで表示され、Maxima-5.12.0 よりも昔の Maxima ではマウスによる直接操作が出来ません。これは Maxima-5.12.0 以降で、MS-Window 環境以外で gnuplot の扱いそのものが異なる為です。

gnuplot の描画の終了はウィンドウ上で q を入力して下さい。

```
set_plot_option([plot_format,gnuplot])
```

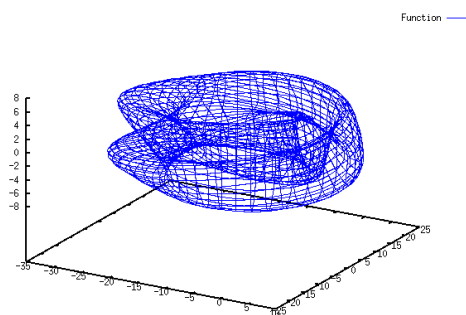


図 2.8: gnuplot による Klein の壺

2.10.2 openmath による Klein の壺

openmath は割と高機能で, そこそこ綺麗な描画が出来ます. こちらは標準でパッケージに附属しています. 視点の変更はマウスで直接操作が出来ます. ウィンドウ左上 (立ち上げ時には Menu Here と表示されており, その後は座標値が表示されている個所) にマウスを移動させ, マウスの左ボタンをクリックすると, メニューが出て来ます. 表示を止めたければ, このメニューの最上段の Dismiss を選択します

```
set_plot_option([plot_format,openmath])
```

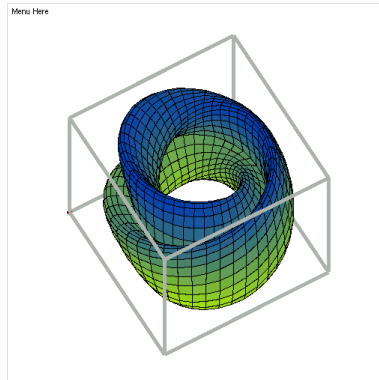


図 2.9: openmath による Klein の壺

2.10.3 Geomview による Klein の壺

Geomview はミネソタ大学の Geometry Center で開発されたツールで, plot3d が扱える外部アプリケーションの中では最も大規模で, その描画も美しいものです. 猶, Geomview は UNIX 環境のみでしか使えません.

描画を開始するとグラフと 2 個の制御パネルが現われ, 視点の変更は長細い制御ウィンドウを用います. 終了は別のウィンドウの File メニューから Exit を選択します.

```
set_plot_option([plot_format,geomview])
```

この Geomview はグラフ表示専門のアプリケーションというよりは, 寧ろ, 様々な幾何学的対象の可視化ツールといった方が良いものです. 更に, モジュールで機能の拡張

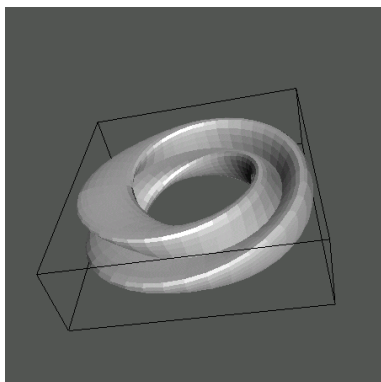


図 2.10: Gemoview による Klein の壺

が行えます. 公開されているモジュールには面白いものが幾つかあります. 例えば, 図 2.11 に示す太陽系シミュレーター Orrey もその一つです.

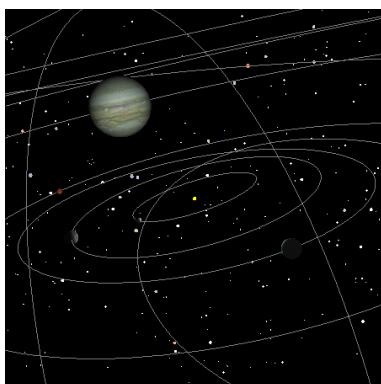


図 2.11: 太陽系シミュレーター Orrey

ここでの例に示す様に, グラフの表示では `set_plot_option` 関数を用いて, 表示アプリケーションを目的に適合したものに変更する等の設定を行えば良いのです. 又, グラフ表示も総合的な機能では商用のものに劣る面もありますが, 実用上問題は殆ど無いでしょう.

2.10.4 番外編

さて、グラフ表示に関して簡単に解説しましたが、Maxima でやっている事は、結局のところ、グラフデータを生成して、外部のアプリケーションに引渡す事です。

従って、あるグラフ表示アプリケーションの入力データ書式が判っている場合、Maxima でそのデータを生成してファイル出力する事さえ出来てしまえば、後は system 関数を使って、そのアプリケーションにデータファイルを読込ませてしまえば良いのです。この手法の簡単な例として、§14 に surf を用いた代数曲線や曲面の可視化について解説しています。

2.11 ファイル

Maxima は Common Lisp で記述されている為、基本的に入出力は LISP の入出力関数を利用したものです。その為、LISP の入出力関数の縮小版の傾向があります。

まず、ファイルに画面と同じ出力を行いたければ、writefile 関数を用います。この writefile 関数は LISP の dribble 関数を用いた関数で、入力と出力をそのまま指定したファイルに保存します。但し、writefile 関数は指定したファイルを新規に生成する為、単純に既存のファイルに記録したければ、appendfile 関数を用います。writefile 関数や appendfile 関数で開いたファイルを閉じる場合は closefile() で開いたファイルを閉じます。では、早速試してみましょう。

```
(%i1) writefile("maxima.tmp");
(%o1)      #<OUTPUT BUFFERED FILE-STREAM CHARACTER maxima.tmp>
(%i2) integrate(sqrt(x^2+1),x);

                                2
                    asinh(x)  x sqrt(x  + 1)
(%o2)      ----- + -----
                    2          2

(%i3) closefile();
(%o3)      #<CLOSED OUTPUT BUFFERED FILE-STREAM CHARACTER maxima.tmp>
(%i4) ratsimp(diff(%o2,x));

                                2
(%o4)      sqrt(x  + 1)
```

では、次に writefile 関数で指定した maxima.tmp の中身を確認してみましょう。

```

----- maxima.tmp -----
;; Dribble of #<IO TERMINAL-STREAM> started 2005-12-21 06:41:28
(%o1)      #<OUTPUT BUFFERED FILE-STREAM CHARACTER maxima.tmp>
(%i2) integrate(sqrt(x^2+1),x);

              2
      asinh(x)  x sqrt(x  + 1)
      ----- + -----
              2          2

(%o2)

(%i3) closefile();
;; Dribble of #<IO TERMINAL-STREAM> finished 2005-12-21 06:41:44
eegin{verbatim}

```

この様に,writefile 関数を実行した時点から,closefile 関数を実行した時点迄の入出力がそのままファイルに書込まれています.

但し,これらのファイルは実質的に記録ファイルであって,Maxima でそのまま再利用は出来ません. 再利用可能なファイルを生成するのは,save 関数と stringout 関数,そして grind 関数です.

ここで,save 関数は Maxima の内部表現を保存する関数で,load 関数や loadfile 関数を用いて Maxima に読込めます.

これに対して,stringout 関数や grind 関数は式の内部表現ではなく,Maxima の入力にそのまま対応する書式のデータ保存を行います.

ここで内部表現と呼んでいますが,save 関数で生成したファイルの中身はどのようなのでしょうか?早速,調べてみましょう.

それでは,最初に以下の式を入力し,最後に save 関数で,ファイル test に Maxima の内部データ全てを保存します.

```

(%i1) 1+2+3;
(%o1)                                     6
(%i2) a1:x^2+y^2+1;
              2      2
(%o2)      y  + x  + 1
(%i3) resultant(x-t,y-t^2,t);
              2
(%o3)      y  - x
(%i4) save("test",all);
(%o4)      test

```

以下に,test ファイルの内容を先頭の部分だけを示します.

— test ファイルの中身 —

```
;;; -*- Mode: LISP; package:maxima; syntax:common-lisp; -*-
(in-package "MAXIMA")
(DSKSETQ $I1 '((MPLUS) 1 2 3))
(ADDLABEL '$I1)
(DSKSETQ $O1 6)
(ADDLABEL '$O1)
(DSKSETQ $I2 '((MSETQ) $A1 ((MPLUS) ((MEXPT) $X 2)
((MEXPT) $Y 2) 1)))
(ADDLABEL '$I2)
(DSKSETQ $O2 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2)
((MEXPT SIMP) $Y 2)))
(ADDLABEL '$O2)
(DSKSETQ $I3
'((($RESULTANT) ((MPLUS) $X ((MMINUS) $T))
((MPLUS) $Y ((MMINUS) ((MEXPT) $T 2))) $T))
(ADDLABEL '$I3)
(DSKSETQ $O3
'((MPLUS SIMP) ((MTIMES SIMP) -1 ((MEXPT SIMP RATSIMP) $X 2)
$Y))
(ADDLABEL '$O3)
(DSKSETQ $I4 '(($SAVE) &TEST $ALL))
(ADDLABEL '$I4)
(DSKSETQ $A1 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2)
((MEXPT SIMP) $Y 2)))
(ADD2LNC '$A1 $VALUES)
以下略
```

如何ですか？Maxima で入力したもの以外の様々な設定で膨れ上っていますね。それに、(MSETQ) とか何か怪しいものが色々ありますね。

それにしても、この括弧だらけのデータは何でしょう？これは LISP の S 式と呼ばれるデータであり、プログラムでもあります。このファイルの内容を知る為に LISP の大雑把な知識が少しでもあれば十分に役に立ちます。

以上で Maxima の簡単な入門を終えます。次の章では、Maxima の根底にある計算機言語 LISP について簡単ですが紹介しましょう。

第3章 LISPについて

Maxima は LISP で記述されています. その為, Maxima の改造に手を出さなくても, 単純にマニュアルを読む時でさえも, LISP の知識の有無によって理解の度合が異なるのが実情です. この章では LISP について簡単に解説します.

3.1 背景

この本は数式処理システム Maxima の本です. だから Maxima の使い方だけを解説すれば良い筈です. 紙面を括弧だらけの変な言語の解説で紙面の無駄と怒る方も居るかもしれませんが, ここは我慢して暫く私につきあって下さい.

実際, LISP を知っていると, Maxima をより深く理解出来るだけでなく, 一寸した修正さえも出来ます. これは修正したソースファイルを load 関数で読込ませるだけで, コンパイルは不要です. この辺が商用の *Mathematica* や *Maple* とは違った大きな特徴であり, 更には長所になっているのです.

ここでは最初に LISP の概要を, その歴史から簡単に解説しましょう.

LISP の歴史は非常に古く, FORTRAN よりも僅かに新しい言語です. FORTRAN が数値計算向けであるのに対し, LISP (LIStProcessor) はリスト (list) と呼ばれるデータを扱う事を目的としています. LISP には car とか cdr といった一見して意味不明な命令があります. これは LISP の歴史を語っているのです. 最初の LISP は FORTRAN で記述され, IBM の計算機 704 で動作するものでした. car や cdr はその計算機の命令の car (Contents of Address Register) と cdr (Contents of Decrement Register) に由来します. 当時の LISP には, car と cdr の他に cons がある程度の非常に単純なものだった様です. そこから LISP は成長したのです.

LISP のプログラムの一見しただけで分る明瞭な特徴は括弧 () ばかりが目立つ言語でしょう. ところが, データは基本的に原子とリストから構成されたデータ (S 式¹) の為に非常に柔軟に問題に対処可能な事. プログラム自体も S 式であるといった特徴, 更に λ -記法が使えるといった数学基礎論の考え方も実装し易くて美しい言語です. そんな事もあるって, 古い言語でありながら, 何時になっても新しく, 魅力的な言語です. LISP による処理は, インタプリタ言語としても比較的高速な部類に入ります. その上, 問題によっては C よりも効率良く, 高速に処理が行える事もあります.

LISP は言語的に非常に柔軟性が高いので, GNU Emacs や AutoCAD の様に, 独自の LISP を処理言語として用いる事も多々あります². その一方で非常に方言の多い言語でもあります.

現在の様に LISP の標準化が行われる以前の 1970 年代には, MIT の MAC 計画で開発された MACLisp³ と Bolt Beranek and Newman Inc. と Xerox Palo Alto 研究所で開発された InterLisp が主な方言でした. 因に, 1975 年には LISP の主要な方言の一つである Scheme が開発されています.

¹FORTRAN 風の M 式というものもあります

²Emacs の場合は寧ろ LISP で記述されていると言った方が正確でしょう

³MACLisp は Macintosh とは無関係の LISP です. MACLisp は MACSYMA の記述に用いられただけでなく, MACSYMA からのフィードバックもありました

1980年代には Common Lisp: The Language 第一版 (1984) を基に京都大学で KCL (Kyoto Common Lisp) が作成され, これが後に GNU に寄贈されて GCL の原型になっています. 現在の LISP の方言として主要なものは Common Lisp と Scheme の二系統です. 以下の節では Common Lisp の中では非常に多くの計算機環境で利用可能な CLISP を中心に LISP の解説をします. 但し, 非常に初歩的な水準に留める為, GCL その他でも違和感が無いと思います.

3.2 数値, 文字列

では早速 Common Lisp で遊んでみましょう. CLISP 等の Common Lisp が使える環境であれば LISP を立ち上げてみて遊んで下さい. 猶, ここでは CLISP の結果を主に載せています. 又, Maxima がインストールされている環境であれば, 先ず, Maxima を起動して `to_lisp();` と入力して下さい. すると, 普段は後に隠れていた Common Lisp⁴ が表に出てきます.

```
(%i1) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
```

```
MAXIMA> (+ 1 2)
3
MAXIMA> (append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
MAXIMA> (to-maxima)
Returning to Maxima
(%o1)                                     true
```

この例で示す様に, Maxima の `to_lisp` 関数で Maxima の裏で動作する LISP を表に出すと, プロンプトが MAXIMA> に切替わります. 以降, `(to-maxima)` と入力する迄は LISP が表になります. この LISP が表に出た状態で `($quit)` を入力すれば, Maxima を終了するだけではなく, LISP を含めて全体を終えます. この `($quit)` は Maxima 上の `quit();` と同じ意味になりますが, この事は Maxima の式の内部表現に深く関連します (§6.4 参照).

それでは LISP の環境で数値や文字列を入力してみましょう.

⁴Maxima のバイナリ版を入手されている方ならば GCL

```
[1]> 1
1
[2]> 1234
1234
[3]> 12/984
1/82
[4]> 0.01
0.01
[5]> "abc"
"abc"
[6]> "abc def/234"
"abc def/234"
[7]> #\a
#\a
```

この例では整数, 分数, 浮動点小数と文字列と文字の入力を行っています. LISP の分数は分母と分子は整数でなければなりません. 又, 分数は自動的に通分されます. 文字列は単純に二重引用符で括ったものになります. これに対して, 文字の場合は `#\` の様に `#\` を文字の先頭に付けます. LISP では整数等の数値や文字や文字列の事を原子と呼びます. この原子が文字通り, 全てのデータを構成する原子となります.

これは以下の様な有様です:

原子はリストを生成し, リストは S 式を生成し, S 式は万物を生成する

複数の数値の和, 積, 商は LISP ではどの様に表現するのでしょうか. 通常の処理言語では例えば, $1+2+3+4$, $1*2*3*4$ や $1/2/3/4$ と記述しますが, LISP の場合は以下の様になります.

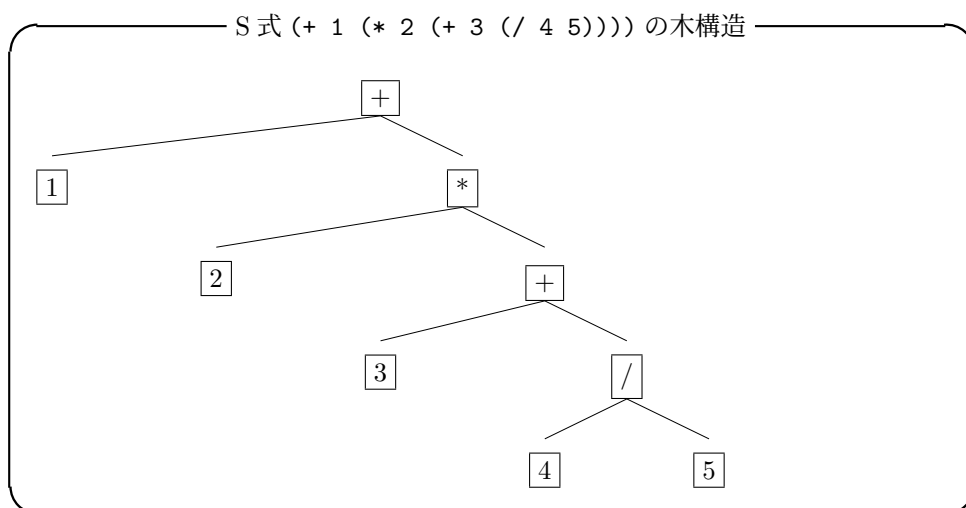
```
[7]> (+ 1 2 3 4)
10
[8]> (* 1 2 3 4)
24
[9]> (/ 1 2 3 4)
1/24
```

LISP ではこの様に演算子を先頭に置きます. この様に演算子や関数を前に置く表現方法の事を前置表現と呼びます. 通常の $1+2$ の様な演算子の置き方を内挿表現と呼び

ます. 参考迄に, 前・内とあれば後置表現もあります. この後置表現で有名な計算機言語に Adobe の PostScript (略して, PS) があります. PostScript は画像データの印象が強いかもしれませんが, 実際はれっきとしたプログラム言語です. 因に, PostScript の様な言語の事をスタック型言語とも呼ぶ事があります. これは手続型言語の一種類ですが, スタックマシンを実行モデルとするものです. 古いものに Forth もあります. 更に KNOPPIX/Math に収録されている kan/sm1 もスタック型言語の一つです.

閑話休題, 上の例では非常に簡単な例を示しましたが, より一般的な式は, 括弧で $1 + 2 * (3 + 4/5)$ の様に括られています. この場合は, $(+ 1 (* 2 (+ 3 (/ 4 5))))$ と括弧を入れ子にして記述します.

この考え方は, $+$ や $*$ が二項演算子である事から小括弧を使って分けて行けば判り易くなります. 最初に $1 + 2 * 3 + 4/5$ は $1 + (2 * (3 + (4/5)))$ へと括弧を使って纏める事が出来ます. この式に対して括弧の部分を外側, 或いは内側から前置表現に置換えると, S 式 $(+ 1 (* 2 (+ 3 (/ 4 5))))$ が得られます. この前置表現を階層的な木構造として表現したものを以下に示しておきます.



このリストの階層構造は非常に重要です. Maxima の数式も内部ではこの様な前置表現のリストで表現され, この内部表現を操作する事によって式の簡易化や値の代入等の処理を行うからです.

猶, LISP では演算子だけではなく関数も前に置きます. 即ち, 関数 f の作用 $f(x_1, \dots, x_n)$ を LISP では $(f x_1 \dots x_n)$ と表記します.

3.2.1 LISP の代表的な数値関数

Common Lisp で利用可能な数値関数に次のものがあります。

LISP の標準的な数値関数

- ・ 四則演算: + - * /
- ・ 剰余: mod
- ・ 三角関数: cos sin tan acos asin atan
- ・ 指数及び対数関数: exp log

これらの関数の書式は全て先頭に関数があり、その後に数値が続きます。

ここで数値関数の例として、LISP の組込関数である sin 関数で試してみましょう。

```
[9]> (sin pi)
-5.01655761368433602461-20
[10]> (sin (/ pi 2))
1.010
```

この例では、 $\sin(\pi)$ と $\sin(\frac{\pi}{2})$ の値を計算しています。 $\frac{\pi}{2}$ の個所で関数表記が入れ子になっている事に注意して下さい。ここで (/ pi 2) の代りに pi/2 と入力すればエラーになります。何故なら、分数の分母と分子は両方が整数でなければならない、演算子としての / は前置表現にしなければならないからです。ここで、LISP の定数 pi は浮動小数なので pi/2 は LISP の分数データとしてはあり得ないものです。

この様に LISP では式は括弧だらけになります。その為、LISP は Lots of Irritating Superfluous Parentheses の略記であるという冗談さえもあります。

この括弧で括られたものを LISP でリストと呼びます。実は LISP の名前の由来はリスト処理 (LISt Processing) から来ています。リストにはリストを含んだり、後述の配列やハッシュ表を混在させる事が可能です。猶、原子とリストを合せたものを S 式と呼びます。

他に Common Lisp で使える数に複素数もあります。例えば、複素数 $a + ib$ を LISP では #c(a b) で表現します。以下に複素数の例を示しておきます。

```
[26]> (* #c(1 2) #c(2 -1))
#c(4 3)
[27]> (* #c(2 1) #c(2 -1))
5
[28]> (* 2 #c(2 1))
```



```
#c(4 2)
[29]> (* #c(0 1) #c(2 1))
#c(-1 2)
[30]> (- #c(1 0) 1)
0
[31]> (= #c(1 0) 1)
t
```

ここで`#c(a 0)`は実数`a`と同じです。上の例の最後の二つで、最初は1と差を取る事で、二度目は`=`演算子を用いて等しかどうかを確かめています。ここで`=`で調べた結果として`t`が返却されています。

LISPでは`t`は特別な意味を持ち、Boole代数の真を意味します。因に、偽は`n`ではなく`nil`と記述します。この`nil`は色々な使われ方をするもので、真理値の偽の他に、空リストやEOF(=end of file)を表現したりと様々な場面で、どちらかと言えば否定的な意味で用いられています。

LISPの文字データは文字列とは別のデータ型です。例えば、文字`a`は`#\a`の様に先頭に`#\`を付けて表現されますが、文字列は`"abcd"`の様に、文字の羅列を二重引用符で括ったものとなります。この様に文字列の表現自体はCやFORTRANと同じです。猶、文字型も`string`函数を用いれば文字列に変換出来ます。

文字列の同士の結合は`concatenate`函数を用います。以下に`concatenate`函数の実例を示します。

```
[23]> (concatenate 'string (string #\a))
"a"
[24]> #\a
#\a
[25]> (string #\a)
"a"
[26]> (code-char 65)
#\a
[27]> (char-code #\a)
65
[28]> (concatenate 'string (string #\a) (string #\b))
"ab"
[29]> (concatenate 'string "abc" "123")
"abc123"
```

この例では、最初に string 関数を用いて文字#a を文字列"a"に変換しています。次の code-char 関数 は整数値から文字に変換を行う関数で、char-code 関数はその逆操作になります。最後の例では concatenate 関数を用いた文字列の結合を行っています。

3.3 リスト

LISP は前述の様にリストと呼ばれる構造を持ったデータ処理を主な目的としています。リストの基本的な形は (1 2 34 'abcd) の様に空行で区切られた原子を括弧で括ったものですが、リストのリストといった複合リストもリストです。リストは C の配列と似てなくもありませんが、C の配列の様に、最初に宣言した型に縛られず、非常に柔軟な構造を持っています。

LISP では一般の関数も (+ 1 2 34) の様にリストの先頭に配置するのが大きな特徴です。因に、この式の意味は、第一成分の+が第二成分以降を引数とする関数で、具体的には $1 + 2 + 34$ を意味します。

LISP のプログラムは簡単に言ってしまうと、関数の集合になります。関数がリストとなっているので、LISP もリストの集合、正確に言えば、S 式で構成されたものになります。その為、汎関数が簡単に定義出来てしまいます。

例えば、1 から指定された自然数 n までのリストを生成し、そのリストに関数 f を作用させる関数 example1 を次の様に定義します。

関数 example1

```
(defun example1 (f n)
  (let((a nil))
    (dotimes(i n)
      (setq a (append a (list (+ i 1))))))
    (apply f a)))
```

関数定義では defun 関数が使えます。詳細は後の節で解説します。

この関数による処理例を以下に示します。

```
[36]> (example1 '+ 10)
```

```
55
```

```
[37]> (example1 '* 10)
```

```
3628800
```

```
[38]> (example1 '/ 10)
```

```
1/3628800
```

```
[39]> (example1 '- 10)
-53
[40]> (example1 'list 10)
(1 2 3 4 5 6 7 8 9 10)
```

この様に C と比較すると随分と雰囲気が違う事が判るかと思います. 因に, C や FORTRAN は手続型言語と呼ばれる範疇に入ります. 実際, プログラムは計算機に実行させる手続を順番に記述する形になっています. これに対して LISP は関数型と呼ばれるプログラム言語となります. 関数型のプログラムでは基本的に関数を記述して, 関数を組合せてプログラムを組立てる形になります.

以降, 非常に基本的な LISP の命令を幾つか紹介しましょう. ここで紹介する命令は他のリストを扱える数式処理でも実装されている事が多いので覚えておいても損はないと思います.

- car

リストの先頭の要素を取り出します.

```
[1]> (car '(1 2 3 4 54))
1
```

- cdr

リストの先頭の要素を除いたリストを返します. 猶, 空のリストの値は nil になります.

```
[2]> (cdr '(1 2 3 4 5))
(2 3 4 5)
[3]> (cdr (cdr (cdr '(1 2 3 ))))
nil
```

- append

複数のリストを結合して一つのリストにします. 新しく生成されたリストは, 基となる各リストの成分から構築されます. 要するに, 各リストの一番外側の括弧を外して, リストを繋いで行く感じになります.

```
[3]> (append '(1 2 3 4) '(2 3 4))
(1 2 3 4 2 3 4)
```

```
[4]> (append '(1 2 3 4) '(5 6 7)) '('a 'b 'c 'd))
(1 2 3 4 5 6 7 'a 'b 'c 'd)
[5]> (append '(1 2 3 4) '(5 6 7)) '(1 2 3 ('a 'b 'c 'd)))
(1 2 3 4 5 6 7 1 2 3 ('a 'b 'c 'd))
```

- cons

二つのリストを結合して新しいリストを生成します。例えば、二つのリスト a と b から cons で生成したリストに対し、car を作用させると a, cdr を作用させると b が戻る様に結合されます。

```
[6]> (cons '(1 2 3 4) '(5 6 7))
((1 2 3 4) 5 6 7)
[7]> (car (cons '(1 2 3 4) '(5 6 7)))
(1 2 3 4)
```

他に, caar, cadr や caddr の様に, car と cdr の組み合わせた関数があります。最初の caar は二度引数に car を作用させる関数と同値です。次の cadr は最初に cdr を作用させ、その結果に car を作用させます。最後の caddr は何でしょうか？答は cdr を二度作用させる事と同値な関数になります。

3.4 t と nil

LISP での真偽値は真であれば t, 真でなければ nil を用います。この nil は LISP では何かと良く使う便利な値です。例えば、空のリストも nil と表現します。

```
[2]> (> 1 3)
nil
[3]> (> 3 2)
```

LISP では評価を行うと真か偽を返す S 式の事を述語と呼びます。述語は論理和 or や論理積 and で結合する事が可能です。

論理和 or は $(\text{or } \langle \text{述語}_1 \rangle \langle \text{述語}_2 \rangle \dots)$ の様に用い、述語の内の何れか一つが t であれば t を返し、それ以外は nil を返します。

これに対し、論理積 and は $(\text{and } \langle \text{述語}_1 \rangle \langle \text{述語}_2 \rangle \dots)$ の様に用い、全ての述語が t の場合のみ t を返し、それ以外は nil を返すものです。

LISP には述語を用いた条件分岐として, if 式と cond 式があります。以下の例では不等号を用いて大小関係を評価した結果を示しています。

```
[10]> (if(< 3 2)(print 'neko)(print 'mike))
```

```
mike
```

```
mike
```

```
[11]> (if(> 3 2)(print 'neko)(print 'mike))
```

```
neko
```

```
neko
```

```
[12]> (setq a 128)
```

```
128
```

```
[13]> (cond ((> a 10)(print 'mike))
```

```
((a <=10)(print 'tama)))
```

```
mike
```

```
mike
```

3.5 配列

LISP は配列が扱えます。配列の生成は `make-array` 函数を用います。生成した配列に一定の初期値を与えたければ、オプションキーワード `:initial-element` を用いて配列の初期値の設定が行えます。これはリストと同様です。

配列で成分の参照を行う場合、`aref` 函数を用います。 `(aref 〈配列〉 n)` で与えられた〈配列〉の `n` 番目の成分を参照します。配列の添字は C と同様に 0 から開始します。

```
[6]> (setq a1 (make-array 5 :initial-element 1))
```

```
 #(1 1 1 1 1)
```

```
[7]> (aref a1 0)
```

```
1
```

```
[8]> (setf (aref a1 2) 2)
```

```
2
```

```
[9]> a1
```

```
 #(1 1 2 1 1)
```

`setq` 函数は原子に値を割当てて函数です。これに対して、`setf` 函数は `setq` 函数に似ていますが、`setf` 函数の場合は函数の返却値を設定する事が可能です。この例で

は、`(setf (aref a1 2) 2)` で配列 `a1` の第3成分に2を設定させていますが、これは配列 `a1` の第3成分を参照すると2が返るといった方式で設定しています。

3.6 ハッシュ表

LISP はリストと配列の他にハッシュ表が扱えます。ハッシュ表の生成は `make-hash-table` 関数を用います。単純に生成するだけであれば、`(make-hash-table a)` でハッシュ表 `a` が生成されます。このハッシュ表は配列とは少し違った使い方が出来ます。配列の場合、値と配列内部での位置が対応しますが、ハッシュ表の場合、キーと値が対応します。ハッシュ表の値の参照は `gethash` 関数を用いてキーを指定して参照します。

```
[3]> (setq a (make-hash-table))
#s(hash-table eql)
[4]> (setf (gethash :one a) 1)
1
[5]> (setf (gethash :two a) '(1 2))
(1 2)
[6]> (setf (gethash :three a) "1 2 3")
"1 2 3"
[7]> a
#s(hash-table eql (:three . "1 2 3") (:two 1 2) (:one . 1))
[8]> (gethash :three a)
"1 2 3" ;
t
```

この例では、`a` にハッシュ表を割当てています。ハッシュ表に値を生成する為に、`setf` 関数を用いて、`gethash` 関数でキーが指定された場合の値として、ここでは値を割当てています。ここで、キーは `:one`, `:two`, `:three` を用いています。

3.7 割当と評価

`setf` 関数と `setq` 関数は変数に値 (原子, 或いはリスト) を割当てる関数です。`setf` 関数と `setq` 関数は通常の変数に対しては同じ作用を行います。が、`setf` 関数は変数以外のものに対する割当も行えます。この性質を上手く使う事で、`setf` 関数だけで済ます事も出来ます。

eval 関数に変数に割当てられた値を評価します. let 関数は指定した値で変数を一時的に束縛する関数です.

```
[26]> (setf a 1)
1
[27]> (* 2 a (+ a a))
4
[28]> (setq a '1)
1
[29]> (setq b '128)
128
[30]> (setq c '(/ a b))
(/ a b)
[31]> (eval c)
1/128
[32]> (let ((x '2)) (* x 2))
4
[33]> x
```

```
*** - eval: variable x has no value
```

eval は先頭に評価をさせない為の前置詞'(逆クオート)が付いた S 式を評価する際に使います. この方式は Maxima, Maple や *Mathematica* でも `'diff(a,x)` といった名詞型の表記を評価する際に, 同名の eval 関数を用います.

let は局所変数を作り出す関数です. 上の例で示す様に一時的に x に 2 を束縛させ, let 文が終了すれば, x には値が最初の状態の無設定に戻ります. 値が設定されていれば, その let 関数を実行する前の値に戻されます. 以下に値が設定されていた時の例を示しておきましょう.

```
[35]> (setf y '3)
3
[36]> (let ((x '2)(y '10)) (* x y))
20
[37]> y
3
```

この例で示す様に,let の中では一時的に 10 が束縛されていますが,let が終了すると元の値の 3 に戻っている事に注意してください.

猶,let の場合, 局所変数の初期値に別の局所変数の値を用いる事が出来ません. 別の局所変数の値を利用したければ,let* 関数 を用います. この let* 関数の使い方は let 関数 とほぼ同じです.

```
[12]> (let ((x 2)(y (sin x)))(+ x y))
```

```
*** - eval: variable x has no value
```

```
the following restarts are available:
```

```
store-value      :r1      you may input a new value for x.
```

```
use-value        :r2      you may input a value to be used instead of x.
```

```
break 1 [13]> :q
```

```
[14]> (let* ((x 2)(y (sin x)))(+ x y))
```

```
2.9092975
```

3.8 構造体

LISP では C の様な構造体を持っています.C と違うのは, 構造体を定義すると, その構造体を扱う為の関数が自動的に生成される事です.

構造体の定義では defstruct 関数を用います. 例として,force という構造体を定義しましょう. この force には fx,fy,fz を持たせます.

```
[1]> (defstruct force fx fy fz)
```

```
FORCE
```

```
[2]> (setf F_1 (make-force :fx 0 :fy 0 :fz -9.80665))
```

```
#S(FORCE :FX 0 :FY 0 :FZ -9.80665)
```

```
[3]> (force-fz F_1)
```

```
-9.80665
```

```
[4]> (force-p F_1)
```

```
T
```

この例の様に,force という構造体を定義すると, この構造体を生成する為の関数 make-force と各成分を取出す関数 force-fx,force-fy,force-fz, そして定義した構造体であるか

どうかを判定する述語関数 `force-p` が自動的に生成されており、実際に構造体の定義と値の取り出しと検証が行えます。

3.9 写像関数

LISP には写像関数と呼ばれ、関数をリストの各成分に作用させる命令があります。このような関数はリストが扱える様に設計された言語にも実装されている事が多いので覚えていても損はありません (例えば, Maple の `map` 関数)。

`mapcar`

`mapcar` 関数は、関数をリストの各成分に作用させる際に用いる関数です。

```
[1]> (mapcar 'sin '(1 2 3 4 5))  
(0.84147096 0.9092974 0.14112 -0.7568025 -0.9589243)
```

この例では `sin` 関数をリスト (1 2 3 4 5) の各成分に作用させたリストを返しています。

`apply`

`apply` 関数も関数をリストに作用させる関数です。簡単に言えば、リストの先頭に関数を挿入した S 式を評価するものと言えるでしょう。

```
[154]> (defun pab( x y ) (+ x y))  
pab  
[155]> (apply 'pab '(1 2))  
3
```

3.10 lambda 式

lambda 式は Church の λ -計算 を LISP に実装したものです。

LISP の大きな特徴の一つが lambda 式の存在です。lambda 式の考え方は、関数の変数を選出し、その変数に対する関数として関数を表記する方法です。例えば $f(x) = x^2$ が与えられた時、この表記方法では、関数 f を表現しているだけでなく、 x での値を表現しているとも言えます。そこで、 $\lambda x x^2$ と表記して関数である事を明記する為に導入された表記方法です。

これを LISP の lambda 式に書き写すと `(lambda (x) (* x x))` となります. この lambda 式だけでは, 計算手順を記しただけのものですが, LISP では, この lambda 式に `apply`, `mapcar` 等の函数を用いて, 非常に柔軟に適用して行きます.

この lambda 函数は Maxima のソースファイルを眺めるとそこらじゅうに出てきます. 更に, Maxima の函数にも lambda 函数が存在し, 色々と使えて便利なので, この使い方を覚えておくのも良いでしょう.

3.11 函数の定義

函数の定義は `defun` 函数を用います. 例えば, 与えられた数値を 2 倍にする函数は `(defun x2 (x) (* 2 x))` の様に定義出来ます. ここで, 局所変数を利用したければ, `let` 函数や `let*` 函数を用います.

`let` 函数では, 局所変数に初期値の設定を行う事は可能ですが, 他の局所変数の初期値を用いて新しく別の局所変数の設定を行う事は出来ません. その様な事を行いたければ, `let*` 函数を用います.

3.12 制御文

条件分岐には `if` 式や `cond` 式が使えます.

if 式

```
(if < 述語 > )(<S 式1>) (<S 式2>)
```

`if` 式は `< 述語 >` が `true` の場合, `<S 式1>` を実行し, 述語が `false` の場合に, `<S 式2>` を実行します.

猶, `<S 式1>` には論理積 `and` や論理和 `or` で結合した S 式が使えます. 更に, `<S 式2>` も論理積 `and` で結合した S 式が使えます.

```
(defun my_matrixp (x)
  (if (and (typep x 'hash-table)
           (numberp (gethash :row x))
           (numberp (gethash :col x)))
      t
      nil))
```

cond 式

```
(cond  (((述語1))  (<S 式1>))
      ...
      (((述語n))  (<S 式n>))
      (t          (<S 式n+1>)))
```

cond 式も初めて $\langle \text{述語}_i \rangle$ が t の場合, $\langle \text{S 式}_i \rangle$ を実行して cond 式を抜けます. もし, 全ての述語が t にならなかった場合, 最後の $\langle \text{S 式}_{n+1} \rangle$ を実行して終了します.

```
(defun set-matrix-element (a x e)
  (if(my_matrixp x)
      (cond(<(> (car a)(gethash :row x))nil)
          (<(> (cadr a)(gethash :col x))nil)
          (t(setf (gethash a x) e))))))
```

3.13 属性リスト

LISP には属性リスト (P リスト) と呼ばれる対象があります. 属性リストは LISP の任意の記号に付与されるもので, 属性 (キー) と属性値の対を並べた平リストです:

属性リスト

```
(<属性1> <属性値1> ... <属性n> <属性値n>)
```

属性の設定と取出に関連する関数を次に纏めておきます:

属性の設定と取出

```
(setf (get <記号> <属性>)) <属性値>)
(push <属性値> (get <記号> <属性>))
(get <記号> <属性>) <属性値>)
(symbol-plist <記号>)
```

最初の setf と push 関数 で記号に対する属性リストの設定が行えます. 猶, pop 関数, inof 関数や deof 関数も属性リストの設定で利用可能です. 又, 属性値の取出は get 関数を用います. そして, 記号に設定した属性リストの表示は symbol-plist 関数を用います.

```
[8]> (push 'みけ (get 'my_cat '名前))
(みけ)
[9]> (push 'たま (get 'my_cat '名前))
(たま みけ)
[10]> (push 'とら (get 'my_cat '名前))
(とら たま みけ)
[11]> (push 'カトリース (get 'my_cat '名前))
(カトリース とら たま みけ)
[12]> (symbol-plist 'my_cat)
(名前 (カトリース とら たま みけ))
[13]> (get 'my_cat '名前)
(カトリース とら たま みけ)
```

属性リストの基本的な考え方は Russell の「還元可能性公理」や集合論の「内包の公理」を LISP に適合させたものと言えるでしょう。ここで [還元可能性公理] は、「任意の階の述語に対して同値な 1 階の述語が存在する」というものです。LISP の対象は原子や S 式になりますが、S 式にはリスト構造から由来する階層が入ります。ここで最下層の記号を 0 階とすると、S 式を指示する記号に対して属性で表現される性質を付加する事は、 n 階の対象を同値な 0 階の記号で置換し、その記号に対して性質を述べる事に対応すると言えます。

属性を用いる事で、属性に対応する集合 (クラス、或いは外延) が一つ定まります。属性を用いる利点は、集合が外延的定義ではなく内包的定義で行える事、つまり、ある対象が何等かの集合の成分である事を述べる際に、集合を構成する成分を列記したリストを必要しない事です。例えば、「 $\times\times$ 高校 3 年 A 組のクラス」とすれば人名を列記したリストは必要ありませんね。そして、実数の集合の様な、より抽象的な概念に対して内包的定義は威力を発揮します。

3.14 入出力

この節では簡単に LISP の入出力について述べます。

先ず、LISP のプログラムファイルの読込は load 関数を用います。

— load 関数 —

(load < ファイル名 >) 指定したファイルの読込を行う。

例えば,tama.lisp という名前のファイルに記述した関数等の読込を行う場合, `(load "tama.lisp")` とします.

Common Lisp はストリームを用いてファイルの入出力が行えます. ストリームを開く場合には,open 関数を用い, 閉じる場合には close 関数を用います.

— open 関数 —

(open ファイル名 :direction :input) 指定したファイルを読込用に開く.
(open ファイル名 :direction :output) 指定したファイルを書込用に開く.

— close 関数 —

(close ファイル名) 指定したファイルを閉じる

では実際に簡単なファイル操作の実例を示しましょう. ファイル test1 を予め準備し, それを LISP で開いて内容を読み取り,test2 ファイルに書出してみましょう. ここで,test1 の内容を以下に示しておきます.

— ファイル test1 の内容 —

1 2 3 4
"TEST"

以下に LISP での処理を示します.

```
[1]> (setf test1 (open "test1" :direction :input))
#<INPUT BUFFERED FILE-STREAM CHARACTER #P"test1" @1>
[2]> (setf x1 (read test1) x2 (read test1) x3 (read test1) x4 (read test1))
4
[3]> (setf str1 (read test1) )
"TEST"
[4]> (close test1)
T
[5]> (setf test2 (open "test2" :direction :output))
#<OUTPUT BUFFERED FILE-STREAM CHARACTER #P"test2">
[6]> (print str1 test2 )
"TEST"
[7]> (print x1 test2 )
1
```

```
[8]> (print x2 test2 )
2
[9]> (print x3 test2 )
3
[10]> (print x4 test2 )
4
[11]> (close test2)
T
```

この例では最初に open 関数で読込用にファイル test1 を開きます. この際に, test1 をストリーム名にします. 以降, read 関数を用いてファイルの読込を行います. それから close 関数でストリーム test1 を閉じる事でファイル test1 を閉じます. それから今度はファイル test2 を書込用に開きます. 今度は print 関数で各原子に割当てられた値をファイル test2 に書込みます. そして, 最後は close 関数でストリーム test2 を閉じています. 猶, print 関数は改行を行う為, ファイル test2 の内容は以下の様になります.

—— ファイル test2 の内容 ——

```
"TEST"
1
2
3
4
```

このファイルでの入出力で出て来た read 関数は, ストリームを指定した場合, 指定したストリームからの入力を読込み, ストリームを指定しなければ, キーボード (正確には *standard-input* ストリーム) からの入力を読込む関数です.

原子を評価して書式無しで出力する関数に prin1 関数, princ 関数と print 関数があります. これらの関数は引数は一つですが, オプションとして出力ストリームが指定出来ます.

書式付きの出力関数には, format 関数があります. format 関数は C の書式付きの fprintf 関数と似た機能を持っています. 但し, 書式は fprintf 関数のものとはやや異なっています.

—— format 関数 ——

```
(format 出力ストリーム 書式文字列 S 式)
(format t 書式文字列 S 式)
```

ここで、書式文字列に書式指示子を幾つ書いても構いませんが、書式指示子の総数と S 式の総数は合せておく必要があります。又、出力ストリームの代りに t を置くと、通常の表示となります。

ここで、指定可能な書式指示子の主なものを以下に挙げておきます。

—— 主な書式指示子 ——

a	エスケープ無し出力
s	エスケープ付き出力
%	改行
w,dF	固定少数点表示
w,dE	固定少数点表示
w,dG	一般小数点表示

```
[75]> (setq a (sin (/ pi 4)))
```

```
0.70710678118654752444L0
```

```
[76]> (format t "a=~a" a)
```

```
a=0.70710678118654752444L0
```

```
NIL
```

```
[77]> (format t "a=~s" a)
```

```
a=0.70710678118654752444L0
```

```
NIL
```

```
[78]> (format t "a=~5,3f" a)
```

```
a=0.707
```

```
NIL
```

```
[79]> (format t "a=~5,3e" a)
```

```
a=7.071L-1
```

```
NIL
```

```
[80]> (format t "a=~5,3g" a)
```

```
a=.707
```

```
NIL
```

以上、大雑把ですが LISP の概要を終えます。猶、§11 や §12 に Maxima の動作や、Maxima の改造に関連した話を載せているので、参考にされると良いでしょう。

—— コラム:Common Lisp 色々 ——

Common Lisp の系統に属するものには, 無課金で利用可能な GCL,CLISP や CMUCL , 商用の Allegro Common Lisp(ACL) 等があります. その中で,Maxima は GCL,CLISP と CMUCL で正式にサポートされています. sourceforge から入手可能な Maxima は GCL を用いていますが,LISP の選択自体は個々人の環境や目的に大きく依存します. 現時点では, 様々な環境で使いたければ CLISP, 速度重視であれば CMUCL, 過去の遺産を使うのであれば GCL で選びましょう. 又, 最近では CMUCL から分枝した SBCL や CMUCL の Mac 向けから派生した openMCL^a もあります.SBCL は 64bit 環境にも対応しているので, 試してみるのも良いでしょう.

^a当然ですが,MACSYMA が記述されていた MACLISP とは直接関係しません.

第4章 数学の色々な事

原初に言葉 (λογος) があった.
言葉は神と共にあった.
言葉は神であった.
これは原初に神と共にあった.
万物はこれによって成り, これに因らぬ物は何一つとして無かった.

ヨハネの福音書, ロゴス讃歌より

この章では Maxima を使う上で知っておくと便利な数学の言葉と考え方について簡単に解説します. Maxima では代数学の考え方が色々と取り入れられています. この考え方は今では計算機代数と呼ばれる分野に相当します.

その為, 群や環, そしてイデアル, そして, 多項式環に対しては, 順序という考え方を紹介します. 又, Maxima の様な数式処理全般に直接的に大きな影響がある数理論理学関連の話も纏めています. 猶, 非常に簡易な説明の為, 興味を持たれた場合, ここで挙げる参考文献を参照されると良いでしょう.

猶, この本は教科書のような真面目な本ではなく, 寧ろ, 鬼火の様にふらふらと飛び回っては, 面白そうな所を摘み食いするとても不健全, 且つ不真面目な本の為, あちらこちらに脱線するかと思いますが, その際は御容赦を.

4.1 集合について

先ず手始めに集合の話から始めましょう。集合というものが出てくる理由は、現代の数学が集合の上に成立している為に幾つかの用語や記号を知っておくと何かと便利だからです。

さて、集合とは具体的なもの(対象)のあつまりの事です。例えば、「偶数の集合」や「1よりも大きな実数の集合」の様に明確にその構成要素が指定出来るもの¹や、記号“{}”を用いて{1,2,3,4,5}の様にその構成要素を列記したものとして表現されます。

さて、集合は記号“{}”を用いて表記します。例えば、「偶数の集合」は“ $\{x|x \text{ は偶数}\}$ ”と記述します。この様に集合を構成する成分の性質(ここでの例では x は偶数という命題)を述べる集合の定義方法を内包的定義と呼びます。もう一つの定義方法は、具体的に $\{a_1, a_2, \dots, a_n\}$ と具体的に集合を構成する成分(ここでの例では $a_1, a_2, \dots, a_n$)を列記する方法で、この方法による集合の定義方法を外延的定義と呼びます。そして、集合を構成する対象の事を「集合の元」と呼びます。もしも、対象 a_i が集合 A の元の場合、 $a_i \in A$ と表記します。猶、集合の元を列記する場合、その順序が異なっても問題はありません。例えば、 $\{1,2\}$ と $\{2,1\}$ は同じ集合になります。

集合 A と B が与えられ、集合 A の元が全て集合 B の元である場合、集合 A の事を部分集合と呼び、この事を $A \subseteq B$ と記述します。更に、集合 A が集合 B の部分集合であり、集合 A に含まれない集合 B の元 b が存在する時、集合 A の事を集合 B の真部分集合と呼び、 $A \subset B$ と表記します。更に、集合 A と集合 B の全ての元が一致する場合は、 $A = B$ と表記し、集合 A と B は等しいと呼びます。

元を持たない集合“{}”という集合も考えます、この集合“{}”の事を空集合と呼び、記号 $\emptyset$ で表記します²。因に、空集合 $\emptyset$ は任意の集合の部分集合になります。

集合 S が与えられた時、この集合 S の全ての部分集合を元とする集合を集合 S の冪集合と呼び、 $\mathfrak{P}(S)$ や 2^S で記述します。例えば、集合 $S = \{1, 2, 3\}$ の冪集合は、 $\mathfrak{P}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$ となり、冪集合の元の個数(基数と呼びます)は $2^3 = 8$ になります。集合 S の基数を $\text{card}(S)$ と表記すると、一般的に $\text{card}(\mathfrak{P}(S)) = 2^{\text{card}(S)}$ が成立し、この事が冪集合を 2^S とも表記する理由になっています。

集合 S の冪集合 $\mathfrak{P}(S)$ に対しては“ $\cup$ ”と“ $\cap$ ”, “ $\setminus$ ”や“ $-$ ”といった演算があります。先ず、 $A, B \in \mathfrak{P}(S)$ に対し、 A と B の元で構成された集合の事を集合 A と集合 B の和集合と呼び $A \cup B$ と記述します。同様に、集合 A と集合 B の共通な成分で構成された集合を共通集合と呼び、 $A \cap B$ と記述します。そして、集合 B から集合 A の元を取

¹この集合の説明には大きな問題があるのですが、暫く類被りしたままにしておきます。

²André Weil による [5]

り除いた集合を, $A - B$ と記述し, 差集合と呼びます. 猶, $S - A$ を A の補集合と呼び A^c と記述します. 猶, 演算 “ $-$ ” と “ ^{c} ” に関しては, 定義から $A - B = A \cap B^c$ が成立します.

4.2 同値関係について

数式処理で扱う対象は数式です. 表面上, 数式処理は式を人間が扱様に処理する事から人工知能の一分野の様に見えますが, 実際は式の代入, 置換等の代数的処理を中心に行うものです. その為, 有限個の語 (word) と呼ばれる文字列集合に対し, 有限回の操作を行う事が中心で, その操作に様々な規則を入れて処理を行うものとも言えます. 数式処理システムによっては, 最初に環と呼ばれる数式を考える世界を指定し, その中で数式や写像を処理して行くものもあります. これは Java の様なオブジェクト指向の言語に似たもので, 環を定義する事で様々な操作が可能になります. その一方で, Maxima, Mathematica や Maple 等の様な汎用数式処理の多くは, 数や多項式と関数の集合を考え, その集合内部で和, 差, 積, 商の四則演算に冪等を加えた基本的な演算に, 利用者定義の演算で結合されたものを数式として, それらの処理を行います.

さて, 計算機にとって入力された式は人間が何らかの指定をしない限りは単なる文字の羅列に過ぎません. では, 与えられた二つの数式が等しいかそうでないかを計算機に判断させる為には何が必要でしょうか? 又, より一般的に, 二つの与えられた式が等しいとはどのような事でしょうか?

ここで Euclid 幾何学を参考に考えてみましょう. Euclid 幾何学では, 二つの図形が与えられた時に, これらが等しい図形 (合同) であるとは, 二つの図形を重ね合わせる事が出来るかどうかに着目します. この重ね合わせる操作は, 平行移動と回転の有限回の組み合わせによる操作で, 一次変換と呼ばれる操作に属するもので一方の図形を片方の図形に変換出来るかどうかと言い換えられます.

これと同様に, 与えられた式が等しいかどうかは双方に或る有限回の操作を加えて互いに移り合う事が出来る場合とすればどうでしょう. 例えば, $x + y + 1$ と $y + x + 1$ が与えられた場合, 両者を繋ぐ演算が和 (+) なので, x と y の位置を入れ換えが可能であり, 一度だけ入れ換え操作をすれば互いに移り合えますね.

それだけではなく, 式に含まれる関数固有の規則が入る場合もあります. 例えば, 三角関数だけでも $\cos^2(x) + \sin^2(x) = 1$, $\cos(2\theta) = 2\cos^2(\theta) - 1$, $\sin(2\theta) = 2\sin(\theta)\cos(\theta)$ や $e^{a+i\theta} = e^a(\cos(\theta) + i \cdot \sin(\theta))$ といった公式を適用する事が可能です. この様に式の変換で考えると, $x + y + 1$ と $y + x + 1$ が等しいかどうかを x や y の値を色々と変更して計算する方法よりも適切である事が理解されるでしょう.

そこで手始めに身近な数である分数を使って解説してみましょう。分数は小学生にとって鬼門の一つですが考え直して見ると面白い数です。ここで分数は分母が零と異なる自然数 $\mathbb{N} - 0$, 分子が整数 $\mathbb{Z}$ の二つの数で構成された数です。さて, $\frac{1}{2}$ と $\frac{2}{4}$ は同じものでしょうか? $\frac{2}{4}$ を約分すれば, $\frac{1}{2}$ が得られるので, これらは同じ数です。ここで注意して頂きたい事は, $\frac{1}{2}$ と $\frac{2}{4}$ は各々 $(1, 2)$ と $(2, 4)$ の自然数の対で考えると全くの別物だという事実です。この事は分数を計算機のプログラムで二成分の配列を使って表現する場合, 二つの分数 a, b が等しいかどうかを判断させる時点で安易に $a[1]=b[1]$ かつ $a[2]=b[2]$ という規則だけで処理を行ってはならないという事です。計算機は石頭なので, 利用者が与えたとおりの処理しかしません。分数の約分を忘れていれば, 同じ分数でも違うと言い張りますね。その為, 約分という規則を計算機に教えてやらなければなりません。

この約分という規則を別の方面から眺めてみましょう。先ず, 分数は数学では有理数 $\mathbb{Q}$ と呼ばれる数です。有理数 $\frac{a}{b}$ を (a, b) の様に整数と零と異なる自然数の対で表現してみましょう。この整数と自然数の対 (a, b) と (c, d) が等しいのは, 零と異なる整数 q が存在して $a = cq, b = dq$ 或いは $c = aq, d = bq$ となる場合ですが, これは $ad - bc = 0$ が成立する事と同値です。そこで, 二つの整数の対 $A = (a, b)$ と $B = (c, d)$ が $ad - bc = 0$ を満す事を $A \sim B$ で記述しましょう。猶, ここでの“ $\sim$ ”は数学では(二項間の)関係と呼ばれるものになります。そして, この有理数 A と有理数 B の関係“ $\sim$ ”は次に示す性質を満す関係となり, この様な性質を持つ関係の事を数学では同値関係と呼びます。

—— 同値関係 ——

1. 反射律: $A \sim A$
2. 対称律: $A \sim B$ ならば, $B \sim A$
3. 推移律: $A \sim B$, かつ, $B \sim C$ ならば, $A \sim C$

そして $(1, 2)$, $(2, 4)$ の様に $(1, 2)$ と同値になるものの集合の事を同値類と呼びます。更に, 同値類から取出した一つの元の事を同値類の代表と呼びます。この見方から, 有理数は整数と自然数の対で分母となる第二成分が最小となるものを同値類の代表としています。即ち, 互いに素な整数 a, b に対し, 集合 $\{\frac{aq}{bq} \mid q \text{ は } 0 \text{ と異なる整数}\}$ が同値類で $\frac{a}{b}$ が同値類の代表です。全体集合を X とする時, 同値類の集合を $X/\sim$ と表記します。例えば, 有理数 $\equiv$ 整数 $\times$ 自然数 $-\{0\}/\sim$ となります。

ここで同値類と呼ばれる何やら凄いものが出てきましたが, 実際は, この様に分数でお

馴染の考え方を、より一般的なものとして言い直したものです。小学校で分数の計算に苦勞された方も多いかもしれませんが、実際、そんなに単純なものではなく、高度な概念を含んでいるのです。寧ろ、約分の様な規則から、同値関係といった考え方に昇華させているのです。

ではもう少し変わったものを考えてみましょう。次は、係数が実数で、変数が x の多項式の集合を考えてみましょう。この集合を $\mathbb{R}[x]$ と記述します。後の節で詳しく説明しますが、これは数学では（一変数の）多項式環と呼ばれるものになります。

次に多項式 f, g に対して $f - g$ が多項式 $x^2 + 1$ で割切れる事を $f \sim g$ と表記しましょう。すると、この関係“ $\sim$ ”も同値関係になります。実際、反射律と対称律は自明ですね。推移律は、 $f \sim g, g \sim h$ の場合、 $f - h = (f - g) + (g - h) = q_1(x^2 + 1) + q_2(x^2 + 1)$ となる多項式 q_1 と q_2 が存在するので、 $f - h$ も $x^2 + 1$ で割切れますね。この様に関係“ $\sim$ ”は同値関係になる事が判ります。では、 $\mathbb{R}[x]/\sim$ は何になるのでしょうか？ 先ず 1 や x はそのままですが $x^2 + 1$ は 0 で置換えられます。では $n \geq 2$ に対し、 x^n はどうなるのでしょうか？ ここで $x^n = x^{n-2}(x^2 + 1) - x^{n-2} \sim -x^{n-2}$ と x の冪を2つ減らせます。更に、 $x^2 = (x^2 + 1) - 1 \sim -1$ なので、結局、冪 x^n の項は n が 1 の場合だけしか残りません。その為、 $\mathbb{R}[x]/\sim$ は $a + bx$ の形に限定され、変数 x は $x^2 = -1$ を満す数と言い換えられます。この型の数は何処かで見た事のある数ですね。つまり純虚数です。従って、 $\mathbb{R}[x]/\sim$ は $\mathbb{R}[i]$ 、則ち、複素数 $\mathbb{C}$ です！

ザミャーチンの SF 小説、「われら」の一節で、純虚数の存在が整数の調和を乱すものとして、主人公は純虚数を嫌悪していますが、事実、純虚数は相当人工的な匂のする数です。この数は、整数係数の方程式 $x^2 + 1 = 0$ に縛り付けられています。この i 様な数の仲間が代数的整数と呼ばれる数です。

数式処理システムは単純な式の展開やまとめといった処理に留まらず、式に対する何等かの同値関係を処理するシステムとも考えられます。その関係を Maxima では演算子の属性（性質）の指定、演算子や式の規則等で設定します。与えられた問題を効率的に解く事は、この関係を如何に上手く Maxima に設定出来るかどうかに大きく依存します。具体的に Maxima で行う処理については §5.4 や §5.7 を参照して下さい。又、この章の参考文献としては、丸山 [35]、寺坂 [22]、Cox [49] と Greuel-Pfister [50] を挙げておきます。但し、寺坂 [22] は中学生にも拾い読みが出来る楽しい本ですが、数学基礎論の章は流石に古いと思えるので、この辺の話題に関してはウリグト [7] や林 [18] の付録、総合的な文献集として Heijenoort の本 [51] を挙げておきます。

4.3 群について

群は性質の良い二項演算を持った集合の事です。ここで集合を A , 演算を $*$ と表記すると, 二項演算 $*$ は集合 A の元の対 (a_1, a_2) に対し, 集合 A の元 $a_1 * a_2$ を対応させる函数になります。ここで, 二項演算を誤解する恐れが無い場合には, 単に演算と呼びます。さて, 群は集合と演算の対 $(A, *)$ として表記されます。但し, 演算を省略しても問題のない場合は, 集合 A のみで群 A と単純に記述します。

さて, $(A, *)$ が群と呼ばれる為には演算 $*$ が良い性質を持つ必要があると述べました。先ず, 集合 A の任意の二つの元 a, b に対し, $a * b$ は必ず集合 A に含まれていなければなりません。この性質を持つ演算 $*$ の事を, 集合 A は演算 $*$ に関して閉じていると呼びます。

演算 $*$ が閉じていれば, 任意の集合 A の元 a, b, c に対して, $a * b, b * c, (a * b) * c$ や $a * (b * c)$ も集合 A に含まれます。では任意の3個の元 a, b, c に対して $(a * b) * c$ と $a * (b * c)$ は同じものになるのでしょうか? この様な保証は実はありません。この $(a * b) * c = a * (b * c)$ となる性質は結合律と呼ばれる性質で, $(A, *)$ が群になる為には, この結合律を満たさなければなりません。

この結合律を満たせば何が良いかと言えば, $a * b$ を計算して c との演算を計算する $(a * b) * c$ と, $b * c$ を計算した結果と a との演算を計算する $a * (b * c)$ で両者に違いがない事が保証されるからです。結合律が成立する場合, $(a * b) * c = a * (b * c)$ なので $a * b * c$ と括弧を外して表記する事も出来ます。この結合律の話は後の式の表現でまた出てきます。

次に, 単位元というものがあります。単位元 u は任意の集合 A の元に対して $a * u = u * a = a$ を満たす集合 A の元です, この単位元 u を 1 と表記する事もあります。この単位元は存在すれば一つだけです。何故なら, 単位元として u の他に v が存在したとしましょう。そこで, $u * v$ を考えると, u は単位元なので $u * v = v$ となります。その一方で, v も単位元なので $u * v = u$ となります。これら両方を併せれば, $u = v$ が得られます。

猶, 演算について閉じており, 演算が結合律を満たし, 単位元が存在する $(A, *)$ の事を半群, 或いは準群と呼びます。この半分群になっている半群が本当に群になる為に必要な条件が逆元と呼ばれる元の存在です。集合 A の元 b が, 集合 A の元 a の逆元と呼ぶのは, $a * b = b * a = u$ を満たす場合です。この様な元 b を a^{-1} と記述します。又, 逆元を持つ元の事を正則元と呼びます。 $(A, *)$ が群になる為には, 半群 $(A, *)$ の全ての元が演算 $*$ に対して逆元を持つ事, 即ち, 集合 A の元が全て正則元でなければなりません。では, 群となる為の条件を以下に纏めておきましょう。

群の条件

- 演算 $*$ に対して閉じている:
 $a, b \in A \rightarrow a * b \in A$
- 結合律が成立:
 $(a * b) * c = a * (b * c)$ となる
- 単位元 1 の存在:
 $a * 1 = 1 * a = a$ となる $1 \in A$ が存在する.
- 逆元の存在:
任意の $a \in A$ に対し, $a * b = b * a = 1$ を満す $b \in A$ が存在する.

猶, 演算 $*$ が常に $a * b = b * a$ を満す場合, 演算 $*$ を可換 (commutative) であると呼び, 群 $(A, *)$ の事を可換群と呼びます. 可換で無い場合は非可換 (noncommutative) と呼び, 非可換な演算を持つ群の事を非可換群と呼びます. 因に, Maxima では記号 $*$ を可換な積に用い, 記号 $.$ を非可換な積で用います. 猶, Maxima で和 $+$ は常に可換です.

4.3.1 群の例

 $(\mathbb{Z}, +)$: 整数全体, 演算が和

群 $(\mathbb{Z}, +)$ の単位元は 0 で, $a \in \mathbb{Z}$ の逆元は $-a$ になります. 和 $+$ は可換なので, この群は可換群になります.

猶, 整数 $\mathbb{Z}$ には積 $*$ もありますが, この場合, 単位元が 1 になります. 積 $*$ に関しては 1 以外逆元を持ちません. しかし, 結合律を満すので, $(\mathbb{Z}, *)$ は半群になります.

 $(\mathbb{Q} - \{0\}, *)$: 0 を除く有理数全体, 演算が積

有理数全体の集合 $\mathbb{Q}$ から 0 を抜いた集合 $\mathbb{Q} - \{0\}$ の全ての元は整数 a, b を使って a/b と記述出来ます. 演算 $*$ の単位元は 1 で, その逆元は b/a です.

猶, $(\mathbb{Q}, *)$ は 0 が逆元を持たない為, 積 $*$ に関して半群になりますが群にはなりません. 但し, $(\mathbb{Q}, +)$ は可換群になります. この様に同じ集合でも演算によって群になる集合もあります.

$(SL(m), *)$

行列式の値が 1 になる m 次の正方行列の集合 $SL(m)$ は行列の積 $*$ に対して群になります. 但し, 任意の $a, b \in SL(m)$ に対し, $a * b$ が $b * a$ となるとは限らないので, $(SL(m), *)$ は可換群になりません.

m 次の正方行列全体の集合 $M(m)$ の場合, 全ての元が逆元を持つとは限らない為, $(M(m), *)$ は群になりません. 但し, 単位元として m 次の単位行列を持ち, 結合律も成立するので, $(M(m), *)$ は半群になります.

猶, Maxima では行列の演算で, 行列の積を非可換積”.”で表記し, その冪は演算子”^^”で表記しています.

語

集合 A を a から z までのアルファベットの小文字を並べた全ての文字列 (語と呼びます) と空白 (ここでは 1 と表記します) を加えたものとします. この集合に対し, 演算”*”を単純に二つの語を繋ぐ操作とします.

例えば, $abc * def = abcdef$ の様に, 演算子”*”の右の語を演算子 $*$ の左の語の末尾に繋げます. 猶, 空白を語の左右に繋いでも, 元の語となるので, 空白がこの演算 $*$ の単位元になる事が理解出来るかと思います. 猶, $WWWW \dots$ の様に同じ語 W が n 回続く場合, W^n と表記します.

語 W_1, W_2, W_3 が与えられた場合, $(W_1 * W_2) * W_3$ と $W_1 * (W_2 * W_3)$ は同じ語 $W_1 W_2 W_3$ になります. 従って, 演算子”*”は結合律を満たします. この様に, 単位元が存在し, 結合律を満たす事から, $(A, *)$ には半群の構造が入ります.

次に, 語 W に対し, W^{-1} を語の左隣か右隣に語 W があればどちらか一方を除去する操作とします. 更に, 語 w が 2 個以上のアルファベットで構成された語であれば, その並びを逆にして各々文字の除去操作に置換します.

例えば, 語 W が $neko$ であれば, W^{-1} を $o^{-1}k^{-1}e^{-1}n^{-1}$ に変換します. こうする事で語 W^{-1} は語 W の逆元となり, 最後に, 空行の逆元を空行とすれば, 全ての A の元に逆元が定まる為, $(A, *)$ は群になります.

この群は $\langle a, \dots, z \rangle$ の様に記述されます. ここで” $\langle \rangle$ ”の中の $a, \dots, z$ の事を生成元と呼びます. 又, この様に記述される群を自由群と呼びます. この群は一般的に非可換群になります.

猶, Maxima ではこの様な群も非可換積”.”を用いて表現する事が可能です. 更に, 非可換積の冪乗は通常の冪”^^”ではなく”^^^”を用います.

$$\langle x, y | x y x^{-1} y^{-1} \rangle$$

x と y が生成元の自由群 $\langle x, y \rangle$ に対して、語の中に $x y x^{-1} y^{-1}$ が出現すると、単位元 1 で置換する規則を入れた群 G が $\langle x, y | x y x^{-1} y^{-1} \rangle$ です。この規則を現わす $x y x^{-1} y^{-1}$ の事を関係子 (relator) と呼びます。

例えば、語 $x x y x^{-1} y^{-1} x$ は中に関係子が存在しますね。関係子の個所を分かり易く括弧で括ると $x (x y x^{-1} y^{-1}) x$ となります、この括弧の個所を 1 で置換えると x^2 が得られます。

ここで注目して頂きたい事は、語の並びを検出して関係子で規定される規則を当て嵌める操作を上処理で行った事です。実際、計算機にとって、与えられた式は最初は文字の羅列でしかありません。この羅列に意味を持たせる処理を Maxima では行っており、更に、利用者がこのような操作を Maxima にさせる事も出来ます。この処理は Maxima では非常に重要な処理になります。この事を §5.7 でより詳しく述べます。

さて、群 $\langle x, y | x y x^{-1} y^{-1} \rangle$ はどのような群でしょうか？語 $y x$ は $1 y x = x y x^{-1} y^{-1} y x$ なので、式の右辺を計算すれば、 $x y = y x$ という関係が得られます。即ち、この群は二つの元 x と y で生成される可換群になります。

すると、こんな写像 f が見えませんか？

$$\begin{array}{ccc} f & : & \langle x, y \rangle \rightarrow \langle x, y | x y x^{-1} y^{-1} \rangle \\ & \Downarrow & \Downarrow \\ & & (x, y) \mapsto (x, y) \end{array}$$

この写像 f は自由群 $\langle x, y \rangle$ から $\langle x, y | x y x^{-1} y^{-1} \rangle$ への写像で、単純に自由群の x と y を群 G の x と y にそのまま対応させている写像です。この写像は、準同型写像と呼ばれるものになります。

次に、集合 $H = \{r | r = h x y x^{-1} y^{-1} h^{-1}, r = h y x y^{-1} x^{-1} h^{-1}, h \in \langle x, y \rangle\}$ を考えると、この集合 H は群 G の部分群になる事が判ります。この群は面白い性質があり、任意の $h \in G$ に対して $h H h^{-1} = H$ を満たします。この性質を満たす群 G の部分群の事を正規部分群と呼びます。

更に、準同型写像 f による像 $f(H)$ は群 G の単位元 1 になります。この部分群 H の事を写像 f の核 (kernel) と呼びます。

この部分群 H を用いて群 G を別の見方をしてみましょう。先ず、自由加群 $A = \langle x, y \rangle$ に次の同値関係 $\sim$ を入れます。

$$a \sim b \Leftrightarrow a b^{-1} \in H \quad \text{ここで } a, b \in \langle x, y \rangle$$

この関係 $\sim$ による同値類の集合 $A / \sim$ も群になります。これを群 A を群 H で割ったものとも呼び、 A/H と表記します。すると、群 G は A/H と一致します。

より一般的には, 群 $G = \langle x_1, \dots, x_m | r_1, \dots, r_n \rangle$ は自由群 $A = \langle x_1, \dots, x_m \rangle$ をその正規部分群 $H = \langle hr_1h^{-1}, hr_1^{-1}h^{-1}, \dots, hr_nh^{-1}, hr_n^{-1}h^{-1} | h \in G \rangle$ で割ったもの $G = A/H$ と同じ群になります.

4.4 環について

整数の集合 $\mathbb{Z}$ には二つの演算, 和”+”と積”*”があります. $(\mathbb{Z}, +)$ は可換群になり, $(\mathbb{Z}, *)$ は可換半群になりますが, $(\mathbb{Z}, *)$ には逆元が 1 以外に存在しない為, 群にはなりません. この整数の集合 $\mathbb{Z}$ の様に演算子を二つ持つ集合 A で, 以下の性質を持つ $(A, +, *)$ を環と呼びます.

環の条件

- 集合 A には二つの演算の積”*”と和”+”があり, これらの演算に対して閉じている.
- $(A, +)$ は可換群になる. この時, 0 が演算”+”の単位元となる
- $(A, *)$ は半群になる. 演算”*”の単位元は 1 である
- 演算”+”と演算”*”は以下の分配律を持つ.
 左分配律: $a * (b + c) = a * b + a * c$
 右分配律: $(a + b) * c = a * c + b * c$

猶, 積”*”が可換の場合を可換環と呼びます. この場合, 左右の分配律に区別はありません.

環 $(A, +, *)$ に対し, $a, b \in A$ で $a * b = 0$ となる 0 と異なる元 a, b が存在する場合, これらを零因子と呼びます. 零因子を持つ環の例としては, 高校で習った事と思いますが, 一般の行列環 $M(n), n \geq 2$ が挙げられます,

更に, 零因子が存在しない可換環を整域と呼びます. 整域の例としては整数環 $(\mathbb{Z}, +, *)$ が挙げられます.

4.4.1 イデアル

環にはイデアルと呼ばれる環 $(A, +, *)$ の部分集合 I があります.

イデアルの定義

- 集合 I は和に関して群になります.
- 環 $(A, +, *)$ の任意の元 a に対し, $a * I \in I$ を満す場合, I を左イデアルと呼びます.
- 環 A の任意の元 a に対し, $I * a \in I$ を満す場合, I を右イデアルと呼びます.
- I が左イデアル, 且つ, 右イデアルでもある場合, I を両側イデアル, 或いは単にイデアルと呼びます.

可換群の場合, 右左イデアルの区別はありません.(左右) イデアルは和“+”の単位元 0 を含みますが, イデアルに積“*”の単位元 1 を含む場合は環そのものと一致します. その為, イデアルが環 $(A, +, *)$ の真部分集合であれば, 積“*”の単位元 1 を含みません. イデアルには以下のものがあります.

- 単項イデアル
イデアル I が生成元 a だけで生成される場合, イデアル I の事を単項イデアと呼びます. 猶, 環 R の イデアルが全て単項イデアルとなる環の事を主イデアル整域, PID(Principal Ideal Domain) と呼びます.
- 素イデアル
 $ab \in I$ であれば, $a \in I$ か $b \in I$ の何れかが成立するイデアル I の事を素イデアルと呼びます. 丁度, 整数環 $\mathbb{Z}$ の素数の様なものです.
- 極大イデアル
環 R のイデアル I で, $R \supset J \supset I$ を満す環 R のイデアル J が存在しない場合, イデアル I を極大イデアルと呼びます. 一般的に極大イデアルは一つだけとは限りません. 極大イデアルが一つだけしか存在しない環の事を局所環と呼びます.

4.4.2 環の例

多項式環

有理数 $\mathbb{Q}$ を係数とする n 変数多項式 $\sum_{i=0}^n a_i x^i$ の集合を $\mathbb{Q}[x_1, \dots, x_n]$ と記述します. すると, $(\mathbb{Q}[x_1, \dots, x_n], +, *)$ は和”+”に関して群, 積”*”に対しては半群となり, 更に, これらの演算に対して分配則が成立します. そして, 積”*”が可換となるので, $(\mathbb{Q}[x_1, \dots, x_n], +, *)$ は可換環になります.

群環

一般的な群 G と可換環 R が与えられた時に, 環 R を係数とする G の元の形式的な有限個の和を考える事で, 群環 (多元環とも呼びます) RG が定義出来ます.

可換環を整数環 $(\mathbb{Z}, +, *)$, 群を $G = \langle x_1, x_2, \dots, x_n | r_1, \dots, r_m \rangle$ とする時, 群環 $\mathbb{Z}G$ の構成方法を具体的に解説しましょう.

まず, 群 G の元 g と環 $\mathbb{Z}$ の元 a に対して, 形式的な積 ag を定めます. この際, $ag = ga$ とします. 猶, $0 \in \mathbb{Z}$ に対しては, $0g = 0$ とします. 次に, 群 G の有限個の元 $g_1, \dots, g_m$ と環 $\mathbb{Z}$ の元 $a_1, \dots, a_m$ との積の形式的な和 $a_1 g_1 + \dots + a_m g_m$ の集合を $\mathbb{Z}G$ とします. この形式的な和に関しては, $f, g, h \in G$ とすると和の可換性 $f + g = g + f$, 和の結合律 $(f + g) + h = f + (g + h)$ を入れます. こうする事で, $\mathbb{Z}G$ は和 $+$ に関して可換群になります. 次に, $f(g + h) = fg + fh, (g + h)f = gf + hf$ と左右の分配律を入れると, 集合 $\mathbb{Z}G$ は環になります.

この時, 群環 $\mathbb{Z}G$ の積に可換性を入れてしまえば整数係数の n 変数多項式環 $\mathbb{Z}[x_1, \dots, x_n]$ の剰余環になります.

この群環は Maxima の動作原理を考える際に非常に重要なものとなります.

整数環 $\mathbb{Z}$ のイデアル

2 の倍数の数の集合 (2), 3 の倍数の数の集合 (3) と 6 の倍数の数の集合 (6) を考えましょう. これらは全て整数環 $\mathbb{Z}$ のイデアルとなります. 実際, 2 の倍数の元の和は 2 の倍数の数の集合に属し, 任意の $a \in \mathbb{Z}$ に対し, $b \in (2)$ との積 ab は当然 2 の倍数となります. 他の (3), (6) も同様です.

更に, $a, b \in \mathbb{Z}$ に対し, $ab \in (2)$ であれば, a か b の何れかが必ず, (2) に含まれます. これは (3) も同様です. この様に, 整数環 $\mathbb{Z}$ の素数 p で生成されるイデアル (p) は素イデアルになります.

その一方で, (6) は, 2 と 3 は (6) に含まれませんが, $2 * 3 = 6$ となるので, (6) は素イデアルではありません. 又, 6 は 2 と 3 の倍数の為, $(2) \supset (6)$, 且つ, $(3) \supset (6)$ で, 更に, $(2) \cap (3) = (6)$ を満します. 即ち, $2 * 3 = 6 \Leftrightarrow (2) \cap (3) = (6)$ と対応関係があります.

整数環 $\mathbb{Z}$ のイデアルは全て単項イデアルになります. 何故なら, I を $\mathbb{Z}$ のイデアルとし, $a \in I$ をイデアル I に含まれる最小の正整数としましょう, すると, I が a で生成される事が判ります. 実際, $b \in I$ で $b = a * q + r$ となる $r < a$ が存在した場合, $r = b - a * q$ から, $r \in I$ となり, a が I に含まれる最小の正整数である事に反します. その事から, $r = 0$, 即ち, $b = a * q$ となり, I は最小の正整数 a で生成される事が判ります. 以上から, 整数環 $\mathbb{Z}$ は主イデアル整域 (PID) となる事が判ります.

有理数環 $\mathbb{Q}$

$(\mathbb{Q}, +, *)$ は和 $+$ に対して可換群, 積 $*$ に対しては半群になり, 積 $*$ は可換であり, 和 $+$ と積 $*$ が分配律を満す事から可換環になります. 更に, $\mathbb{Q}$ の 0 を除く元 a/b は全て逆元 b/a を持ちます. その為, $\mathbb{Q}$ のイデアルは 0 から生成される (0) 以外はありません. この事から (0) は有理数環 $\mathbb{Q}$ の唯一の極大イデアルとなるので $\mathbb{Q}$ は局所環になります.

可換環 R の局所化

R を可換環とし, p を R の素イデアルとしましょう, ここで R から p の元を除去した $R - p$ は積 $*$ に関して閉じており, 和 $+$ の単位元 0 を含まない集合になります. これらの性質を持った集合を積閉集合と呼びます.

次に, $(R - p)^{-1}R = (a, b) | a \in R, b \in R - p / \sim$ を考えます. ここでの同値関係は $(a, b) \sim (c, d) \Leftrightarrow ad - bc = 0$ とします. この関係は有理数で見られましたね. そこで同値類の代表元を $\frac{a}{b}$ と記述します. この操作で生成された環 $R/(R - p)$ は極大イデアルとして, $p \cdot (R - p)^{-1}R$ のみを持つ局所環になります. そして, この操作を局所化と呼びます.

猶, 有理数環 $\mathbb{Q}$ は整数環 $\mathbb{Z}$ を局所化する事で得られた環です. 即ち, $\mathbb{Q} = (\mathbb{Z} - (0))^{-1}\mathbb{Z}$ です. この様に, 整数から有理数を構成する手法は可換環の局所化としてより一般的なものとなったのです.

集合の演算

$\mathfrak{P}(S)$ を集合 S の冪集合とし、演算を $\cup$ と $\cap$ とします。この時、 $(\mathfrak{P}(S), \cup, \cap)$ は、 $A, B \in \mathfrak{P}(S)$ に対し、 $A \cup B = B \cup A$ と $A \cap B = B \cap A$ が成立する為、 $\cup$ と $\cap$ は可換演算子となります。又、演算 $\cup, \cap$ 共に結合律も満たします。

空集合 $\emptyset$ に対しては、 $A \cup \emptyset = A$ となり、 $\emptyset$ は演算 $\cup$ に対する単位元になります。

又、演算子 $\cup$ と演算子 $\cap$ に関しては、以下の分配律が成立します。

$$\bullet A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

補集合を取る演算子“ ^{c} ”に対しては De Morgan の法則と呼ばれる性質があります。

$$\bullet (A \cup B)^c = A^c \cap B^c$$

$$\bullet (A \cap B)^c = A^c \cup B^c$$

但し、演算 $\cup$ の集合 A の逆元が存在しない為、 $(\mathfrak{P}(S), \cup, \cap)$ は可換環にはなりません。

4.5 体について

体は可換体で、更に、その零元 0 を除いた任意の元が積“ $*$ ”に関して逆元を持つものの事です。例えば、有理数環 $(\mathbb{Q}, +, *)$ は可換環であり、その零元 0 を除く全ての元 a/b は逆元 b/a を持つ事から体になります。猶、非可換環で零元 0 を除いた任意の元が積“ $*$ ”に対して逆元を持つ場合は斜体と呼びます。

剰余環 R/I

環 R とそのイデアル I が与えられた時、 $x, y \in R$ に対し、 $x - y \in I$ であれば、 $x \sim y$ とします。この関係“ $\sim$ ”は同値関係になります。そこで、同値関係“ $\sim$ ”の同値類の集合 $R/\sim$ を R/I と記述します。この時、 R/I は環になり、この環の事を剰余環、或いは紛らわしいですが商環とも呼びます。

環 R が可換環の場合、イデアル I が素イデアルであれば R/I は整域になります。実際、 $\bar{x} * \bar{y} = 0 \Leftrightarrow x * y \in I$ より、 $x \in I$ か $y \in I$ 、即ち、 $\bar{x} = 0$ か $\bar{y} = 0$ の何れかが成り立ちます。更に、イデアル I が極大イデアルの場合、 R/I は体になります。

4.6 準同型写像について

群や環の準同型写像は群や環の性質を調べる上で、非常に性質の良い函数の事です。どの様に性質が良いか説明しましょう。

最初に群準同型写像について簡単に説明しましょう。まず、群 G_1 と G_2 が与えられ、それらの演算を各々 $*_1$ と $*_2$ とします。この時、群準同型写像 $f: G_1 \rightarrow G_2$ は演算を保ちます。即ち、 $f(a_1 *_1 b_1) = f(a_1) *_2 f(b_1)$ となります。次に、群準同型写像は単位元を保ちます。即ち、群 G_1 の単位元を 1_1 とすると、 $f(a_1) = f(1_1 *_1 a_1) = f(1_1) *_2 f(a_1)$ となる事から判ります。これを纏めると次の様になります。

$$f(a_1 *_1 a_2) = f(a_1) *_2 f(a_2) \quad (4.1)$$

$$f(1) = 1' \quad (4.2)$$

次に、環の準同型写像はもう少し複雑ですが、これも演算 $*$ と和 $+$ をそのまま保つ性質を持っています。今度は環 R_1 と R_2 が与えられ、各環の積と和を各々、 $(*_1, +_1)$ と $(*_2, +_2)$ とします。

この時、以下の性質を満たすものになります。

$$f(a_1 +_1 a_2) = f(a_1) +_2 f(a_2) \quad (4.3)$$

$$f(a_1 *_1 a_2) = f(a_1) *_2 f(a_2) \quad (4.4)$$

$$f(1) = 1' \quad (4.5)$$

猶、準同型写像 $f: A \rightarrow B$ が、写像として 1 対 1 の写像であれば単射、 $f(A) = B$ であれば全射、そして、単射かつ全射であれば同型と呼びます。

群や環の準同型写像 $f: A \rightarrow A$ は写像の合成 $\circ$ を積とする半群となります。実際、 f と g を A から A への準同型写像とすると、 $f \circ g$ は A から A の準同型で、 $(f \circ g) \circ h = f \circ (g \circ h)$ を満たします。更に、単位元 1 は恒等写像 $1: A \rightarrow A$ になります。

良く使われる用語で、準同型写像 $f: A \rightarrow B$ の核と呼ばれるものがあります。これは $\ker(f)$ と記述され、 B の単位元を 1_B とする時、準同型 f による 1_B の逆像 $f^{-1}(1_B)$ の事です。即ち、 $\ker(f)$ の元は準同型写像 f によって B の単位元 1_B に写されるものの集合です。この $\ker(f)$ は A, B が環の場合は環 A のイデアルになり、 A, B が群の場合は群になります。

ここで、数式の処理に戻ると、数式は多項式環に幾つかの函数を追加した環の元として考えられます。二つの異った書式の数式が等しいとは、式の変形、函数や変数に対して

与えた規則によって、共通の式に変形可能であるという同値関係“ $\sim$ ”をこの環に入れたものとして考えられます。従って、規則は自然な写像から誘導される準同型の核を構成するものとして考えられます。

4.7 数式の表現

与えられた数式が等しいとは、式の展開等の処理に加え、同値関係を利用する事で、互いに共通の式に変形出来る事だと述べました。では、計算機でどの様に処理すべきでしょうか？又、どうすれば効率良く処理が行えるでしょうか？この事を考える上で、式そのものの表現について考察する必要があります。

この節では、多項式環に含まれる式で、式の表現を考えてみましょう。先ず、式 $x+y$ と $y+x$ が与えられた場合、これらの式が等しいかどうかは、少なくとも計算機にとっても自明な事ではありません。第一、これらの式は計算機にとって単なる文字の羅列でしかありません。その上、配列にこれらの式を頭から入れた場合、 x と y の順序が逆なので一致しません。これらの式が等しいと計算機に認識させる為には、どのような規則が存在しているのかを計算機に教えてやる必要があります。

この場合、 $x+y$ と $y+x$ は被演算子の順序を変更したもののなので、被演算子の入換えが可能（演算子の可換性）と言う演算子の属性がなければ等しい式になりません。例えば、 x/y の場合は如何でしょうか？ x/y と y/x は一般的には異なった式になります。が、これは演算子/が非可換だからです。

この事から、計算機には、文字列として与えられたデータ（与件）の演算子を抽出し、演算子の属性から規定される規則を用いて式を変形して、二つの式が移り合えば等しい式と判断出来ます。

二つの式 A と式 B が式に含まれる演算子の性質を利用して、互いに変形可能な場合に $A \sim B$ と記述しましょう。すると、この二項関係“ $\sim$ ”は同値関係になります。先ず、最初の反射律ですが、これは同じ式の場合、入れ換え操作を行うまでもなく、同じデータ型となるので妥当です。対称律は式 A から B に変形可能なら、その逆操作を行えば良いので、これも大丈夫です。推移律は式 A から B への操作に続けて B から C への変換操作を行えば良いので、この操作で式 A から C に変換出来ます。

変形処理の関係“ $\sim$ ”による同値類の代表の選び方を考えましょう。 $x+y$ の場合、演算子を鍵として、演算子の属性（可換性）から被演算子を入換える規則で式の変形を行って、同じ式が得られれば等しいと判断しています。

そこで、非常に安易ですが、鍵となる演算子を先頭に置いて、“ $+ x y$ ”の様な文字列で計算機に処理させればどうでしょうか？勿論、“ $+ x y$ ”の様に表記していると複雑な

式の場合は流石に判り難くなるで、 $(+ x y)$ と括弧で括ってしまいましょう。何処かで見た事がありますね。LISP の S 式と同じ格好になっていますね。この $(+ x y)$ の様に演算子を前に置いた数式の表現を前置表現と呼びます。又、 $x + y$ の様に演算子が二つの被演算子の中に置かれた数式の表現を内挿表現と呼びます。

では、引数を増して $x + y + z$ はどうでしょう？ この式では最初の演算 $+$ の前の x が第一引数で $y + z$ が第二引数となりますね。先程の表記では、 $(+ x (y + z))$ になります。ここで部分式の $y + z$ も同じ考え方で処理すると $(+ y z)$ になります。即ち、部分式 $y + z$ の y は本体側の第二引数の中の第一引数、 z は本体側の第二引数に含まれる第二引数となります。この様に処理を行うと、 $(+ x (+ y z))$ が得られます。ここで、演算子を節に、被演算子を葉と看做せば、図 4.1 に示す木構造が数式に入ります。

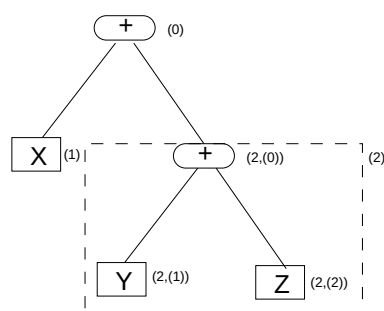


図 4.1: $X + (Y + Z)$ の木構造図

ここで演算子 $+$ はどのような性質を持っているのでしょうか？最初に可換性 $a + b = b + a$ があります。次に結合律 $(a + b) + c = a + (b + c)$ も成立します。この結合律がある御陰で括弧を外して $a + b + c$ と記述出来ます。だから、 $(+ (+ a b) c)$ と $(+ a (+ b c))$ も共通の形式で表現されるべきです。この $a + b + c$ は演算の括り方は問わず、その部分式は a, b, c のみの為、 $(+ a b c)$ と平坦に表記されるのが妥当です。これと似た演算子に可換積 $*$ があります。可換積 $*$ も同様に $x_1 * x_2 * \dots * x_n$ を $(* x_1 x_2 \dots x_n)$ と表記します。この考え方は Maxima の nary 型 演算子の扱いに反映されています。次に、差と商に関しては、 $a - b$ は $a + (-b)$ 、 a/b は $a * (1/b)$ で置換します。最後の冪に関しては、 a^b を $(^ a b)$ で表現すれば良いでしょう。

すると、数式は数、変数や函数に、それらで構成された部分式を演算子 $+$ や演算子 $*$ 等の演算子で結合したものとして表現出来ます。

例えば、 $1 + x + (y - 1)/(z^3 + 2xy)$ を LISP の S 式で表現すると、

$(+ 1 x (/ (+ y -1) (+ (^ z 3) (* 2 x y))))$ となり、その木構造図を見れば、

図 4.2 に示すものとなります.

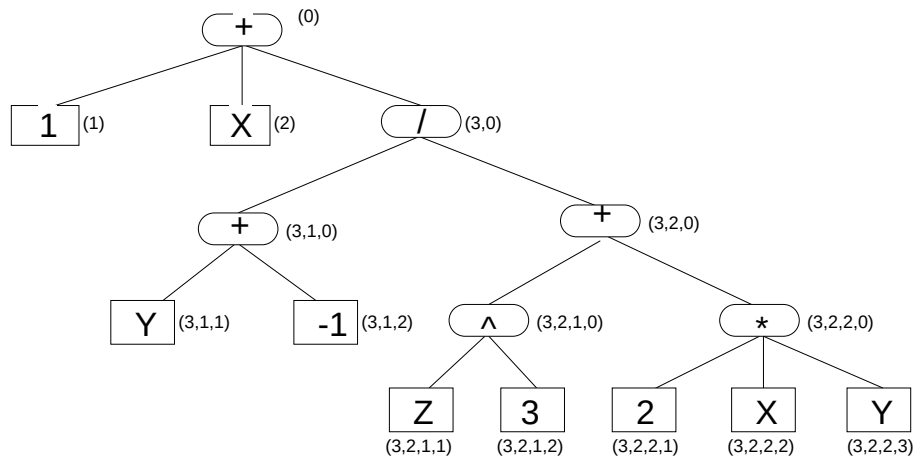


図 4.2: $1 + x + (y - 1) / (z^3 + 2 * x * y)$ の木構造図

木構造と S 式は 1 対 1 に対応が付きませんが, 式には項の順番の問題があって一意に定まりません. 例えば, $x + y + z$ は項の並び換えにより, $y + x + z$ や $x + z + y$ 等の 6 通りの並び方があります. 計算機で計算させる場合, この様に式が一意に決らないのは処理をする上でも困ります. $x + y + z$ の第一項 x に 1 を代入し…といった処理を行う場合でも, 並びが一意に決ってしまえば, 第一成分に 1 を代入するとか簡単に出来ますが, そうでなければ, 項を頭から一々確認しなければなりませんね. 実用的な意味でも, ある規則で一意に決定する様にしておけば, 計算機にその規則を教えてやると, 後は安心して処理をさせられます. ではどうすれば良いのでしょうか?

そこで, 項に順番を入れましょう. すると, その順序に対して一意に数式の並びが決定されるので, 同時に数式の表現も一意に定まります.

例えば, 変数をアルファベット順に並べる様にすればどうでしょう. $y + z + x$ は $x + y + z$ になり, 式の表示が一通りに決りますね. これは結構良さそうですね.

では次の節では変数の順番について考えましょう.

4.8 順序について

数学では順序という考えがあります. この順序は集合 S の二つの元に対して成立する関係 (二項関係と呼びます) の一つです. 例えば, 集合を実数とすると, 大小関係 “ $\geq$ ” は

順序関係の一例です. この大小関係を一般化したものです.

ここで, 集合 S の二項間の関係 " $\geq_{\text{order}}$ " が以下の性質を満たす時に順序と呼び, 順序の入った集合 S の事を順序集合と呼びます.

順序の定義

反射律: $x \geq_{\text{order}} x$

対称律: $x \geq_{\text{order}} y$ かつ $y \geq_{\text{order}} x \Rightarrow x = y$

推移律: $x \geq_{\text{order}} y$ かつ $y \geq_{\text{order}} z \Rightarrow x \geq_{\text{order}} z$

集合 S の順序 " $\geq_{\text{order}}$ " で集合の任意の元 x, y に対し, $x \geq_{\text{order}} y$, 或いは $y \geq_{\text{order}} x$ の何れかが成立する場合, 順序 " $\geq_{\text{order}}$ " を全順序, 集合 S の事を全順序集合と呼びます. 例えば, 集合を実数 $\mathbb{R}$ とすると, 二項関係 " $\geq$ " は全順序, 集合 $\mathbb{R}$ が全順序集合となります. 但し, 二項関係を包含関係 " $\supseteq$ " とした場合, 包含関係 " $\supseteq$ " は集合 $\mathbb{R}$ の順序になりますが, 全順序にはなりません. 何故なら, 偶数全体の集合 A と奇数全体の集合 B を考えるとどうでしょうか? 集合 A と B の間には包含関係がありませんね. 全順序となる為には包含関係が無ければなりません. 但し, この包含関係の様に, $a \subseteq b, a \supseteq b$ の何れかが成立しない順序の事を半順序と呼びます.

数式処理の場合, 集合は数, 変数や関数で構築された式の集合を考えます. 式は和や積の様に可換な演算で結合された数, 変数や関数の為, 数と変数には変数の方が大きいと順序付け, 変数よりも関数の方が大きいと順序付け, 変数同士にも順序を入れると, 後は項の順序が残ります.

1 変数の場合, 単項式 x^m と x^n との順序を冪の次数 m と n の大小関係 " $\geq$ " で決めれば良いでしょう. では, 多変数の場合はどうすれば良いのでしょうか? この場合, 変数の間に順序を入れ, その変数の順序で変数を入れ換えた項の次数リストで比較する方法があります.

例えば, x, y, z の 3 変数多項式環 $K[x, y, z]$ で, 変数間の順番をアルファベット順で並べ, yz の様に変数が抜けている場合, 抜けている変数の次数を 0 とすれば, 項は $x^{i_1}y^{i_2}z^{i_3}$ の形式になります. これから長さ 3 の次数のリスト (i_1, i_2, i_3) が得られ, このリストと項は一対一に対応します. 従って, n 変数 $x_1, \dots, x_n$ の単項式 A と B が与えられた時, n 個の変数 $x_1, \dots, x_n$ の並びを固定します. 次に, 単項式 A と B の中の変数 x_i が項に存在しない場合, x_i^0 を挿入します. こうする事で, 単項式 A と B を表現する長さが n の次数リスト $(\alpha_1, \dots, \alpha_n)$ と $(\beta_1, \dots, \beta_n)$ が一意に決ります. 後はこれらの次数リストに対して順序を入れれば良い事になります. ここで, 項の順序としては, 二つの項 x^α, x^β に対して, $x^\alpha > x^\beta$ であれば, x^γ を両辺にかけた場合でも, $x^{\alpha+\gamma} > x^{\beta+\gamma}$ を満たすものの方が都合が良いですね. この項に対する性質 $x > y \Leftrightarrow x \cdot a > y \cdot a$ を満

す順序の事を項順序と呼びます.

では, 先程の例の二つの次数リスト $x^2 y^2 z (= (2, 2, 1))$ と $x y^2 z^3 (= (1, 2, 3))$ が得られた時, 順序はどの様に入れれば良いでしょうか. 単純に x の多項式と見れば, $x^2 y^2 z$ の方が大きく, 次に z の多項式と見れば, $x y^2 z^3$ の方が大きくなります. ところが, $x = y = z$ として x の多項式として変形すると, $x y^2 z^3$ の方が次数が大きくなります. この場合は, 次数の大きさも含めて考えた方が良さそうですね. この様に多項式の場合, 項の順序には色々な考え方があります. 次の小節では代表的な項の順序について説明しましょう.

4.8.1 色々な順序

項順序として代表的なものに, 辞書式順序, 斉次辞書式順序, 逆辞書式順序, 逆斉次辞書式順序, これらの順序に, 変数の重みを加味したもの等があります. ここでは, 基本的な辞書式順序, 斉次辞書式順序, 逆辞書式順序と斉次逆辞書式順序について簡単に説明しましょう. 猶, 変数の並び順を $x_1, \dots, x_n$ とし, $x_1^{a_1}, \dots, x_n^{a_n}$ の次数リストを $a = (a_1, \dots, a_n)$ $x_1^{b_1}, \dots, x_n^{b_n}$ の次数リストを $b = (b_1, \dots, b_n)$ とします. 更に, $|a| = a_1 + \dots + a_n$ で次数リスト a に含まれる次数の総和を表記します. 又, 順序の定義に含まれる $>$ は通常の整数環 $\mathbb{Z}$ の元に対する大小関係です.

辞書式順序 $>_{\text{lex}}$

$$a >_{\text{lex}} b \Leftrightarrow a_1 = b_1 \quad \dots \quad a_i = b_i \text{ かつ } a_{i+1} > b_{i+1}$$

この関係で定められる順序を辞書式順序と呼びます. この順序を示す記号として, ここでは $>_{\text{lex}}$ を使います.

辞書式順序では二つの項を比較した時に, 左側の変数の次数が大きなものが大きな項となります.

例えば, $x^2 y^2 z$ と $x y^2 z^3$ を各々表現する整数リスト $(2, 2, 1)$ と $(1, 2, 3)$ に対しては, 先頭の 2 と 1 を比較した段階で, $2 > 1$ から $(2, 2, 1) >_{\text{lex}} (1, 2, 3)$, 即ち, $x^2 y^2 z >_{\text{lex}} x y^2 z^3$ となります.

斉次辞書式順序 $>_{\text{glex}}$

$$a >_{\text{glex}} b \Leftrightarrow \begin{cases} |a| > |b| \text{ または} \\ a_1 = b_1, \dots, a_i = b_i, a_{i+1} > b_{i+1} \end{cases}$$

この関係で定められる順序を斉次辞書式順序と呼びます. この順序を示す記号として, ここでは $>_{\text{glex}}$ を使います.

この斉次辞書式順序の場合、総次数で最初に項を比較し、総次数が一致すれば、今度は項を辞書式順序で比較する二段方式となっています。

例えば、 x^2y^2z と xy^2z^3 を各々表現する整数リスト $(2, 2, 1)$ と $(1, 2, 3)$ に対し、 $|x^2y^2z| = 5$ と $|xy^2z^3| = 6$ となるので、 $(1, 2, 3) >_{\text{glex}} (2, 2, 1)$ 、即ち、 $xy^2z^3 >_{\text{glex}} x^2y^2z$ となり、辞書式順序とは逆の結果になります。

逆辞書式順序 $>_{\text{revlex}}$

$$a >_{\text{revlex}} b \Leftrightarrow a_n = b_n \cdots a_i = b_i \text{ かつ } a_{i-1} < b_{i-1}$$

この関係で定められる順序を逆辞書式順序と呼びます。この順序を示す記号として、ここでは“ $>_{\text{revlex}}$ ”を使います。辞書式との違いは、先ず、調べる方向が逆で、次数の小さい方を取る点で、逆になっています。

具体的には、二つの項 $x^3y^2z^3$ と xy^2z^3 が与えられた場合、各々の項を表現する整数の列 $(3, 2, 3)$ と $(1, 2, 3)$ が得られます。逆辞書式順序では、この列の後から比較を行います。すると、後から先に移動して、リストの先頭の 3 と 1 に到達すると、定義から $(1, 2, 3) >_{\text{revlex}} (3, 2, 3)$ 、即ち、 $xy^2z^3 >_{\text{revlex}} x^3y^2z^3$ を得ます。

斉次逆辞書式順序 $>_{\text{grevlex}}$

$$a >_{\text{grevlex}} b \Leftrightarrow \begin{cases} |a| > |b| \text{ または} \\ a_n = b_n, \dots, a_i = b_i, a_{i-1} < b_{i-1} \end{cases}$$

この関係で定められる順序を斉次逆辞書式順序と呼びます。この順序を示す記号として、ここでは“ $>_{\text{grevlex}}$ ”を使います。

この順序は項の総次数で最初に項を比較し、総次数が等しければ、逆辞書式順序で項の順序を決める二段方式のものです。

x^2y^2z と xy^2z^3 の場合、総次数が各々、5 と 6 になるので、 $xy^2z^3 >_{\text{grevlex}} x^2y^2z$ となります。因に、逆辞書式の場合は、次数リストの右端の z の次数から見るので、 $x^2y^2z >_{\text{revlex}} xy^2z^3$ となり、斉次逆辞書式順序とは逆の結果となります。

これらの項順序は多項式の計算で非常に重要です。特に Groebner 基底の計算では、項順序を用いた計算を行います。この Groebner 基底は項順序に対しては一意に決まりますが、順序を替えると、普通は別の基底が得られます。この事もあって為、順序は新しい数式処理システムでは目的に応じて様々な順序が扱える様になっています。

猶、代数学の様々な定理は、対象に順序を入れる事で証明出来るものが多いです。その意味でも、項順序は非常に重要です。

4.9 多項式の表現

ここでは多項式の表現について、再度、考えてみましょう。

変数に順序を入れ、その順序で並べた項に対しても順序を入れました。これで項を順番に並べてしまえば、与えられた数式の表記に対応する前置表現により、式が一意に決定します。但し、ここでの前置表現は与えられた式そのものを単純に置換えるだけで、関係“ $\sim$ ”で同値な式が全て同じ前置表現で置換えられる訳ではありません。その為、多項式は予め展開しておき、項毎に簡易化を行っておく必要があります。この簡易化を行えば、少なくとも、式の変形操作による同値関係“ $\sim$ ”に関しては前置表現による式と本来の式が一对一に対応します。

とは言え、内挿表現は無駄が多い表現です。これをより簡単にしたもの、正準表現というものがあります。この正準表現を解説する為に、多項式 $3x^2 - 1$ を使って正準表現を構成を説明しましょう。

先ず、与式が何の変数の多項式であるかが重要ですね。その重要なものを内挿表現から前置表現に変換する時と同じ様に先頭に変数 x を置きましょう。次に、この式は $3x^2 + (-1)x^0$ と同値になります。するとこの式は x の多項式としては2次の項と0次の項があり、2次の項の係数は3で、0次の係数の項は-1ですね。そこで $(x \ 2 \ 3) \ (0 \ -1)$ と記述すればどうでしょう。前置表現であれば、 $(+ \ (* \ 3 \ (^ \ x \ 2)) \ -1)$ となり、随分とすっきりしますね。更に、内部の括弧は冪と係数の対で表現されると決まっているので、 $(x \ 2 \ 3 \ 0 \ -1)$ で十分です。そして、この表現 $(x \ 2 \ 3 \ 0 \ -1)$ と多項式 $3x^2 + (-1)x^0$ は一对一で対応します。この様に、式と表現の関係が一对一になる表現を正準表現と呼びます。

この方法を単変数多項式に適用すれば、以下の正準表現が得られます。

—— 単変数多項式の正準表現 ——

$(\langle \text{変数} \rangle \langle \text{次数}_1 \rangle \langle \text{係数}_1 \rangle \langle \text{次数}_2 \rangle \langle \text{係数}_2 \rangle \cdots)$

これで、一変数の場合は片付けました。では一般の多変数多項式の場合はどうでしょうか？ 実際、多変数多項式 $K[x_1, \dots, x_n]$ に対しても、同様の手法で正準表現が構成出来ます。但し、多変数多項式の場合で重要な事は、項に順序を入れる事です。そこで、辞書順序 $>_{\text{lex}}$ を多項式環 $K[x_1, \dots, x_n]$ に入れてみましょう。

先ず、変数の並びを $x_1, \dots, x_n$ で固定し、項 $x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ を次数リスト $(\alpha_1, \dots, \alpha_n)$ で表現します。二つの項 $x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ と $x_1^{\beta_1} \cdots x_n^{\beta_n}$ の比較は、次数リストを用いて行います。この際、リストの左端から順番に比較しますが、 $\alpha_1 = \beta_1, \dots, \alpha_{i-1} = \beta_{i-1}$ で、 $\alpha_i > \beta_i$ の場合、 $x_1^{\alpha_1} \cdots x_n^{\alpha_n} >_{\text{lex}} x_1^{\beta_1} \cdots x_n^{\beta_n}$ となります。

次に、多項式 $\sum a_{(i_1, \dots, i_n)} x_1^{i_1} \cdots x_n^{i_n}$ が与えられると、最初に x_1 の多項式と看做して、

一変数の場合と同様の考え方で表現を構成します. 先ず, 式に含まれる変数が x_1 だけの場合は, 一変数の方法で正準表現が得られます. もし, x_1 以外の変数が存在する場合は, 最初に x_1 を変数とする多項式と看做して式を纏めます. すると, x_1 の各項の係数は高々 $n-1$ 変数の多項式 ($\in K[x_1, \dots, x_{n-1}]$) となります. 今度は各係数に対し, 同様の考え方で x_2 の多項式表現を構成します. 以降, 係数に対して帰納的に処理を行う事で, 以下の様な正準表現が得られます.

— 多変数多項式の正準表現 —

($\langle$ 変数 $\rangle \langle$ 次数 $_1 \rangle \langle$ 係数多項式の正準表現 $_1 \rangle \langle$ 次数 $_2 \rangle \langle$ 係数多項式の正準表現 $_2 \rangle \dots$)

具体的に多項式環 $\mathbb{Z}[x, y]$ の元 $yx + 2xy^3 - 3$ に対し, 順序を辞書式順序 $>_{\text{lex}}$ で考えましょう. 猶, 変数の並びは x, y とします. この場合, 最初に与式を x の多項式と看做し, 変数 x で式を纏めましょう. その結果 $(2y^3 + y)x - 3$ が得られます. そこで, 第一段目は $(x \ 1 \ 2y^3 + y \ 0 \ -3)$ となります. 次に係数の処理に移ります. この場合, リストの係数各成分の処理を行います. そこで, 係数を x の次の変数 y の多項式として書き直します. すると, 第二成分の $2y^3 + y$ が $(y \ 3 \ 2 \ 1 \ 1)$ で置換えられるので, 最終的に $(x \ 1 \ (y \ 3 \ 2 \ 1 \ 1) \ 0 \ -3)$ が $yx + 2xy^3 - 3$ の正準表現として得られます. この様に, 順序を決めていけば, 正準表現が得られますが, 変数の並びや順序を変更する事で, 同じ多項式でも表現が一般的に異なります.

4.10 Gröbner 基底の紹介

この節では Gröbner 基底について述べます. Gröbner(標準)基底考え方そのものは, 広中先生の標準基底 (standard basis) として知られていましたが, Gröbner 基底の計算アルゴリズムは Buchberger によるものです. この Buchberger の Gröbner 基底を計算するアルゴリズムを簡単に言えば, 多項式に対して Euclid の互除法を適用し, 生成元の最大公約数を計算する手法に似ています.

先ず, 体係数の多項式環 $K[x_1, \dots, x_n]$ の任意のイデアルは有限生成となる事が知られています (Hilbert の基底定理). その為, 任意のイデアルには有限個の生成元が必ず存在します. 特に, 1 変数多項式環 $K[x]$ の場合, 任意のイデアルが単項イデアルとなるので, $K[x]$ は単項イデアル整域 (PID) になります.

ここで $K[x]$ の任意のイデアルは, そのイデアルに含まれる最も低い次数の多項式で生成されます. この最も低い次数の多項式の存在は次の様にして分ります. 最初に次数による順序 $>_{\text{deg}}$ を多項式の間に入れます. この時, 写像 $\text{deg} : K[x] \rightarrow \mathbb{N}$ から得られる自然数 $\mathbb{N}$ の部分集合 $\text{deg}(K[x])$ は下に有界な空でない集合になります. この写

像 $\deg$ の定義域を $K[x]$ のイデアルに限定した場合, 必ず最小値が存在する事が Zorn の補題によって保証されます. この最小値を与える多項式が f と g の二つが存在した場合, f と g を簡単の為に, 最大次数の項の係数を 1 とすると, $f - g$ はより小さな次数の多項式となります. この $f - g$ もイデアルの元でなければなりません, $f - g$ が 0 でなければ f や g よりも小さな次数の元が存在する事となって矛盾が生じます.

更に, h をイデアルに含まれる元とすると, h は f で割切れなければなりません. 何故なら, $h = fk + r$ となる場合, r が 0 でなければ, f がイデアルの最小次数の元である事に反するからです.

この様に, 体 K を係数とする一変数多項式環 $K[x]$ は単項イデアル整域になる事が分りました. ところが, 多変数多項式環の場合は事情が異なります. 多変数の多項式環ではイデアルを生成する多項式系は一つであるとは限りません. 例えば, 一次の連立方程式 $[2x + y = 4, x - 3y = -5]$ を考えてみましょう. この方程式の解は $[x = 1, y = 2]$ となりますね. ここで頭を切り替えて, 二つの多項式 $p_1 = 2x + y - 4$ と $p_2 = x - 3y + 5$ で生成されるイデアル E_q として考えてみましょう. 次に Gauss の消去法を思い出して下さい. さて, p_1 から変数 x を消去してしまいましょう. これは $p_1 - 2p_2$ を計算すれば可能です. すると, $7y - 14$ が得られます. この式はイデアル E_q の元になりますね. 更に, 係数環が体なので, 全体を 7 で割った $y - 2$ もイデアル E_q に含まれます. 今度は p_2 から変数 y を取り除いてみましょう. これは $p_2 + 3(y - 2)$ を計算すれば出来ますね. これから直ちに, $x - 1$ が得られますが, この式も勿論, イデアル E_q の元になります.

今度は $E_{q_2} = \langle x - 1, y - 2 \rangle$ としましょう. すると, E_{q_2} の生成元は多項式 p_1, p_2 で生成されるので, $E_{q_2} \subset E_q$ となりますが, 一方で p_1 と p_2 はイデアル E_{q_2} の元の線形結合で記述されるので, 結局, $E_q = E_{q_2}$ となります.

ここでは連立一次方程式を例に出しましたが, 連立一次方程式の解を計算する事は, イデアルとか環といった言葉を使うと, 与えられた方程式を構成する多項式で生成されるイデアルからより単純な形, この場合, 変数が $x_1, \dots, x_n$ であれば, $x_1 - a_1, \dots, x_n - a_n$, 但し, $a_1, \dots, a_n \in K$ の形のイデアルの生成元を計算する事へと言い換えられます.

従って, 多項式で構成された連立方程式に対し, そのイデアルを考える事で多数の複雑な式から, より計算し易い生成元で置換えてしまえば, その方程式の根を求める数値計算さえも, より簡単, 且つ, 効率的に行える可能性がある事を意味します.

ではどの様なイデアルの生成元がより計算し易いと言えるでしょうか? 単純に考えるのであれば, 新しい生成元は前の生成元と比べて, その次数が低いもので, 生成元が他の生成元の線形結合で生成されないもの, 即ち, 線形独立であれば良いでしょう.

その前に筆頭項 (Leading Term) を説明しておきましょう. 先ず, 多項式環に入れられた順序を使って, 多項式 f の最も大きな項の事を筆頭項 (Leading Term) と呼び, を

$LT(f)$ (Leading Term) と表記します. そして, その項から得られる単項式を筆頭単項式 $LM(f)$ (Leading Monomial) と表記します.

そして, 多項式環の部分集合 S に対して $LM(S)$ を, S に属する多項式 f の $LM(f)$ から生成されるイデアルとして定義します. ここで Gröbner 基底を導入しましょう.

Gröbner 基底の定義

多項式環 $R[x_1, \dots, x_n]$ のイデアル I に対して, イデアル I に含まれる多項式の集合 $S = \{g_1, \dots, g_m\}$ が以下の条件を満たす時, 集合 S をイデアル I の Gröbner 基底と呼びます.

- $I = \langle g_1, \dots, g_m \rangle$ ($g_1, \dots, g_m$ が I を生成)
- $LM(I) = LM(S)$

この定義からも判る様に, Gröbner 基底は多項式環に入れた順序に依存する為, 順序が異なれば, 一般的にイデアルを生成する多項式系も異なる性質を持っています.

4.11 集合論の話題から

4.11.1 色々な数

複素数を多項式環 $\mathbb{R}/\langle x^2 + 1 \rangle$ で表現する話をしましたが, その時, 純虚数は代数的整数と呼ばれる集合に属すると説明しました. 復習をすると, 代数的整数は最高次数の係数が 1 となる整数係数多項式 (この様な多項式の事を monic な多項式とも呼びます) の解となる数の事です. 例えば, 任意の整数 n は方程式 $x - n = 0$ の解となるので代数的整数になります. 更に, $x^2 - 2 = 0$ の解となる $\sqrt{2}$ も代数的整数になります. この様に代数的整数は整数の集合 $\mathbb{Z}$ を含む集合ですが, 純虚数も含むので, 複素数 $\mathbb{C}$ の部分集合になります.

この代数的整数に加え, 最高次数項の係数が 1 であるとは限らない整数係数多項式の解となる数の事を代数的数と呼びます. 又, 整数係数多項式の事を代数方程式と呼びます. 代数的数には有理数が勿論含まれます. 又, 方程式 $2x^2 + 1 = 0$ の解 $\frac{i}{\sqrt{2}}$ は代数的整数になりませんが, 代数的数になります. ここで a を代数的数とすると, この a には定義から a を解に持つ代数方程式が必ず存在します. その様な多項式の中で最小冪の多項式が存在します. この最小冪の整数係数多項式の事を最小多項式と呼びます. 例えば, i は, $x^4 + x^2 = 0$ の解ですが, i の最小多項式は $x^2 + 1$ です. この最小多項式で

生成される多項式環 $\mathbb{Z}[x]$ のイデアルに $x^4 + x^2$ が含まれます。即ち、代数的数 a の最小多項式は、 a を解に持つ多項式から生成されるイデアルの生成元になります。

全ての数が代数方程式の解であれば話は楽ですが、実際は代数方程式では表現出来ない数が存在します。この様に代数的方程式の解にならない数の事を超越数と呼びます。超越数の有名なものには、円周率 π や自然底 e といった数があります。そして、これらの超越数を計算する為には級数の計算が必要になりますが、何れにしても有理数ではないので計算出来るのは近似値でしかありません。

その意味では代数的数は有理数係数の代数方程式と関連する為に、有理数の様な扱いは出来ないにしてもより明確なものと言えるでしょう。例えば、 $\sqrt{2}$ は代数方程式 $x^2 - 2 = 0$ の正の解になりますが、有理数環 $\mathbb{Q}$ に $\sqrt{2}$ を追加した環 $\mathbb{Q}[\sqrt{2}]$ は $\mathbb{Q}[x]/(x^2 - 2)$ に対応するので、超越数と比較して格段にすっきりとした扱いが出来ます。

因に、19世紀では確実なものは自然数と有理数だけで、無理数を疑いの目で見ている人も居る程です³。代数的数は出所の明確な有理数を使って定義する事が可能な数なので、その意味では、まだ受容されていた方です。

この様に実数 $\mathbb{R}$ は分数で表現可能な数の有理数 $\mathbb{Q}$ と分数で表現出来ない数の無理数に分類出来ますが、整数係数の代数方程式を介在させる事で、代数的整数で実数に含まれるものと超越数に更に分類する事も可能なのです。

4.11.2 濃度

実数 $\mathbb{R}$ は有理数と無理数に分類されます。そして、無理数には代数的数の一部や超越数が含まれています。そこで、素朴な疑問ですが、整数、代数的数、そして超越数の集合の中で、どの集合の個数が一番多いのでしょうか？どれも沢山ありそうです！その上、これらの集合は少なくとも有限ではありませんね。

実際、整数は1を加え続ける事で、幾らでも大きな整数が作られるので有限ではありません。又、 $\sqrt{2}$ に整数を掛けた数は代数的数になるので、代数的数は整数よりは沢山ありそうです。又、 π に整数を掛けたものも超越数なので、こちらも整数よりは沢山ありそうです。ではどうやって比較するのでしょうか？

そこで、手始めに有限集合の場合を考えてみましょう、二つの有限集合 A, B が与えられた場合、 A と B の元の個数を数えますが、これは集合 A と集合 B の一つ一つの元に1から順番に自然数を対応させる事です。すると自然数の最大値がその集合の個数になります。この事は、集合 A, B と一対一の関係がある自然数 $\mathbb{N}$ の部分集合 S_A と S_B

³自然数、全部創ったのは神様、その他全部は人様の創作 (Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.) - Leopold Kronecker

を考え、 S_A と S_B の比較に置換えているのです。ここで、集合 A と B が同じ個数である事は、集合 A から B への一対一の写像が存在する事と言い換えられます。もし、集合 A の方が集合 B よりも大きければ、集合 A に対応する自然数の集合 S_A の方が S_B よりも大きくなります。そして、集合 B から集合 A への自然な単射が存在する一方で、逆向きの単射が存在しない事も分ります。

この方法を一寸乱暴かもしれませんが、無限集合にも適用してみましょう。そこで、自然数 $1, 2, 3, \dots$ と平方数 $1, 4, 9, \dots$ を比較します。平方数の集合は明らかに自然数全体の集合に含まれており、更に、5 は平方数の集合に含まれない事から、平方数の集合は自然数全体の集合の真部分集合になります。ところが、自然数 n に対して平方数 n^2 を対応させてみましょう。この写像は自然数の集合から平方数の集合に一対一の写像になります。これは有限個の集合同士であれば、同じ個数である事を意味します！

この例は Galileo の新科学対話の中に出てくるものです。Euclid の幾何学原論で述べられている命題「部分は全体よりも小である」に明らかに反する事です！ところで、Galileo は、「等しい」、「大きい」、「小さい」といった事は有限個のもの同士の比較でのみ言える事で、無限個のものには使えないと述べ、それ以上の言及をしていません ([10], pp.59-61)。この無限により深く入った人は 19 世紀末から 20 世紀初頭に活躍した Cantor です。

さて、集合論では自然数の集合 $\mathbb{N}$ との間の一対一写像が定義出来る集合の事を可附番集合、或いは可算無限集合と呼びます。先程の自然数と平方数の例では、写像 $f: x \rightarrow x^2$ で対応が付き、一対一写像なので、平方数の集合は可附番集合になります。

ここで、集合の元の個数の事を濃度、又は基数と呼びます。ここで濃度と基数の違いですが、濃度は Cantor が集合の元の個数として用いていますが、Frege や Russell の基数は正確には「命題の外延 (クラス) と一対一の写像を持つ命題の外延で構成される外延」ですが現在は同じ意味で用いています。

有限個の集合であれば濃度は自然数になりますが、可算無限集合の場合は可附番濃度、可算濃度や可算個等と呼び、ヘブライ文字の $\aleph$ (アーレフ) を用い、 $\aleph_0$ (アーレフ・ゼロ) でその濃度を表記します。因に $\aleph$ はヘブライ文字の第一番目の文字ですが、無音の文字です。因に、ヘブライ語で $\aleph$ は数字の 1 や牛を意味し、更に、カバラでは空気を意味するそうです。

この $\aleph_0$ に等しい濃度を持つ集合としては、先程の平方数の集合に加え、偶数の集合や奇数の集合といったものが挙げられます。

では、有理数 $\mathbb{Q}$ の濃度はどうでしょうか？ 有理数は $\mathbb{Z} \times (\mathbb{N} - \{0\})$ として考える事が出来ましたが、その濃度も $\aleph_0$ になります。

更に、 $\mathbb{N} \times \dots \times \mathbb{N}$ の濃度も $\aleph_0$ になる事が知られています。更に、代数的数は代数方程式に対応する為、方程式の係数から有限個の整数で構成されたリストが得られます。この

様に、代数的数は有限個の整数リストに対応させる事が可能になるので、その濃度も $\aleph_0$ になります。

次に、実数 $\mathbb{R}$ とその部分集合となる開区間 $(0, 1)$ の関係はどうでしょうか。これは以下の一対一の写像 f が存在するので同じ濃度になります。

$$f(x) = \frac{1}{1-x} : (1, 0) \rightarrow \mathbb{R}$$

この様に全体と等しい濃度を持つ真部分集合が存在する集合の事を Dedekind 無限集合と呼びます。と、何とも凄い事になりましたね。

それでは開区間 $(0, 1)$ と自然数 $\mathbb{N}$ の関係はどの様になるでしょうか。実は開区間 $(0, 1)$ の方が自然数の集合 $\mathbb{N}$ よりも圧倒的に大きな集合なのです。この事は Cantor の対角線論法と呼ばれる非常に有名な手法を用いて示す事が出来ます。これは先ず、開区間 $(0, 1)$ が可符番であったと仮定します。すると、開区間 $(0, 1)$ の元を以下の様に並べる事が出来る事になります。

$$\begin{array}{rcl} 1 & 0.x_{11}x_{12}x_{13}\cdots x_{1n}\cdots \\ 2 & 0.x_{21}x_{22}x_{23}\cdots x_{2n}\cdots \\ \vdots & \ddots \\ n & 0.x_{n1}x_{n2}x_{n3}\cdots x_{nn}\cdots \\ \vdots & \ddots \end{array}$$

それから新しい数列 $0.y_1y_2\cdots$ を各 y_m が m 番目の数の小数点下第 m 桁と異なる様に設定します。即ち、この数の小数点下 k 番目の成分は、上の表の k 番目の対角成分と異なる数になりますね。従って、この数はこの表には現われません。何故なら、この数が表の k 番目に出る場合、小数点下第 k 桁目の数 y_k は定義の方法からこの表の x_{kk} とは異なっていなければならないからです。この矛盾が得られた事から、開区間 $(0, 1)$ は可算では無く、自然数よりも大きな濃度を持つ集合である事が分ります。従って、開区間 $(0, 1)$ と同じ濃度になる実数 $\mathbb{R}$ の濃度を $\aleph$ と記述します。因に、集合 $\mathbb{R}^n$ の濃度も $\aleph$ になる事が知られています。

ここで、任意の 1 以上の整数 n に対して、 $\mathbb{R}^n$ の濃度が $\aleph$ になるという事は、次元の考え方が一見すると無意味なものに思えます。但し、この写像は連続性という性質を持たないものの為、その意味で次元というものは意味を持ちます。猶、この様な写像による像の次元はフラクタル次元というものがあります。この次元は通常の整数値の次元ではなく、実数値の次元を持ちます。例えば、平面を埋め尽す曲線の例として Peano 曲線というものがありますが、この Peano 曲線に対してもフラクタル次元を定める事

が可能で、その値は 1 と 2 の中間の実数値になります。

4.11.3 自然数

では、ここで何気なく使っている自然数 $\mathbb{N}$ はどのような数なのでしょうか？兎に角、自然数は代数的整数の一部ですねえ…。そして、この自然数を料理する事で色々な数が出来るだけではなく、可附番集合の定義では、自然数の集合 $\mathbb{N}$ との一対一対応となる写像が存在するとか、何かと物事の基礎にある数です。その割には $1, 2, 3 \dots$ と無数に続く数として漠然としか考えておらず、既知のものとして安心して使っている数ですね。この様に 20 世紀初頭迄は、自然数を特徴付ける公理も特に無く、悪く言えば曖昧な非常に数だったのにも関わらず、その上に壮麗な数学が構築されているという砂上の楼閣の如く、何とも危い事になっていたのです。

この自然数に関する基礎付けは Leibniz も行っているそうです。19 世紀末に Frege は「算術の基礎」[30][20]にて Leibniz による $2+2=4$ の証明を補完しています。先ず、Leibniz は 2, 3, 4 を次に様に定義しています。

- 2 を $1+1$ と定義する
- 3 を $2+1$ と定義する
- 4 を $3+1$ と定義する

それから公理として、「同じもので置き換えを行っても式は同じである」と述べ、 $2+2=4$ の証明を行っています。

この $2+2=4$ の証明の概要ですが、式 $2+2$ の演算子 $+$ の右辺の 2 は上の自然数 2 の定義から $1+1$ になります。従って、 $2+2=2+1+1$ が得られます。すると自然数 3 の定義から $2+1+1=3+1$ となり、最後に自然数 4 の定義から目出たく 4 が得られると Leibniz は述べています。

ところが、Frege はこの「証明」には演算子 $+$ の結合律が抜けているので、 $2+2=2+1+1=3+1$ と結論付けるのは正しくなく、 $2+2=2+(1+1)$ から結合律により $2+(1+1)=(2+1)+1$ 、そして、 $(2+1)+1=3+1$ と修正しています。

更に、Frege はこの「算術の基礎」[30]で彼の論理主義による自然数の構築を実行しています。この概要については後の §4.12.3 で述べる事としましょう。

さて、この Frege と独立して Peano は自然数の公理を構築しています。猶、Peano の 1889 年の論文「The principles of arithmetic, presented by a new method」([51]に英訳が収録)に現在の数理論理学の源流とも言える記号を用いて記述していますが、この

公理化での自然数は1から開始する数です. 現在は0から開始する数として次の様に公理化を行っています:

— Peano の公理系 —

自然数全体の集合 $\mathbb{N}$ は次の5条件を満たす.

1. $0 \in \mathbb{N}$
2. $x \in \mathbb{N}$ ならば, $x + 1 \in \mathbb{N}$
3. $x \in \mathbb{N}$ ならば, $x + 1 \neq 0$
4. $x + 1 = y + 1$ ならば, $x = y$
5. $0 \in M$, かつ $x \in M$ かつ $x + 1 \in M$ ならば $M \supset \mathbb{N}$

ここで自然数 n に1を加える関数 s と表記すると, 任意の自然数 n は0に n 回, 関数 s を作用させて得られた数と言えます. そして, この自然数 $s(n)$ の事を自然数 n の後者と呼びます.

又, この公理系の最後の性質が帰納法の原理となっています.

実際, 自然数 n に対する命題を $P(n)$ とすると帰納法は以下のものになります:

— 数学的帰納法 —

1. 命題 $P(0)$ が真である.
2. 任意の自然数 k に対して命題 $P(k)$ が真であれば, 命題 $P(k + 1)$ も真である.

これらの二つの条件が証明されていれば, 全ての自然数 n に対して, 命題 $P(n)$ が成立する.

さて, 数学的帰納法の原理で出て来る集合 M を命題 $P(k)$ が真となる自然数の集合としましょう. すると原理の $0 \in M$ は「 $P(0)$ が真」に対応しますね. 更に, $x \in M$ かつ $x + 1 \in M$ は $P(k)$ が真であれば $P(k + 1)$ も真である に対応し, 最後の $M \supset \mathbb{N}$ は「全ての自然数 n に対し命題 $P(n)$ が真である」に対応します.

4.11.4 超限順序数

自然数には色々な意味を持たす事が出来ます. 例えば, 袋の中の林檎の個数は, 「ひとつ」, 「ふたつ」, …と数えます. この個数に対応するのが基数, 又は濃度です. 又, 競争の順序の様に, 「いちばん」, 「にばん」, …と順序に対応するものがあります. こちらは順序数と呼びます. こうすると, 個数としての 2 と順序としての 2 は字面が同じなので区別が付きませんね. この辺をすっきりさせてみましょう.

さて, 自然数は実は Cantor の集合論からも構築する事が可能です. その為に次の集合の列を考えます.

$$\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}, \dots,$$

この列は単純に左側の集合を元として持つ集合と空集合の和集合を続々と生成する事で得られた列です. この集合の列の濃度 (基数) には自然数との次の関係があります.

$$\begin{aligned} 0 &\Leftrightarrow \emptyset \\ 1 &\Leftrightarrow \{\emptyset\} \\ 2 &\Leftrightarrow \{\emptyset, \{\emptyset\}\} \\ 3 &\Leftrightarrow \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ &\dots \end{aligned}$$

そこで, $\emptyset$ を 0 とします. 次の $\{\emptyset\}$ は $0 = \emptyset$ とした為に, $\{0\}$ となります. そこで $\{0\}$ を 1 とします. 次の, $\{\emptyset, \{\emptyset\}\}$ は $\{0, 1\}$ になり, これを 2 とします. 以降, この作業を繰返すと, 上記の集合は $\{1, 2, \dots, n-1\}$ の形の集合の列になり, $\{1, 2, \dots, n-1\}$ が自然数の n に対応します.

又, この集合の列を構成する任意の集合には, その構成方法から包含関係 $\subset$ による順序が自然に入ります. このことから, これらの集合で構成された集合 C は順序 $\subset$ による全順序集合になります. 更に, この集合の列から任意の部分列を取出した場合, 包含関係による最小元が, 集合の構成方法から必ず存在します. この性質を整列性と呼び, 整列性を持つ集合の事を整列集合と呼びます.

一般的に, 全順序集合 $(A, >_A)$ と $(B, >_B)$ 与えられた時, 写像 $f: B \rightarrow A$ が一対一の写像で, $a, b \in B$ に対し, $a >_B b$ であれば, $f(a) >_A f(b)$ を満す場合に $(B, >_B)$ は $(A, >_A)$ と同じ順序型を持つと言います. 又, 同型写像 f の事を順序同型と呼びます. 更に, 集合 $(A, >_A)$ が整列集合の場合を特に順序数と呼びます.

さて, 集合 C の元と自然数を集合 C の元の濃度を返す函数 card で対応関係を付けましたが, この函数は集合 C の元 a, b に対し, $a \subset b$ であれば, $\text{card}(a) < \text{card}(b)$ が成立します. 即ち, この函数 card は順序を保つ函数です. このことから, 自然数 $\mathbb{N}$ は整列集合 C と同じ順序型を持つ為に, 順序数としての特徴も持つ事が判ります.

この集合の定義方法を使うと、個数を表現する基数と順番を表現する順序数が厳密に区別されていますが、一寸判り難いかもしれないので、LISP を使って、この考え方を実際に試してみましょう。

まず、空集合 $\emptyset$ は空リスト "()" を用いて表現します。すると、上記の集合を作る操作は、与えられたリストを成分に持つ単リストを生成して、その単リストを与えられたリストに加える操作になります。

ここで、この操作を行う関数を s と名付けましょう。この関数 s は `(defun s (x) (append x (list x)))` で定義出来ませんが、より LISP らしく以下の関数で定義します。

—— 関数 s の LISP による表現 ——

```
(defun s (x) (cons x x))
```

さて、この関数の実行例を示しておきます。先程の集合と順序が逆になっていますが、同じものが得られていますね。

```
[2]> (s '())
(NIL)
[3]> (s (s '()))
((NIL) NIL)
[4]> (s (s (s '())))
(((NIL) NIL) (NIL) NIL)
[5]> (s (s (s (s '()))))
((((NIL) NIL) (NIL) NIL) ((NIL) NIL) (NIL) NIL)
[6]> (s (s (s (s (s '())))))
((((((NIL) NIL) (NIL) NIL) ((NIL) NIL) (NIL) NIL) (((NIL) NIL) (NIL) NIL)
  ((NIL) NIL) (NIL) NIL)
```

LISP ではリストの長さは `length` 関数で求められます。このリストの長さが集合の基数に対応します。実際に確認してみましょう。

```
[7]> (length '())
0
[8]> (length (s '()))
1
[9]> (length (s (s '())))
2
[10]> (length (s (s (s '()))))
```


3

[11]> (length (s (s (s (s '())))))

4

[12]> (length (s (s (s (s (s '()))))))

5

この様に、基数と順序数が対応しています。猶、この様にして構成した集合の基数が有限の場合、有限順序数と呼びます。

この方法の面白い事は、より大きな数を続々作る事が出来る事です。そこで、この作業を延々と続けて得られる $\{\emptyset, \{\emptyset\}, \{\{\emptyset, \{\emptyset\}\}, \dots\}$ の事を超限順序数と呼び、 ω と記述します。この ω の濃度は $\aleph_0$ になります。それから、この ω に対して $\omega + 1$ を $\{\emptyset, \{\omega\}\}$ で定義します。後は、この操作を延々と続けて行く事で以下の順序数の列が得られます。

$$1, 2, \dots, \omega, \omega + 1, \omega + 2, \dots, 2\omega, \dots, 3\omega, \dots, \omega^\omega, \dots, \omega^{\omega^\omega}, \dots, \omega^{\omega^{\omega^\omega}}, \dots$$

この様に、Cantor の集合論は集合を際限無く生成する事も出来るのです⁴。ところが、この際限も無く集合を生成する事の出来る能力から様々な問題も出て来たのです。

4.11.5 数学の基礎を巡る論争

最初に昔からの有名な逆理を一つ：

クレタ人の逆理

クレタ人はうそつきである。

これを当時のエジプト人やギリシャ人が言ったのであれば問題は無いのですが、Epimenides という名前のクレタ人⁵が言った為に有名になった一言です。この主張が本当の事であれば、これを主張したクレタ人がうそつきでなければならないので、この主張が嘘でなければならないくなります。逆に、このクレタ人がうそつきではないとすれば、彼を含めたクレタ人がうそつきであると彼は言っているので、当然、彼はうそつきでなければなりません。何ともややこしいですね

この逆理に似たのものに「Russell の逆理」⁶があります。この逆理を判り易く Russell 自身が言い換えたものが床屋の逆理で知られています：

⁴色不異空、空不異色、色即是空、空即是色、受想行色、亦不如是。

実体は空に他ならず、空は実体に他ならない。実体は即ち空であり、空は即ち実体であり、受、想、行、識もまた、これと同様である（般若心経より）

⁵紀元前 6 世紀頃のクレタのクノッソスの哲学者だそうです

⁶1901 年の春に Russell が見付けていますが、Peano の助手であった Cesare Burali-Forti も 1897 年に発見しています。

床屋の逆理

とある村には床屋が一軒だけあります。その床屋の主人は日頃、自分で髭を剃らない村人の髭しか剃らないと言っています。では、その床屋の主人の髭は誰が剃るのでしょうか？

先ず、この床屋の主張が正しいとしましょう。ここで床屋が自分の髭を剃れば、床屋は自分で髭を剃る人になってしまうので、床屋の主張からは自分の髭を剃る訳にはいきません。だからといって、彼の髭を剃らなければ、自分で髭を剃らない人になってしまうので、彼の主張から、自分の髭を剃らなければならない事になります。

この Russell の逆理は本質的に $\{x|x \notin x\}$ といった形の集合を考えた時に生じる問題です。要するに自分自身が属さないものの集合という摩訶不思議な代物です。実際、 $v = \{x|x \notin x\}$ と置いて、 $v \in v$ とすると、集合 v の定義から $v \notin v$ でなければなりません。ところが、 $v \notin v$ であれば集合 v の定義から $v \in v$ でなければなりません。この様にお手上げな状態に陥ってしまいます。この集合は、「めろめろな猫」の様に不明瞭な命題による定義ではなく、「 $x \notin x$ 」の様な「ちゃんとした論理式」⁷で定義されている事が非常に大きな問題になります。

次に、Cantor が見付けた逆理 (Cantor の逆理) を解説しましょう。これも非常に厄介です。

一般的に基数 $\text{card}(S)$ はその集合 S の冪集合 $\mathfrak{P}(S)$ の基数 $\text{card}(\mathfrak{P}(S))$ よりも小になります。では M を全ての集合の集合としましょう。そして、集合 M の冪集合 $\mathfrak{P}(M)$ の基数 $\text{card}(\mathfrak{P}(M))$ を考えます。すると、 $\text{card}(M) < \text{card}(\mathfrak{P}(M))$ となる筈ですが、ここで全ての集合を元とする集合 M を考えているので、 $\mathfrak{P}(M) \in M$ が成立します。従って、 $\text{card}(M) \geq \text{card}(\mathfrak{P}(M))$ となるので矛盾が生じます。

この様な逆理を分析したフランスの数学者 Poincaré は非述語的 (non-predicative) な集合の定義をした場合に逆理が生じていると述べています。例えば、「偶数の集合」や「身長 170cm 以下の人の集合」といった集合の定義では、自然数の集合や人間の集合といった集合の概念全体に触れる事無しに集合がきちんと定義が出来ます。これらの例の様な集合の定義の事を述語的と呼びますが、上記の逆理に出てくる集合はどれも、その集合の概念に直接触れなければ定義出来ない集合です。この様な集合の定義の事を非述語的な定義と呼ぶのです。

Cantor が創った頃の集合論は今では素朴集合論と呼ばれていますが、素朴集合論では集合の定義自体もあやふやな点があり、Russell の逆理や Cantor の逆理に対して全く無防備でした。

猶、19 世紀の数学では数学の厳密化と数学の算術化を進めています。例えば、函数の連

⁷Russell が後述の「悪循環原理」 (§4.12.4) で問題視する迄の事です

続性にしても, $\varepsilon - \delta$ 論法が出る迄は曖昧でした. 実数に関しても, Dedekind の切断によって明確化が為されています. ところが, 数学の算術化を進めて行く上で土台となる自然数の概念自体は, 実の所, 非常に曖昧なままでした. 1884 年に出版された Frege の「算術の基礎」[30] は, 論理から算術を導出する事を目指したのですが, そこでは当時の数学の哲学で主流であった, 心理主義, (Hilbert 以前の粗野な) 形式主義, そして経験主義等による数の考え方を批判しています. そして, Frege と同時期に Peano は自然数の公理化を行っています. 但し, こちらは 0 や 1 に関して Frege の様に踏み込んだものではありません. 論理学から数学自体を導出して見せたのが, Russell の「数学の原理」(The Principles of Mathematics, 1903, [53]) ですが, ここでは極端な实在論に基づいています ([2], 第三章を参照).

ところで, Cantor の集合論から導出された逆理は超限数論を用いたものであった為に, 集合論に批判的な人達にとっては, 集合論の問題として捉えられる事でしょう. しかし, Russell の逆理には超限数論的なものは何も無く, 数学の根幹に関わる逆理となっています. そして, 数学の厳密化と抽象化を進める為には様々な逆理とも関わらなくてはなりません. この数学の厳密化の方向は Russell に代表される論理主義, Brouwer の直観主義, Hilbert の形式主義といった主要な三つの立場で代表される様になります. ここで各立場の概要を説明しておきましょう.

論理主義

論理主義の立場は, 論理学から数学を導出しようとしたもので, これは Boole が論理学の代数化を図った事とは丁度逆の方向になります. この論理主義では, 数学を論理学の一部と捉え, 少数の公理で構成された公理系によって数学を再構築するものです.

この論理主義は Boole, Frege や Peano の仕事に大きな源を持っています. Peano は現在の数理論理学で用いられている表記の大本を作った人で, 前述の様に算術の公理化を行っています. Frege は Peano と異なる独自の二次元的な表記の「概念記法」([29], [31], [51], [20], 及び, §4.12.3 を参照) で変項・函数による命題の表記, 量化記号の導入を行い, Aristotle から続く論理学を一新しています. 更に, Frege は「算術の基礎」[30] や「算術の基本法則」[31] で論理学から算術を導出しようとしています. これは Frege によると, 算術はインドに起源を持つ事から曖昧な面が残っており, その為に Euclid 幾何学をモデルに算術を再構築しようとしたものです. この点では, Aristotle の方針に従って, 幾何学を構築した Euclid に倣ったものと言えるでしょう. この方針は, 「算術の基本法則」で頂点に達します. ここで, 「概念の外延」, 即ち, クラスを根底に自然数を構築し, Cantor の $\aleph_0$ に相当する ∞ を定義しています. ところが「外延(クラス)」という「集合論」的なものを導入した為, 最初から厄介な問題 (Russell の

逆理という時限爆弾)を包含する事となります. この「算術の基本法則」は出版社に出版を渋られた為に、I 巻、II 巻と分けて出版していますが、II 巻の出版直前に、Russell からの手紙で、彼の基本法則の公理系から「Russell の逆理」が導出可能な事を報らされます⁸. Frege はこの逆理に最終的に対処出来ず、晩年には論理学から数学を導出する事を諦めていた様です [20].

この Frege と入れ替わる様に Russell が論理主義を推し進めます. 「数学の原理」(The principles of Mathematics, 1903) では、Frege と独立して Frege と同様の自然数の定式化を行うだけではなく、「数学」を論理学から導出しています. 猶、Appendix で Frege の紹介も行っていますが、Russell は「数学の原理」の著作中に「Russell の逆理」を発見します. そして、その対処方法として分岐型理論を論文「The Mathematical logic as based on the theory of types」(1908, [51] 参照) で導入し、大著「数学原論」(Principia Mathematica, Vol.1 1910, Vol.2 1912, Vol.3 1913) で一つの体系として完成させました. Russell は「論理学は数学の青年時代であり、数学は論理学の壮年時代である」と語っている様に、数学を論理学の一部と見做す立場です.

Russell の論理主義では自分自身に言及しなければ定義が出来ない非述語的な命題に対しては「悪循環原理」や「型」⁹によって体系から排除していますが、「悪循環原理」によって幾つかの解析学上の結果が無効になり、その上、「型」で数学的帰納法の適用も使えなくなるという難点が新たに出て来ます. そこで、Russell は還元可能性公理 (Axiom of Reducibility [51] 参照) を導入しました. この還元可能性公理は、「任意の階の命題関数には、それと同値な述語関数が存在する」というものです¹⁰. この還元可能性公理を用いる事で帰納法の適用が可能になりますが、直観的に判り難い性質を持つ事、更に、還元可能性公理は述語の階差を無くしてしまう性質を持つ為、型をわざわざ導入した必要性が無いのではないかと非難されています. 又、還元可能性公理の導入によって、論理主義は実質的に公理的集合論とも言える側面を持つ事になり、Russell もこの還元公理には満足していませんでした. 更に、「数学は最早論理の上ではなく、論理主義者達の楽園の上に建設されているのである」(H. Weyl) とさえも皮肉られています.

猶、この論理主義から論理的推論の形式化を Hilbert は取り入れています ([26], pp.20).

⁸論理主義に懐疑的だった Poincaré は Russell の逆理に対して「論理主義はついに実りの無いものではない事を実証した. 何故なら、それは矛盾という結実を得たからだ」と喜んだとか ([43] 参照)

⁹型を用いた数学基礎論の本としては、[34] が挙げられます.

¹⁰Principia Mathematica の表記を用いると、 $(\exists \varphi). \psi x. \equiv_x . \varphi!x.$ と記述されます

直観主義

直観主義は数学的知識の基礎を「直観」に置くものです。代表的な人物は、Brouwer ですが、古くは Kronecker や Poincaré もこの考えに繋がります。Brouwer は、この観点から数学的知識の基礎を「論理」に置く論理主義や、数学の公理化で得られた超数学を研究対象とする形式主義に反対しています [7]。

直観主義では、厳密さを追求したが為に、数学は非常に窮屈なものになり、重要な数学の原理の幾つかが全く無効なものとなります。Hilbert が Brouwer の直観主義に反発した大きな点の一つに、Brouwer の排中律 (law of excluded middle) ¹¹ への攻撃が挙げられます。ここで排中律は、命題 P が真であるか P が偽であるか何れかが成り立つというもので、Aristotle の形而上学にもあります。この排中律を利用した証明の手法が背理法 (帰謬法) です。背理法を適用した証明では、命題 P が成立しない場合に矛盾が生じる事を示す事で、命題 P が成立する事を証明する事になります。即ち、命題 P かその否定 $\neg P$ の何れかが成立する事から、命題 P が偽ならば命題 $\neg P$ が真となる事を利用した証明方法です。

この証明法は非常に古くからあり、Euclid の幾何学原論には、素数が無限個存在する事の証明も背理法を用いた有名な証明方法があります。この証明の手順を簡単に説明しておきましょう。

先ず最初に証明すべき命題の否定「素数が無限個存在しない」を仮定します。無限個存在しなければ、「素数の最大値 M が存在する」事になりますね。さて、今度は最大の素数 M 以下の素数 $a, b, c, \dots, M$ の積に 1 を加えた数 $a \cdot b \cdot c \cdots M + 1$ を作ります。この数は M 以下の素数で割切る事が出来ません、従って、この数は最大の素数 M よりも大きな素数になるので、 M が最大の素数である事に矛盾します。この事から最初の仮定が否定されるので、その仮定の否定の命題である「素数が無限個存在する」が証明されたと主張するのです。

さて、Brouwer によると、対象が有限個であれば、命題が成立するかどうかを一つ一つ検証する事が可能ですが、相手が無限個になると最早、全ての対象に対して命題が成立するかどうかを検証する事が可能であるとは限らない事を挙げています。例えば、Brouwer は「円周率で 0123456789 と連続して出現する個所があるか」¹² という命題を挙げています。この場合、その箇所を計算して見付けるか、その様な数列が出現しない事を厳密に証明する必要がある、この命題が成立するか間違っているか何れかであるとは簡単に片付けられないのです。この様に、排中律を検証が十分に行えない無限集合に対して制限無しに利用する事を問題視しています。

¹¹ 中国を排除する法律ではありません。何故か google で出ないんですね。まあ、変な意図を持って機械的に情報の取捨選択をやっているのかもしれない。

¹² 近年の π の計算によると実際にその個所が存在するそうです

ここで、もし排中律を排除した場合、排中律を前提とした背理法が無効になるので、背理法を用いた証明は効力を失い、別の証明方法で置換えなければならない事になります。もし、それが出来なければ直観主義に立脚した数学では使えないのです。この直観主義の有名な犠牲者は、有界な実数の部分集合は上界を持つという Weierstrass の定理です。その為、Hilbert は排中律の攻撃を「数学から排中律を奪う事は天文学者から望遠鏡を、ボクサーから拳を奪う様なものだ」と大いに反発しています。

この様に、直観主義は厳密であるものの、その生産性の低さもあって数学の主流にはなりませんでしたが。但し、排中律への疑問等と非常に重要な主張をしており、それらは部分的に形式主義にも取り込まれています。更に、この考えは、数理論理学の直観主義数理論理学等に引き継がれています。

形式主義

Hilbert の形式主義は、数学を徹底して公理化する事によって数学自体を形式化し、有限回の機械的な処理で問題を解く (有限の立場) という考えがあります。形式主義の数学は生産性が非常に高いものであり、直観主義の様に数学を狭めるものでもなかった為に、主流となりました。現在の数学もこの形式主義の延長線上にあると言っても良いでしょう。

さて、この形式主義を推し進める方針として、Hilbert 計画と呼ばれるものがありました。次に、この Hilbert 計画について軽く触れておきましょう。

4.11.6 Hilbert 計画

Hilbert の構想では、数学は従来の素朴な非形式的数学、公理を使って形式化された数学と超数学の三種類で構成されます。ここで形式的数学は素朴な従来の数学に公理等を導入する事で形式化して得られるもので、逆に、非形式的数学は形式的数学の解釈で得られるものです。又、超数学は形式的数学を対象としたもので、その妥当性 (無矛盾性) を考察するものとなり、この超数学は所謂、有限の立場を堅持したものになります。超数学や形式的数学では、数理論理学の論理演算の様に最早命題の内容は問わずに、機械的な処理が可能となります。この事は、問題の本質を理解していない石頭の計算機でも、手順をきちんと与えさえすれば処理が可能である事を意味します。これはタイプライターを猿に与えて、猿が打ち出す出鱈目な文字の羅列の中から、やがて文学的作品が得られるかもしれないという確率的な話よりも、より現実的な話です。この考えが計算機の基礎にも繋がっているのです。

実際,Hilbert は形式化を推し進める事で, 数学を確固たる基盤の上に載せ, 数学自体を機械的な計算処理で済ます事を目的としていました. これが Hilbert 計画と呼ばれるもので, 具体的には以下の三段階を実施する研究プロジェクトです. この歴史的経緯とその詳細は林 [18] に詳しく出ています.

— Hilbert 計画 —

1. 現実の数学を形式化し, 数学の実体ではなく形式化したものを数学の実体と見做す
2. 形式系の無矛盾性を示す. これによって数学の絶対的な安全性が示される.
3. 形式系の完全性. 即ち, 任意の命題が決定可能である事を示す. これによって数学の任意の問題は常に解決が出来る事になる.

この計画が完了すれば数学は安全であると同時に完全なものとなる筈です. それに加え, 機械的な処理も安心して行える事になるのです. この Hilbert 計画の事は同時代の高木の著書 ([19],pp.269) で「ロボット式数学」として簡単に触れており, 「ロボットが出来て円滑 (無矛盾) に動くであろうかが興味のある問題だ」と書かれています. さて, 次の節では, この Hilbert の構想を眺める為, 命題や述語の話をしてしましょう.

4.12 命題と述語

4.12.1 小史

現在の論理学には二つの流れがあり、一つはギリシャに源を発する西洋論理学、もう一つはインドのインド論理学です。これらの他に戦国時代の中国にも、墨家の論理学の萌芽がありましたが、残念ながら、こちらは漢代には途絶え、「墨子」[32]の経編等に断片的に残っているものの、その全体像を窺うには不十分な状態です¹³。

ここで日本にはインドの論理学(因明)が仏教と共に伝来しており、奈良の南都仏教で研究されていたそうです。

そこで最初に古因明による推論の例を挙げておきます：

古因明の例

主張(宗) あの山は火を有する。
理由(因) 煙を有するが故に。
実例(喩) 煙を有するものは火を有する。竈の様に。
適合(合) 竈の様に、あの山もそうである。
結論(結) あの山は火を有する。

この様に古因明では宗、因、喩、合、結の五段(五分作法)で構成されており、宗が命題、因が命題が成り立つ理由、喩が具体的な事例、合が因との適合を示し、結が結論となります。この古因明を更に洗練したものが新因明と呼ばれるもので、この新因明では宗、因、喩、合、結が整理されて、宗、因、喩で構成される三枝作法になっています。三枝作法と後述のAristotleの三段論法を比較すると、宗が結論、因が小前提、喩が大前提に対応しそうです。但し、インドの論理学は真なる結論を導き出す論証の学([38]参照)であり、喩の存在から、帰納的な傾向が強い論証を行っています¹⁴。

¹³「墨子」は、論理学、幾何学、技術的な断片を含んでいる為に清朝末の西洋科学の導入で大きく取り上げられています。因に、中国・朝鮮の様な伝統的社会では、新規なものを導入するにあたって、自らの古典で似たものを「再発見」して導入するという手続を踏む傾向があります。その為、実質的な革新運動が復古運動の衣を被ったりします。古くは宋明理学、最近では清朝末期の戊戌の変法の指導者である康有為は董仲舒の公羊学を思想的な柱としています。そういえば、明治維新も最初は王政復古でありながら、徹底した文化革命であり、北一輝の昭和維新も表向きは復古的でありながら、実体は象徴的な天皇を頂く国家社会主義とハイパーモダンなものです。もっとも、2.26事件の青年将校達がどれだけ理解していたかは別問題ですが、これと比べると、大東亜戦争開戦後の座談会「近代の超克」[8]は名前の仰々しさの割に、中身は第一次世界大戦中の同盟国側のプロパガンダ(商人根性 v.s. 軍国主義精神)の焼き直し、転向者の「真面目に教えてくれなかった大人が悪いからこうなった」式の反省文、そして、周回遅れのドイツ浪漫主義の詰合せと羊頭狗肉の観があります。この毒にあたった場合、解毒剤として夏目漱石の「我輩は猫である」(日露戦争中の著作!)を私は強く薦めておきます。

¹⁴桂[9]では帰納的な推論であると述べていますが、異論も多い様です。又、インドの論理学は弁論と討論の術としての側面もあります。「屍鬼二十五話」[17]では「判っていて質問に答えないと頭が破裂するよ」と屍鬼が主人公を脅しますが、討論も論敵の質問に答えられなければ「頭が破裂する」程です。

一方のギリシャでは、Aristotle は名辞論理によって論理学を整理しましたが、メガラ派、ストア派といった哲学諸学派の論理学にも命題論理の萌芽がありました。但し、これらは忘却されてしまいます。

ローマ帝国の最盛期になると、ストア派の哲学、新プラトン主義の哲学、グノーシス等が盛んになります¹⁵。ここで初期のキリスト教はユダヤ教や、その他の宗教に対する理論武装の為、Plotinos 等の新プラトン主義の影響を受けて、その神学を構成して行きます¹⁶。

ローマ帝国の衰退以降、ヘレニズム文明を伝承したのがイスラーム世界です。イスラームにも独特の論理学があります。これは必要によって生じたものです。先ず、イスラーム法では、コーランとハディース（預言者の言行録）を基にしています。ところで、コーランは体系的な経典では決して無く、それどころか、寧ろ矛盾が多い事で知られています。それに加え、イスラーム世界の拡大や文化の発達等の要因で、コーランやハディースでは言及されていない物に対し、それが宗教的に禁止されたもの、則ち、ハラーム（harām）であるかどうかを判断しなければならない事が数多く出て来ました。

例えば、葡萄酒は預言者によって宗教的に禁止されていますが、他の酒、例えば、ラム酒には言及がありません。この為、キヤース（qyās）と呼ばれる三段論法が用いられています。

キヤースの例 [3]

- （大前提） 発酵させて造り、人を酩酊させる飲料は harām である。
- （小前提） ラム酒は発酵させて造り、人を酩酊させる飲料である。
- （結論） 故に、ラム酒は harām である。

猶、このキヤースの大前提は何でも良いものではありません。この大前提を求める為には帰納法を用いなければならず、更に、一般性を持った根拠があるものでなければ大前提としては使えないのです。この点は、イスラム神学の一派で（超）合理主義的思想のムアタズィラ（Mu'tahzilah）派が三段論法を駆使して、ともすれば異端的な結論を出していたそうですが、その歯止として必要になるでしょうか。

因に、イスラム思想では真理をハック（Ḥaqq）と呼びます。そして、哲学者キンディー以降、神の事もハックとも呼びます。そんな訳で、日本語で「ハックする」と言えば真

¹⁵ この時期の風刺作家 Lucianos の短編「逃亡者」[44] に檻を纏ってストア派や犬儒派の哲学者の真似をし、俗物の金持に集る輩の話があります。又、五賢帝の最後の皇帝 Marcus Aurelius Antoninus はストア派の哲学者としても知られています。

¹⁶ 新プラトン主義の絶対的な一者からの万物の流出を説く流出論による解釈があつて初めて、キリスト教やイスラーム教といったセム的一神教への受容が容易になったと言われています。但し、この事によって、Aristotle の思想の伝播も新プラトン主義的な夾雑物を多分に含む事になります。猶、後に異端とされますが、グノーシスの影響を受けたものも多くあります。

理を探求する事、「ハッカー」とは真理の探求者であると駄洒落に出来ますね¹⁷.

閑話休題、ヘレニズム文明のイスラームへの導入では、シリアのキリスト教徒が大いに活躍しています。先ず、シリアのキリスト教徒の医者が医学書や自然科学の文献をアラビア語に翻訳し、やがて、アッバース朝¹⁸では非常に合理主義的なイスラム神学の一派であるムアタズィラ (Mu'tahzilah) が公認され、学問好きのカリフ (教皇) からも奨励されていた事もあって、組織的にギリシャ哲学、自然科学、医学等の文献をアラビア語へと精密に翻訳しています¹⁹.

このムアタズィラは自らを「正義と神の唯一性の提唱者」と称し、その合理性は徹底したものであり、この徹底した合理主義は Ghazali に非難され、後の「正統派」によってその著作が根絶させられています。

例えば、次の特徴があるそうです ([3], pp.54-65. 参照):

- 予定論を認めない.
- ムハマンドによる執り成しを認めない.
- 神を人格化して表現する事を認めない等

予定論は神によって人間の運命は完全に定められたものであると言うものです。これは正義の神が人間に不正義を行わせる筈が無く、人間の行為は全て人間に帰着するものであり、この事から人間は神と並んで第二の創造主となると言うものです。猶、善悪は理性に照して判断されるものである為、神ですら理性の規準に従わなければならないという程です。猶、予定論も受け身であれば、何かと問題が出ますが、これを積極的に捉えたのがキリスト教のカルバン派等のプロテスタントで、彼等が商業等で社会的な成功を収める原動力の一つであったと言われています。

執り成しは、イスラム教徒であれば最後の審判でムハマンドが信徒の為に神に執り成しを行って罪の軽減が計られるというものです。人間が第二の創造主であるとすれば、全ては自分に責任があり、悪行故に地獄の業火で焼かれるのは当然という事になるでしょう。

次の神の人格表現の否定は、コーランに描かれた神の人間的表現を比喻として捉えるもので、知識や理性の神は哲学者にとっては望ましい事としても、より具体的なものを

¹⁷だからと言って、過労で「即身成仏」、或いは「神と合一」する事は後述の神秘思想とは無関係です。

¹⁸千夜一夜物語の舞台となる王朝です。

¹⁹猶、学問的な事以外に、ローマの都市文明を受け継ぎ、発展させたのもイスラーム世界です。例えば、ローマの公衆浴場は、イスラーム世界でもハンマーム/ハمامとして引き継がれ、江戸の浴場の様に、都市に居住する市民にとって無くてはならないものとなっています。一方で、ヨーロッパでは、中世以降、消えていますが、19世紀に入って導入されています。その理由に富国強兵と衛生 [11] といった見方もあります。

望む一般の信者にとっては非常に迷惑な事でしょう。この考えを推し進めると、「哲学こそが全てであり、宗教は一般大衆向けの幼稚な哲学である」となります²⁰。

さて、イスラーム世界の哲学者で最も有名なのが、Ibn Sina と Ibn Rushd です。Sina はヨーロッパでは Avicenna と呼ばれ、彼がイスラーム哲学の頂点とも言われています。彼は哲学者であるだけでなく、医学者としても著名でした。因に、同時代のイスラーム世界の学者から死後、「自らの哲学で自らの魂を救えず、自らの医術で自らの肉体を救えず」と評されています。

Rushd はヨーロッパでは Averroes と呼ばれています。彼は Aristotle 注釈で著名であり、Aristotle の原典に戻れと主張した様に、従来の Aristotle 解釈が、新プラトン主義に汚染されたものである事を知っており、その様な夾雑物の排除を目指していました。但し、新プラトン主義的なものの排除は困難であり、西洋のスコラ哲学にも、この痕跡は残ったままで、この影響は 17 世紀迄及んでいます²¹。

一方のヨーロッパでは十字軍に伴って交易が活発になった事もあり、イスラーム哲学を経由する事で Aristotle 等の古代ギリシャ哲学が再導入されています。その結果が中世のスコラ哲学として結実します。その事もあって西洋論理学では、Aristotle に端を発する論理学が長く主流となります。

4.12.2 命題と述語について

まず、論理学で扱う命題とは「人間は死すべき生き物である」、「ソクラテスは人間である」、「 $4 = 2^2$ 」や「 $1 > 10$ 」の様に、ある事柄について述べた文の事です。そして、Aristotle の論理学では、この命題を主語 (subject)、述語 (predicate) の二項と、これらの二項を繋ぐ繫辞 (copula) 分けて考え、論証の形式的な正しさに重点が置かれていました。

ここで命題を「ソクラテスは人間である」とすると、主語が「ソクラテス」、述語が「人間」、繫辞が主語と述語を繋ぐ、「…は…である」になります。因に、命題の解釈として、内包的と外延的の二通りの解釈があります。これは「全ての鳥は黒い鳥である」という命題を解釈する場合、「鳥」の属性として「黒い鳥」という概念が付属すると解釈するのが内包的解釈、「鳥」の外延 (集合) が「黒い鳥」という外延に含まれるという解釈が外延的解釈になります。ここで Aristotle は前者の内包的解釈で、その事も

²⁰徹底した合理主義のムアタズィラがある一方で、神との合一 (我は汝、汝は我) を目指すスーフィズムに代表される神秘主義もあります。このスーフィズムに関しては仏教の影響も指摘されています。又、年寄には回教徒の事を「フィイ」と呼んでいる人もいますが、このスーフィに由来します。そして、ダルビッシュはスーフィの修行僧 (乞食僧) に由来します。

²¹Dante の「神曲」では、Sina, Rushd, 更には、Plato, Aristotle 達の未洗礼の偉人が送られる Limbo (リムボ) に居ます。この事からも、著名さの度合いが判るかと思います。

あって、伝統的西洋哲学では内包的解釈が主流でした。現在、集合の定義で述語を用いて表記する方法が内包的定義、要素を列挙する方法が外延的定義と呼ばれます。

さて、命題の判断というものがあります。これは命題を構成する主語と述語の持つ概念が一致するかどうかを断定する事です。この判断には「肯定判断」、「否定判断」、「全称判断」と「特称判断」といった代表的な判断があります。

命題の判断

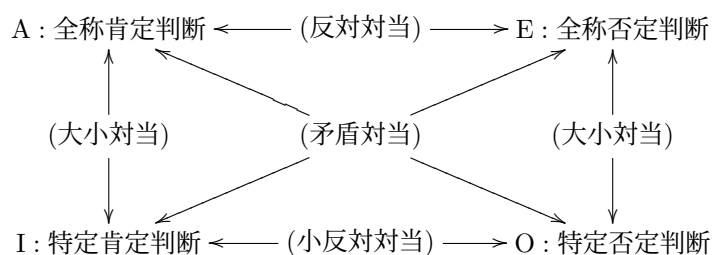
- 肯定判断： 命題「 A は B である」の様に、あるもの(A)に対してある事(B)を認める判断
- 否定判断： 命題「 A は B ではない」の様に、あるもの(A)に対してある事(B)を否定する判断
- 全称判断： 命題「全ての A は B である」の様な判断
- 特称判断： 命題「全ての A は B である」の様な判断

そして、肯定的判断、否定的判断に対して全称判断、特称判断を組合せる事で、以下の、A: 全称肯定判断 (Universal Affirmative), E: 全称否定判断 (Universal Negative), I: 特称肯定判断 (Particular Affirmative), O: 特称否定判断 (Particular Negative) が得られます。

A, E, I, O

- (A) 全称肯定判断： 「すべての A は B である」
- (E) 全称否定判断： 「いかなる A も B ではない」
- (I) 特称肯定判断： 「ある A は B である」
- (O) 特称否定判断： 「ある A は B ではない」

猶、A, E, I, O はラテン語の動詞 AFFIRMO(肯定する) と NEGO(否定する) に由来します。更に、これらの AEIO に対しては、以下の対当関係表 (SQUARE) が得られます。



この対当関係から、命題の真偽について次の推論が可能です。

- 大小対当 (A と I, E と O)

1. 全称が真の場合, 必ず特称も真
 2. 特称が偽の場合, 必ず全称も偽
 3. 全称が偽の場合, 特称の真偽は不定
 4. 特称が真の場合, 全称の真偽は不定
- 反対対当 (A と E)
 5. 何れか一方が真の場合, 必ず他方は偽
 6. 何れか一方が偽の場合, 他方の真偽は不定
 - 小反対対当 (I と O)
 7. 何れか一方が偽の場合, 他方は必ず偽
 8. 何れか一方が真の場合, 他方の真偽は不定
 - 矛盾対当 (A と O, E と I)
 9. 何れか一方が真の場合, 他方は必ず偽
 10. 何れか一方が偽の場合, 他方は必ず真

猶, 伝統的論理学には扱う命題には「存在含意 (external import)」が含まれています. これは, 伝統的論理学で命題を主語と述語と繫辞に分解しますが, ここで主語となる対象の存在を前提にして命題を扱っています. 例えば, 「A is B」であれば, 主語が「A」, 述語が「B」, 連辞が「is」になりますが, それに加え, 伝統的論理学では主語の「A」には, 「A exists」を含めて考えています. その為, 「あるドラゴンは火を吹く」といった命令を扱おうにも, ドラゴンの存在が肯定されない限り, この命題は扱えない為に真偽が不定になります. 更に, 本当に正しいかどうか判らない仮説を基に推論を行う事もままならない事になります.

この辺は, Aristotle が Plato の弟子でありながら, 師のイデア論に批判的で, 現実的であった事もある様です.

次に, 推論の形式で代表的なものとして三段論法 (Syllogism) があります. 三段論法は, 大前提, 小前提と結論の三つの命題から成る推論の規則です. 更に, 三段論法には, 定言三段論法, 仮言三段論法, 選言三段論法と両刀論法 (ディレンマ (Dilemma)) の四種類があります. これらの非常に安易な例を以下に示しておきましょう.

—— 定言三段論法の例 ——

大前提: 人間は死すべきものである
 小前提: ソクラテスは人間である
 結論: 故にソクラテスは死すべきものである

—— 仮言三段論法の例 ——

大前提: ソクラテスは人間である
 小前提: 人間は死すべきものである
 結論: 故にソクラテスは死すべきものである

—— 選言三段論法の例 ——

大前提: 禿であるかフサフサであるか
 小前提: フサフサではない
 結論: 故に禿である

—— 両刀論法の例 ——

大前提: 雨なら傘が要る, 且つ, 雪なら傘が要る
 小前提: 雨ないし雪だ
 結論: 傘が要る

最後に, Aristotle は排中律と矛盾律を「形而上学」([1], pp.120-152) で述べています. ここで排中律は命題 P あるいは, 命題 P の否定の何れかが成り立つというもので, 矛盾律は命題 P と, 命題 P の否定が同時に成立する事は無いというものです.

この Aristotle の影響は絶大で, 後に Kant が「Aristotle 以来進歩もなければ後退もない, いわば完成された学問」と (伝統的) 論理学の事を呼んだそうですが, 実際は, 伝統的論理学で 19 世紀末に至る迄, 満足な解決が出来なかった問題, 例えば, 多重量子化を含む推論もあったのです ([2] 参照).

Aristotle 以降で興味深い人は Leibniz です. 彼は最後の万能の天才と言える人で, 普遍言語と普遍記号を考えています. 先ず, 普遍言語は, 日常言語の曖昧さを防ぐ為に, 命題に自然数を対応させ, A かつ B をそれらの命題に対応する素数の積として表現する事を考えています²². 例えば, 命題 A を「動物である」, 命題 B を「理性的」であるとする, 「理性的な動物」である人間は「理性的, かつ, 動物」なので $2 \cdot 3 = 6$ で表

²²Gödel 数みたいです

現されます。同様の考察で Leibniz は猿が 10 であると述べ、このことから、人間と猿が異なると結論付けています。逆に、6 と 10 には共通因子として 2 がありますが、これは人間も猿も共に数 2 で表現される「動物」だからです²³。猶、命題と数値を結び付ける事は、ピタゴラス学派でもやっています。例えば、結婚を表現する数 5 は女を表現する数 2 と男を表現する数 3 との和というのですが、こちらは寧ろ秘術的なもので Leibniz と異なります。

更に、Leibniz は命題 A に対して $A + A = A$ (Leibniz の表記では $A + A \infty A$) と後述の Boole に 150 年も先立って述べています ([55], [4])。

Leibniz といえば現在では微分記号 dx や積分記号 $\int$ で知られていますが、Newton の表記 (微分は $\dot{f}$) と比較して明瞭である事は、彼が普遍言語やそれを表現し得る記号に関する考察があったから出来た事と言われています。因に、イギリスでは Newton 流の表記が長く用いられており、Leibniz 流儀の表記が導入されたのは、Herschel, Babbage, Peacock の解析協会 (Analtical Society)²⁴によるものですが、この Newton 流儀の微分積分を使い続けた事が 19 世紀初頭にイギリスが大陸側の数学と比較して大きく遅れる原因となったと言われています。

その他に、Leibniz は歯車式の計算機を作成したり、二進法による数値の表現を考案しています。その点では、現在の計算機のパイオニアとしての側面も持っています。特に、二進法に関しては、宗教的神秘主義めいた考えさえ持っていました。例えば、数字の 7 について、天地創造が 7 日で行われた事と、二進法で 7 が 111 と表記される事を三位一体と結び付けています ([27], pp.207)。その点では、ピタゴラスの輩とも似てなくもありませんね。

ところで、Leibniz は自らの学派を構成する事も無く、様々な思索を行った膨大なメモを残した為、彼の再評価は 19 世紀に末から始まります²⁵。

19 世紀に入ると、イギリスで集合を用いて論理式の定式化を行った De Morgan, 0 と 1 を使って定式化を行った Boole([46],[47]), Venn 図で知られた Venn が現われます。

先ず、De Morgan は論理式に集合を導入しています、その際に全体集合を導入しています。そして彼独自の記号を導入しますが、その記号を使って三段論法の A, E, I, O がどの様に表現されるかを見てみましょう [55]:

²³Swift の「ガリバー旅行記」にはラピュタ国の学者の言語の研究が紹介されていますが、これは当時の普遍言語に対する皮肉なのでしょう。更に、このラピュタ国では Newton も風刺されています。因に、計算機内部の番地の処理で、Intel の little-endian と Motrola の big-endian という名称は小人国の戦争の原因 (卵を細い方から先に食べるか、太い方から食べるか) で付けられたものです。

²⁴現在の Cambridge Philosophical Society。猶、学生の冗談で協会の目的は、‘大学の dot-世代への純正 d-主義原理’ (‘the principle of pure d-sim as opposed to the dot-age of the university’) の宣伝だと言われていたそうです。因に、Herschel は天文学者、Babbage は解析機関の Babbage です。

²⁵Voltaire の哲学的コント「カンディード」[6]では徹底して Leibniz の哲学 (楽観主義や予定調和、「この最善なる可能世界においては、あらゆる物事はみな最善である」) が茶化されています

De Morgan による AEIO の表記

- (A) 「全ての A は B である」 $\Leftrightarrow A \supset B$
 (E) 「全ての A は B ではない」 $\Leftrightarrow A \not\supset B$
 (I) 「ある A は B である」 $\Leftrightarrow A \cap B \neq \emptyset$
 (O) 「ある A は B ではない」 $\Leftrightarrow A \cap B = \emptyset$

De Morgan の記号では、否定の記号はありませんが、命題 A の否定を小文字を用いて a と表記します。そして、「 PQ の否定は q, p 」, 「 P, Q の否定は pq 」であると述べています。これが現在、De Morgan の法則と呼ばれているものです。この意味を現代の集合の記号を用いて書くと以下のものになります:

De Morgan の法則

1. $(A \cap B)^c = A^c \cup B^c$
2. $(A \cup B)^c = A^c \cap B^c$

この様に、(E) 全称肯定判断と (I) 特称否定判断は否定によって互いに移り会える事が判ります²⁶。

Boole は初めて論理式の代数的処理の体系を創った人です。Boole は論理式に対し、その論理式を満たす集合を考え、この集合に対して演算を行っています。この演算は集合の和 (+) と積 ($\times$), 補集合 ($-$) の三種類になります。又、論理式を X とする時、 X を満たすものを elective symbol と呼ぶ小文字の x で表現し、この小文字に対して演算を行います。そして、特別な記号として 1 と 0 を導入します。先ず、1 は全体集合を表現し、Boole の代数的処理で積の単位元としての働きをします。これに対して、0 はその様な命題が成立しない事を示し、和の単位元としての働きをします。猶、 $1 \neq 0$ は明確に述べていないものの、実際の計算では $1 \neq 0$ として処理をしています。

以下に基本的な論理式の基本的な代数化を示しておきましょう:

²⁶14 世紀には既に知られていた事だそうです (Willians of Ockham) [55].

Boole による命題の代数化

- $x1$, 即ち x は X を満す事
- $y1$, 即ち y は Y を満す事
- $xy1$ 即ち, xy は X を満し, 且つ, Y も満す事
- $1 - x$ は X でない事
- $x(1 - y)$ は X を満し, 且つ, Y を満さない事
- $(1 - x)(1 - y)$ は X を満たさず, 且つ, Y を満さない事

更に, $x + y$ を集合 X と集合 Y の和集合とし, この和と積に対しては, 以下の性質があると述べています:

Boole による演算子の性質

可換性:	$x + y = y + x$	$xy = yx$
結合律:	$(x + y) + z = x + (y + z)$	$(xy)z = x(yz)$
単位元:	$x + 0 = x$	$x1 = x$
分配律:	$x(y + z) = xy + xz$	
冪の性質:	$x^n = x$, 但し, n は 1 以上の正整数	

この様に, 和と積に対しては分配律が成立し, 積に対しては可換であると述べています. この二番目の可換性の説明では, Boole は同じ意味を持つ日常語の「善良で賢い男」と「賢くて善良な男」を挙げています. 即ち, 「善良である」という命題を X , 「賢い」という命題を Y とすると, 「善良で賢い」は xy となり, 「賢くて善良」は yx となりますね. そして, 意味を考えると, $xy = yx$ になる訳です.

又, $x^2 = x$ から, 式の変形によって, $x(1 - x) = 0$ が得られますね. この式は「 X 且つ X でない事はありません」と読めます. この事から Boole による体系が 1 を真, 0 を偽とする事が理解出来るでしょう.

さて, 定言三段論法で現われた以下の命題はどの様に表記されるでしょうか?

Boole による AEIO の表記

(A) 「全ての Y の元は X の元である」	$\Leftrightarrow$	$y = vx$
(E) 「全ての Y の元は X の元ではない」	$\Leftrightarrow$	$y = v(1 - x)$
(I) 「ある Y の元は X の元である」	$\Leftrightarrow$	$vy = v'x$
(O) 「ある Y の元は X の元ではない」	$\Leftrightarrow$	$vy = v'(1 - x)$

最初の全称肯定判断は、「 Y ならば X 」と読み替えられます。Boole の考えでは、ある時点で y が、ある x となる事を意味しており、その為に、 $Y \subset X$ の包含関係が生じます。この「ある x 」に対しては、集合記号 V を導入する事で vx と表現し、 $y = vx$ が求める代数式になります。この集合記号 V は命題を表現する集合 X や Y と同様の性質、 $v^2 = v$ や $v(1 - v) = 0$ を満すものです。この様に、正体不明な集合が突如出ているのが気持ち悪い事と、考え方に必然的に「時間」が入っています。但し、命題は真か偽に判断可能なものとして考えています。

二番目の全称否定判断は、「 X ではない」を $1 - x$ と表現し、全称肯定判断の定式化から、 $y = v(1 - x)$ となる事が判ります。三番目の特称肯定判断は、「ある Y 」が「ある X 」に一致すると言い換えられる為、 $vy = v'x$ と表現される事になります。最後の特称否定判断は「ある非 X の元である」と言い換える事で、 $vy = v'(1 - x)$ となる事が判ります。

この様に、命題の代数的な処理はある程度出来る訳ですが、この体系で三段論法が上手く表現出来るかと言えば、Frege が「ブールの論理計算と概念記法」[29] で非難している様に、代数式と日常語が入り交じった式になってしまいます。又、 $x^n = x$ や $x + \dots + x = x$ となる点、更には、 $0/0$ や $1/0$ といった表記が出る等と、通常の演算とは全く別の演算となるのが厄介な点です。

例えば、Boole の挙げた例で、ユダヤ教では清浄な動物を、「蹄が割れていて、反芻する動物」としています。ここで、 X を「清浄な動物」、 Y を「蹄が割れている動物」、 Z を「反芻する動物」とした場合、清浄な動物の定義は $x = yz$ と記述されます。ここで普通の式ならば、 $z = x/y$ とする事も可能ですね。ではこの $z = x/y$ はどのような意味を持つのでしょうか？

その他にも、Boole の演算では $1/0$ や $0/0$ といった計算も出て、随分と通常の代数的演算とは趣が異ったものになります。猶、この Boole による命題の代数的処理は利用価値の非常に高いもので、その為に、計算機でも活用されるものとなっています。

1878 年になると、Frege が「概念記法」にて述語 (函数) という概念を導入し、述語論理を彼独自の表記で構築しています。Frege は、従来の命題を主語と述語といった関係ではなく、寧ろ、その命題の判断に重きを置き、命題を函数と変項で表現したのです。例え

ば,「水素ガスは炭酸ガスよりも軽い」という命題を Frege は主語を「水素ガス」, 述語を「炭酸ガスよりも軽い」といった分析ではなく, “軽い (水素ガス, 炭酸ガス)” の様に命題を変項を持つ関数として表現したのです. 次の §4.12.3 で Frege による命題の形式化を眺めてみましょう.

4.12.3 Frege の概念記法

概念記法 (Begriffsschrift) の概要

ここでは Frege の概念記法 (Begriffsschrift) の解説をします. Frege の著作「概念記法」[29], 「算術の基礎」[30] と「算術の基本法則」[31] は非常に面白く, 論理主義を素材から造り上げ, 壮麗な大聖堂を構築する様な凄さがあります.

Frege の命題の表記は非常に独特のものです. 現在, 用いられている Peano 流儀の線的な論理式の表記方法とは異なり, 平面的な図式で命題を表現しています.

まず, 「命題 A がある」という事を $\text{---}A$ と表記します²⁷. この図式の --- は「概念記法」[29] では内容線, 「算術の基本法則」[31] では水平線, と呼ばれる複数の命題を繋ぐ構成要素です. 猶, 「算術の基本法則」では関数を構成する要素として扱われており, 例えば, 「 x 」や「 1 」といったものは表示と呼ばれるもので, 論理式ではありませんが, $\text{---}\xi$ で ξ の値が真の場合に真を返し, その他の場合は偽を返す関数として扱います. その為, 水平線 --- は原子や S 式から S 式を新たに構築する LISP の括弧“()”に似た作用素になっています.

次に, 「命題 A が判断出来る」という事を先程の表記に対し, 水平線線の左側に判断線と呼ぶ記号 $\vdash$ を導入して $\vdash\text{---}A$ と表記します.

この判断線は, 判断線の右側にある命題の真偽が判断出来るという事を示す為に用います. 即ち, Frege の表記では判断可能な命題と判断可能かどうか不明瞭な命題が分けて表記出来ています. 猶, 現在の数理論理学で論理式が証明可能である事を表す記号 $\vdash$ はここから来ています.

命題の否定は内容線の中程に短い縦棒を追加します. 例えば, 論理式 $\text{---}A$ の否定は $\text{---}\text{---}A$ と表記します. この表記で短い縦棒が命題の否定を表現し, 縦棒の右側の横棒 (---) が命題 A の内容線, 縦棒の左側の横棒 (---) が命題 A を否定した命題 $\text{---}\text{---}A$ の内容線となります. 猶, この表記から現在の論理学で用いられている否定 $\neg$ が出ています.

²⁷Frege の概念記法の表記に `begriff.sty`[86] を利用しています. TeXlib にも含まれていますが, <http://www.ctan.org/tex-archive/macros/latex/contrib/begriff/> から入手出来ます.

次に, 命題 A と命題 B が与えられた場合に,

1. A が肯定され, かつ, B が肯定される
2. A が肯定され, かつ, B が否定される
3. A が否定され, かつ, B が肯定される
4. A が否定され, かつ, B が否定される

といった四つの可能性があります, この三番目の事を排除し, 残りの三つ内の一つが生じるという判断を意味する表記として $\begin{array}{c} \text{---} B \\ | \\ A \end{array}$ を導入しています.

この図式の左側最初の横棒 (---) が命題全体の水平線, そして, 縦棒 (|) が条件線と呼び, 各命題の左にある横棒 (---) が各々の命題の水平線となります.

この図式は「 A ならば B 」に対応し, 現在の記号で記述すれば, $A \supset B$ 或いは $A \rightarrow B$ と表記されます. この $A \rightarrow B$ は $\neg A \vee B$ と同値になります. この様に, 論理学の「 A ならば B 」は日常で用いる「 A ならば B 」とは意味合いが少し異なる事に注意して下さい. 因に, 現在の数理論理学で用いられている論理式の表記は Peano の表記を基にしたものです.

さて, この Frege の表記を用いると, 「 A または B 」は $\begin{array}{c} \text{---} B \\ | \\ A \end{array}$, 「 A かつ B 」は $\begin{array}{c} \text{---} B \\ \text{---} A \end{array}$ と表記されます.

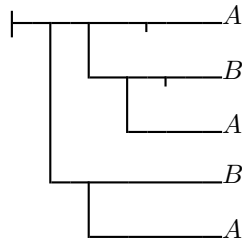
因に, 現在の論理学の表記では, 「 A または B 」は $A \vee B$, 「 A かつ B 」は $A \wedge B$ と表記されます.

ここで現在の表記, Peano の表記と Frege の概念記法の比較を表にしておきましょう.

現在の表記, Peano の表記と概念記法		
現在の表記	Peano の表記	Frege の概念記法
$a \vee b$	$a \cup b$	$\begin{array}{c} b \\ \text{---} \\ a \end{array}$
$a \wedge b$	$a \cap b$ 又は ab	$\begin{array}{c} b \\ \text{---} \\ a \end{array}$
$a \supset b$ 又は $a \rightarrow b$	$a \supset b$	$\begin{array}{c} b \\ \text{---} \\ a \end{array}$
$\neg a$ 又は $\bar{a}$ 又は $\sim a$	$\neg a$	$\text{---} a$

現在の表記が Peano 流儀の系統にある事が判るかと思います.

さて、少し複雑な命題になるとどうなるでしょうか？例えば、後述の Hilbert の命題論理式の公理系の公理 7. $(A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A)$ は



となります。同様に Peano の表記では

$$a \supset b : a \supset \neg b \supset \neg a$$

となります。この様に Peano の表記では括弧を使わずに、区切記号として $.$, $:$, $\neg$, $\supset$ を用いる方法で、最初の括弧の区切には $.$ 、次の括弧には $:$ と点が増えるので、雰囲気的には漢文の返り点の要領、例えば、「在_二伯樂_一」で、「伯樂在_レ」と読ませる方法に似ています。この様に、Peano の表記は Frege の概念記法と比較すると印刷は楽でしょうが、長い式になると実際は面倒です。

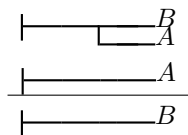
この Peano の表記と比較して、Frege の表示にはそれなりの領域を必要とし、論理式の表記は面倒な様でも、基本的に、判断 $\vdash$ 、水平線 $\rule{1cm}{0.4pt}$ 、条件 $\rule{1cm}{0.4pt}$ 、否定 $\neg$ と後述の量化



記号 $\rule{1cm}{0.4pt}$ の組合せであり、論理式の構造を把握には意外と便利な図式です。この点では漢字と同様に、この表記は、現在の GUI を用いる計算機にとって親和性が非常に高いのではないかと私は思っています。

Frege は命題を独自の表記による図式にだけでなく、この表記を用いて推論も表現しています。ここで Frege が採用している推論は前提肯定 (Modus Ponens, MP とも略記) のみで、これは次の様に表記されます。

Frege による前提肯定の表記



この表記で上の二つの論理式が仮定の命題 $A \rightarrow B$ と命題 A で、これらの仮定から命題 B が判断出来るという事を明瞭にする為に、仮定と結論の間に線分, Frege が移行記号と呼ぶ記号を入れています. この表記方法も現在の論理学の証明図式で用いられています.

因に、この前提肯定は三段論法の一つで、次の様な推論を行うものです.

——— 前提肯定の例 ———

(大前提) 雨であれば運動会は中止である。

(小前提) 雨である。

(結論) だから、運動会は中止である。

ここでの大前提「 P ならば Q である」に対し、小前提が「 P である」と大前提の前件(前提である P を肯定的に主張している為に前提肯定と呼ばれます. 猶、前提肯定は、前件肯定、或いは、分離規則とも呼ばれます.

因に、後件肯定もあります. 後件肯定は前提肯定に似た推論で、次に示す推論を行います.

——— 後件肯定の例 ———

(大前提) 雨であれば運動会は中止である。

(小前提) 運動会は中止である。

(結論) だから、雨である。

ここで、大前提の後件「運動会が中止である」を小前提が肯定している為に後件肯定と呼びます. 但し、こちらは正しくない推論になります. 実際、雨でなくても、火山の噴火でポンペイ同様になっていれば運動会をやりたいくても出来ませんから….

この Frege の概念記法は二次元的な広がりを持つ為、仮定が多くなると証明図式の表記が流石に難しくなります. そこで、Frege は幾つかの略記方法を導入しています.

例えば、前提肯定の図式で大前提の命題 $\frac{}{A \rightarrow B}$ を $\frac{}{B}$ とする事で、この命題にラ

ベル α を割当て、同様に、小前提の命題 $\frac{}{A}$ に $\frac{}{B}$ (β) と記述する事でラベル β を割当てます. そうすると次の様に略記する事が可能です.

——— Frege による前提肯定の略式表記 ———

(α): $\frac{\frac{}{A}}{\frac{}{B}}$ あるいは (β): $\frac{\frac{\frac{}{B}}{A}}{\frac{}{B}}$

先ず、最初の式では左側に (α): の様に移行記号の左側に式番号とコロン“:” が置かれ

ています。この一つのコロンで命題 α が前提肯定の大前提である事を示します。次の式では移行記号の左側に $(\beta) ::$ と表記されています。ここでの二重のコロン“::”は、命題 β が小前提である事を示すものです。更に、二つの小前提を用いた推論を行う場合、移行記号の左側に $(\alpha, \beta) ::$ と記述し、二重コロン“::”の右側にある移行記号の横棒二本引く表記になります。

更に、図式を明瞭に表現する為に、図式の代入が行われる事を明示する表記も考えています。この表記は公理の解説で説明します。

Frege は続けて、従来の論理学ではかなりの数の推論式を列挙しなければならない²⁸が、自分はこれだけを用いると述べています。

論理学の刷新

Frege は命題の意味 (Bedeutung) と意義 (Sinn) を区別しています。例えば、命題 $2^2 = 4$ と命題 $2 + 2 = 4$ は何れも真となる命題です。ところが、これらの命題の 2^2 と $2 + 2$ が処理している事は異なります。実際、前者は「2 の二乗する事」であり、後者は「2 を二度加える事」ですね。そして、結果として両者は 4 になり、論理式は同じ真理値の真になっているのです。この様に、真理値として現われる値を論理式の意味とし、論理式自体が示す主張を意義と分けて考察しています。このような命題に関する思索から Frege は分析哲学の始祖とされている所以です。その他に重要な事として、「語の意味は文での関連に於て問われるべきであって孤立して問われるべきではない」という「文脈原理」は非常に有名です。

Frege の功績で最も重要な事は、変項・関数方式をはじめて論理学に取り入れた事です。こうする事で、「 x は 0 よりも大きい」という命題を、「 x 」が主語で述語が「0 よりも大」と命題を分割するのではなく、命題自体を $\text{Greater}(x, 0)$ の様に関数で記述する様になったのです。ここで関数の考え方ですが、Frege は関数の表現は補完を要するものであると述べています。例えば、 $y = (2 + 3 \cdot x^2) \cdot x$ といった式が与えられた時、文字 x は数値を代入する事、即ち、補完によって式の意味が確定される為の場所を空ける事に役立っているだけです。そこで、Frege は文字 x をギリシャ文字の小文字の ξ で置換えます。この ξ が置かれる場所の事を項場所とし、この項場所を占有するものを項と呼びます。そして、Frege は関数の項場所を表記する為に、ギリシャ文字を母音以外の小文字を用います。この表記方法によって Frege は関数と関数値を区別しています。実際、式 $x^2 + x$ は $\xi^2 + \xi$ と表現されますが、これは Church の λ 記法による表記 $\lambda \xi. \xi^2 + \xi$ の先駆的な表記となっている事が判ります²⁹。

²⁸三段論法だけで、主語・述語の組合を考慮すると 256 種類に上るそうです

²⁹Church の λ -記法は Frege の表記の影響を受けていない独自のものだそうです [23]

そして, Frege は函数 $F(\xi)$ の値が真理値となる場合に, 函数 $F(\xi)$ の事を概念 (Begriff, concept と英訳) と名付けています. それから, $F(\Delta)$ が真となる場合, この対象 Δ の事を概念 F に属する対象と呼びます. 例えば, 概念 $\xi^2 - 4 = 0$ が真になる為には, ξ は -2 か 2 でなければなりません. 従って, -2 と 2 が概念 $\xi^2 - 4 = 0$ のもとに属する対象となります.

ここで函数 $F(\xi)$ の ξ が取り得る対象で構成されるものの事を値域と呼びます. 更に, 函数 $F(\xi)$ が真理値を取る場合, 函数 F の値域を概念の外延と呼びます. そして, 「 Γ を外延とする概念」を「 Γ -概念」, 「概念の外延」を「クラス」と呼びます.

Frege は値域を表現する為, Frege は函数の変項をギリシャ母音小文字で表現し, この変項と無氣息記号 ($\dot{}$) を組合せて表現します. 例えば, 数値函数 $\sin \xi$ の値域は $\dot{\epsilon}(\sin \epsilon)$ となり Church の λ 式風になります. 又, 函数 $\xi^2 - 4 = 0$ の値域も $\dot{\epsilon}(\epsilon^2 - 4 = 0)$ となりますが, この値域に含まれる対象は -2 と 2 です. これらの例で示す様に, $\dot{\epsilon}$ を函数の左側に置きます. そして, 右側の函数の変項は無氣息記号が置かれたギリシャ母音小文字で充足されていなければなりません. 猶, この無氣息記号が影響を及ぼす範囲を作用域と呼びます. この値域の作用域は, 複数の無氣息記号が置かれたギリシャ母音小文字がある式で, 無氣息記号を伴うギリシャ母音小文字が現われた位置から開始し, 無氣息記号を伴う同じギリシャ母音小文字が現われた個所で途切れます. つまり, この表記は LISP の λ 式に似た表記になります. 実際, $\dot{\epsilon}(\epsilon = \dot{\epsilon}(\epsilon^2 - \epsilon))$ は函数 $\xi = \dot{\epsilon}(\epsilon^2 - \epsilon)$ の値域であり, $\dot{\epsilon}(\epsilon = \dot{\alpha}(\alpha^2 - \epsilon))$ は函数 $\xi = \dot{\alpha}(\alpha^2 - \xi)$ の値域となります.

猶, Frege は真理値に関しても面白い事を述べています. これは函数の値域を一層厳密に規定する為の考察との関連で, 「 $\dot{\epsilon}(\text{---}\epsilon)$ が真であり, $\dot{\epsilon}(\text{---}\epsilon = \alpha)$ を偽であるべしと約定しよう」と述べています ([31]§10). これは Church の真理値 $\text{TRUE} =_{\text{def}} \lambda xy.x, \text{FALSE} =_{\text{def}} \lambda xy.y$ と比較しても興味深い事でしょう.

次に, 概念 $\xi + 2 = 5$ の様に帰属する対象が一つの場合, Frege は記号 $\backslash$ を用いて, $\backslash \dot{\epsilon} F(\epsilon)$ で, 概念 F に帰属する唯一の対象を表現します. この記号 $\backslash$ は印欧諸言語での定冠詞の代用を意図したものです. 例えば, 概念 $\xi + 2 = 5$ の場合, $\backslash \dot{\alpha}(\alpha + 2 = 5)$ でその対象を表現し, $\backslash \dot{\alpha}(\alpha + 2 = 5) = 3$ は真になります. この場合, 記号 $\backslash$ は現在の数理論理学の ι 記号³⁰に相当するものです.

但し, 概念 $F(\xi)$ に帰属する対象が複数存在する場合や概念 $F(\xi)$ が偽の場合は $\backslash \dot{\epsilon} F(\epsilon)$ の値は $\dot{\epsilon} F(\epsilon)$ になります. 従って, $\backslash \dot{\epsilon}(\epsilon^2 = 1) = \dot{\epsilon}(\epsilon^2 = 1)$ は $\dot{\epsilon}(\epsilon^2 = 1)$ が唯一の対象を含まないので真となり, 同様に $\backslash \dot{\epsilon}(\text{---}\epsilon = \epsilon) = \dot{\epsilon}(\text{---}\epsilon = \epsilon)$ も $\dot{\epsilon}(\text{---}\epsilon = \epsilon)$ が空となるので真になります.

Frege は変項が二つの二項函数についても議論しています. ここで二項函数とは二重

³⁰ ι 記号の詳細は, 前原 [34], pp.59-60 を参照

の補完を必要とする函数, 即ち, 項を二つ持つ函数での事です. Frege は $(\xi + \zeta)^2 + \zeta$ の様に ξ 項と ζ 項を用いて二項函数を表現します. 更に, $\xi = \zeta$ や $\xi > \zeta$ の様に値として常に真理値を持つ二項函数の事を Frege は関係と呼びます. そして, 関係 $\Phi(\Gamma, \Delta)$ が真の場合, 対象 Γ は対象 Δ に対して関係 $\Phi(\xi, \zeta)$ にあると呼びます.

そして, 二項函数 $\Phi(\xi, \zeta)$ の値域の事を Frege は重積値域と呼びますが, これは $\dot{\alpha}\dot{\varepsilon}\Phi(\alpha, \varepsilon)$ で二項函数の値域を表現したものです. 例えば, 二項函数 $\xi + \zeta$ の重積値域は $\dot{\alpha}\dot{\varepsilon}(\varepsilon + \alpha)$ と表現されます. ここで, α が二項函数の ξ 項, ε が二項函数の ζ 項に対応します.

命題を函数として表現した結果, 函数 $F(\xi)$ に対し, この函数記号 F 自体を函数記号 G で置換える事も可能になります. 現在の言葉では, Frege の概念記法は一階の述語論理ではなく, 高階の述語論理でもあります.

さて, Frege のもう一つの大きな功績は量子子を導入した事で, 多重量化を取り入れた事です.

例えば, 「全ての x に対して $\Phi(x)$ が成立する」は現在, $\forall x\Phi(x)$ と記述しますが, Frege の方法では $\overline{\text{---}}\text{---}\Phi(x)$ と表記されます. この様に, 水平線を凹まして, その窪みに変項を記述するのです. ここで, この窪みに置かれる変項の事を束縛変項と呼びます. この量化記号による作用範囲 (scope) は, この窪みから右側になります. ここで, Frege は量化記号の束縛変項を他の項と区別する為, ドイツ文字を用い, 量化記号に束縛されない自由変項は通常のラテン文字のローマン体を利用するといった工夫を行っています. そして, Frege は存在記号 $\exists$ に対応する表記を創ってはいませんが, 否定と全称記号の組合で表現しています. 例えば, $\exists x\Phi(x)$ は $\overline{\text{---}}\text{---}\Phi(x)$ で表記しています. この量化記号を用いると, 全称肯定の「全ての Φ は Λ である」は「全ての α に対し, $\Phi(\alpha)$ は $\Lambda(\alpha)$ である」と言い換えられるので, $\overline{\text{---}}\text{---}\Lambda(\alpha)$ で表現出来ます.

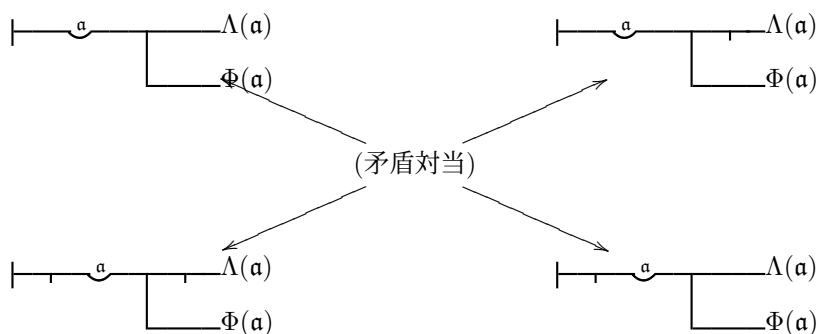
$\overline{\text{---}}\text{---}\Lambda(\alpha)$
 $\quad\quad\quad\downarrow$
 $\quad\quad\quad\Phi(\alpha)$

そこで Frege の概念記法を用いると, AEIO は以下の表記になります.

Frege による AEIO の表記

(A) 「全ての $\Phi(a)$ は $\Lambda(a)$ である」	$\overline{\text{---}}\text{---}\Lambda(a)$ $\quad\quad\quad\downarrow$ $\quad\quad\quad\Phi(a)$
(E) 「全ての $\Phi(a)$ は $\Lambda(a)$ ではない」	$\overline{\text{---}}\text{---}\Lambda(a)$ $\quad\quad\quad\downarrow$ $\quad\quad\quad\Phi(a)$
(I) 「ある $\Phi(a)$ は $\Lambda(a)$ である」	$\overline{\text{---}}\text{---}\Lambda(a)$ $\quad\quad\quad\downarrow$ $\quad\quad\quad\Phi(a)$
(O) 「ある $\Phi(a)$ は $\Lambda(a)$ ではない」	$\overline{\text{---}}\text{---}\Lambda(a)$ $\quad\quad\quad\downarrow$ $\quad\quad\quad\Phi(a)$

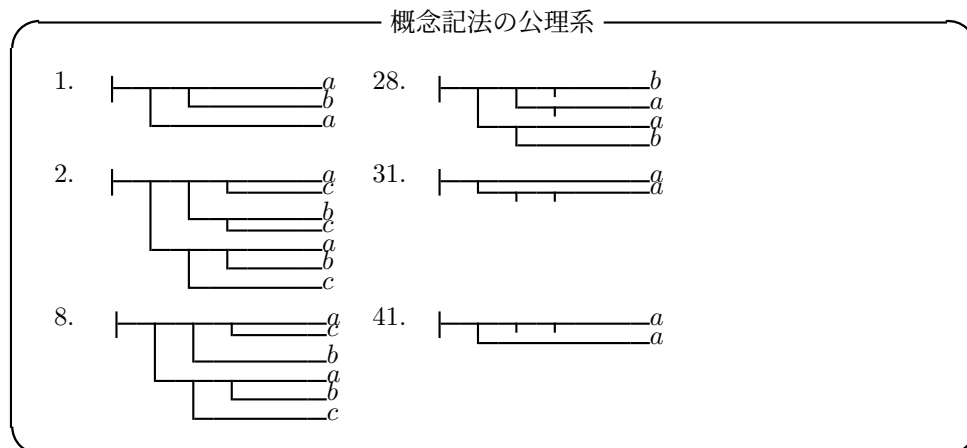
この Frege の表記で対当関係表を描くと以下のものになります.



伝統的論理学と違い, 反対関係にある全称肯定と全称否定, そして, 特称肯定と特称否定は, 共に真でも偽であり得ます. これは Frege の肯定判断の表記に存在含意 (Existence Import) が無い為です ([31], §13 の注 1 を参照). 因に, 現代の数理論理学でも存在含意は含まれていません.

「概念記法」の公理系

「概念記法」には表 4.12.3 に示す 6 個の公理があります. 表中の公理の番号は概念記法 [29] の式の番号です. ここで番号 8. の式は一つは他の公理から導出可能な為, 独立な公理は 5 個です [20].

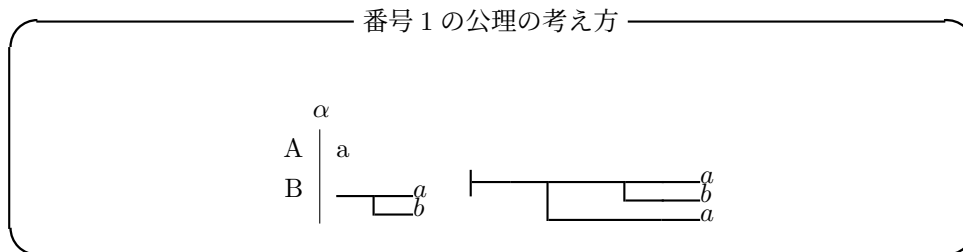


これらの公理を今風に記述すると以下のようになります.

1. $a \rightarrow (b \rightarrow a)$
2. $(c \rightarrow (b \rightarrow a)) \rightarrow ((c \rightarrow b) \rightarrow (c \rightarrow a))$
8. $(c \rightarrow (b \rightarrow a)) \rightarrow (b \rightarrow (c \rightarrow a))$
26. $(b \rightarrow a) \rightarrow (\neg a \rightarrow \neg b)$
31. $\neg \neg a \rightarrow a$
41. $a \rightarrow \neg \neg a$

最初の番号 1. の公理で $\vdash B$ の様に命題にラベル α を付ける事で, 次の様に表記

出来ます.



この図式の意味は, 左側の縦棒 | の上のラベル α で示される式 $\vdash B$ の A に a を, B に

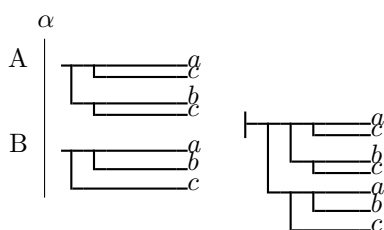
$\vdash a$ を代入する事で右側の式 $\vdash a$ が得られるというものです. ここで, $\vdash B$

の意味は, B が否定され, 且つ, A が肯定される事態を除外するので, 番号 1. の公理は $\vdash a$ が否定され, a が肯定される事態を除外する事を意味します. 更に, この $\vdash a$

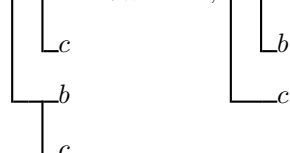
が否定されるのは, a が否定され, b が肯定される場合なので, 以上から, この公理は a が否定され, b が肯定されて更に a が肯定される事態が生じない事を意味しますが, a が肯定されると同時に a が否定されるという事は矛盾律から生じない為, この式が恒真式 (tautology) である事が判るのです.

更に, 番号 2. の公理も同様に考えられます.

番号2の公理の考え方



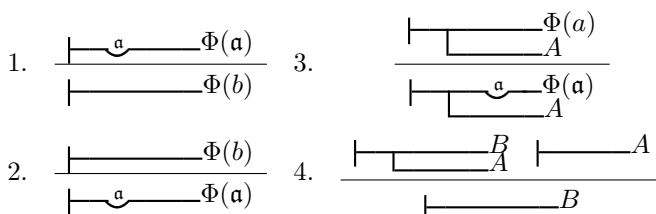
この場合も, a が否定され, a が肯定される場合を除外すると読みます.



以降, 簡単に公理の内容を解説すると, 番号8. の公理は条件の可換性を示すものです. 番号26. の公理は命題の対偶, 番号31. と番号41. の公理が二重否定の除去³¹になります.

次に, Frege はこれらの公理に加え, 正しい推論として以下の式を挙げています.

概念記法の推論規則



これらの推論を今風に記述すれば次のようになります.

$$\begin{array}{ll}
 1. \frac{\forall x \Phi(x)}{\Phi(a)} & 3. \frac{A \rightarrow \Phi(b)}{A \rightarrow \forall x \Phi(x)} \\
 2. \frac{\Phi(a)}{\forall x \Phi(x)} & 4. \frac{a \quad a \rightarrow b}{b}
 \end{array}$$

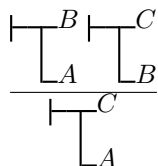
推論規則1.,2.,3. は量化記号 $\forall$ に関連したもので, 最後の推論規則4. は前提肯定 (MP) です. 次に, これらの公理系や推論規則を用いた例を示しておきましょう. その為に,

³¹五賢帝の頃の風刺作家 Lucianos の「にわとり」([45] に収録) に「二つの否定は一つの肯定である」等の哲学談義で主人公をうんざりさせるストア派の哲学者の話があります.

$\vdash B$ で $\vdash A \vee B$ を定義します.

$\vdash A$

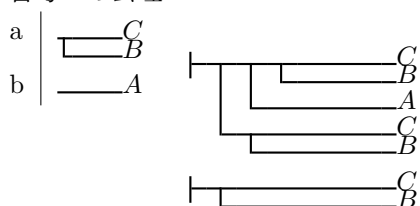
そして, $\vee$ の可換性, 即ち, $\vdash A \vee B = B \vee A$ である事を証明しましょう.



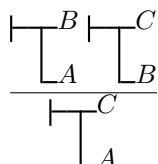
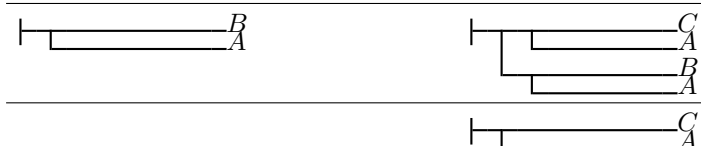
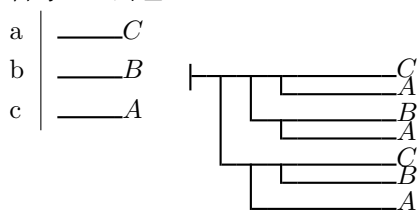
その為に, 準備として推論規則 $\frac{\vdash A \quad \vdash B}{\vdash C}$ を証明しておきます. この証明で用いる

公理は番号 1 の公理と番号 2 の公理, そして, 推論規則は MP のみです. この証明図を次に示しておきます.

番号 1 の公理

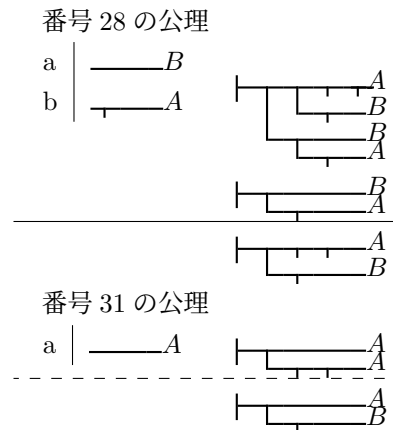


番号 2 の公理



この図式によって, 新しい推論規則 $\frac{\vdash A \quad \vdash B}{\vdash C}$ が得られます.

この証明図で $(\alpha):$ や $(\beta)::$ やを使った表記でより簡易に表示する方法があります. その表示方法で上の証明図を簡単にしてみましょう.

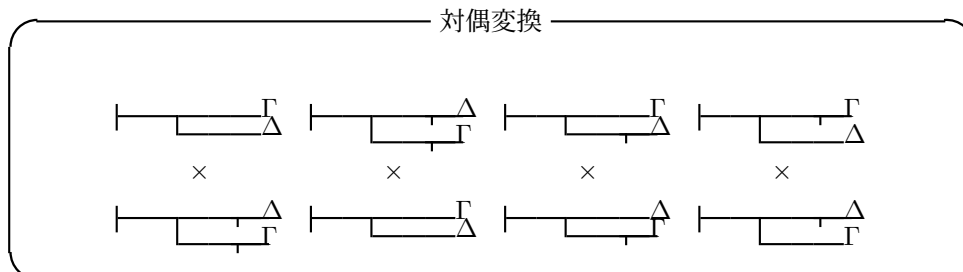


ここで、この図式の最初の段 (実線の箇所) の推論では MP, 最終段 (破線の箇所) の推論では先程証明した推論規則 (仮言三段論法) を用いています。

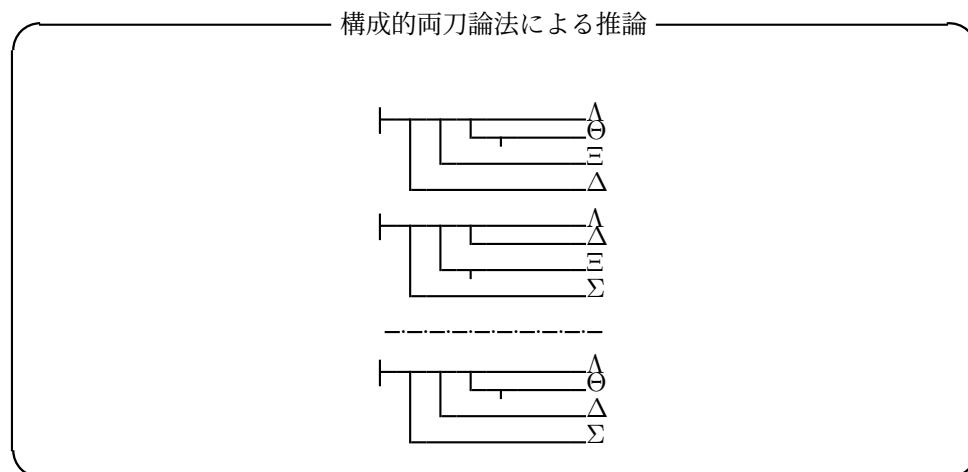
この図式から、 $\vdash B \vee A$ と判断出来、 $\neg A$ と $\neg B$ を交換する事で、 $\vdash A \vee B$ も同様に示せる為、 $A \vee B = B \vee A$, 即ち、演算子 $\vee$ は可換演算子であると判断出来ます。

同様に、 $\vdash A \wedge B$ を $\vdash \vdash B$ で定義した場合、この図式は $\vdash \left(\vdash (\neg B) \right)$ と分解されます。従って、 $\vdash A \wedge B = \vdash ((\neg A) \vee (\neg B))$ となり、ここで括弧の中の $(\neg A) \vee (\neg B)$ に注目すると、先程の $\vee$ の可換性から目的の $\wedge$ の可換性も言えます。猶、現在の表記による証明は前原 [34]pp.13-37 にもあります。この様に、論理主義は基本となる少数の公理から数学を構築するものだという事が理解されたでしょうか？

次に、Frege は記号 $\times$ を用いて命題の対偶を取る操作を表現しています。この変換によって、前者から後者 (演算子 $\times$ を挟んで上から下) にも、後者から前者者 (演算子 $\times$ を挟んで下から上) にも移行が可能です。

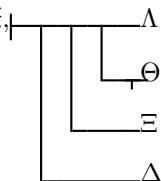


そして、推論規則として MP と対偶変換を用いて構成的両刀論法も証明しています³³.
更に、構成的両刀論法を用いた場合次に示す様に移行記号として --- を用いています.

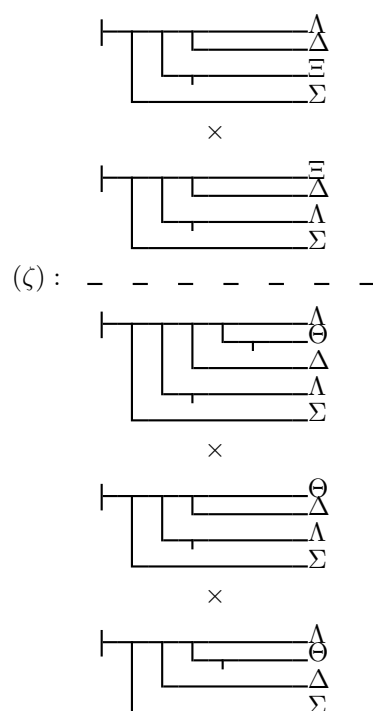


この推論の証明図は対偶変換と前提肯定を用いたものです.

実際、 Λ を ζ とすると、構成的両刀論法の証明図は次に示すものになります.



³³[31]pp.105

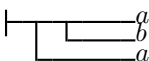
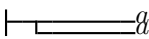
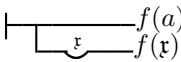
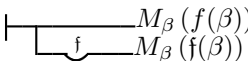
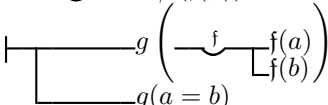
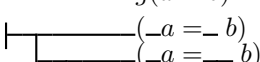
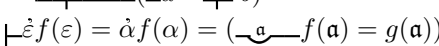


この図式に示す様に, 最初に Δ と Σ に対して対偶変換を行った命題に対し, ζ の命題を合せた前提肯定から破線下の命題が得られます. そして, $\frac{\text{---}\Delta}{\text{---}\Delta}$ が $\frac{\text{---}\Delta}{\text{---}\Delta}$ と

真理値が等しい事から, 一つの Δ で纏めて, それから Δ と Θ に対偶変換を行う事で中間の式を得, 最後に Δ と Θ に対して対偶変換を行なう事で, 目的の命題が得られる事が判るかと思います.

「算術の基本法則」の公理系

さて, Frege の論理主義は「算術の基本法則」[31]で頂点を迎える事になります. そこで, この「算術の基本法則」の公理系³⁴から代表的なものを次の表に纏め, 現在の表記との対照を付けておきましょう.

「算術の基本法則」の公理系		
Frege の表記		現在の表記
Ia. 		$a \rightarrow (b \rightarrow a)$
Ib. 		$a \rightarrow a$
IIa. 		$\forall x f(x) \rightarrow f(a)$
IIb. 		$\forall f M_\beta(f(\beta)) \rightarrow M_\beta(f(\beta))$
III. 		$g[a=b] \rightarrow$ $g[\forall f(f(a) \rightarrow f(b))]$
IV. 		$\neg(a = \neg b) \rightarrow (a = b)$
V. 		$\dot{\epsilon}.f(\epsilon) = \dot{\alpha}.g(\alpha) \equiv$ $\forall x f(x) = g(x)$
VI. $a = \lambda \dot{\epsilon}(a = \epsilon)$		$a \equiv \iota \lambda \epsilon.(a = \epsilon)$

まず, 公理 Ia は「概念記法」の公理に含まれている公理です.

公理 IIb は現在に記号で書換えると $\neg A \vee A$ で, 命題 A かその否定 $\neg A$ の何れかが成立する事, 即ち, 排中律になります. この様に, Frege の概念記法では真理値は真か偽の何れかを必ず取ります.

公理 IIa は 1 階の函数に対する普遍例化になります. 次の, 公理 IIb は単変項の 2 階函数に対する普遍例化です. ここで, M_β は一つの変項 (β) を持つ 2 階の函数を意味しています.

公理 III は代入則になります.

公理 V を現在の数理論理学の記号で書くと, $\dot{\epsilon}.f(\epsilon) = \dot{\alpha}.g(\alpha) \equiv \forall a(f(a) = g(a))$ になり, 値域が同じものに対しては, その値も等しいというものです. この公理は「算術の基本法則」で数の定義に関わる重要な公理です.

公理 VI の λ は現在の数理論理学の ι 記号に相当するもので, その値域に帰属する唯

³⁴ 「算術の基本法則」[31], pp.209-210 を参照

一の対象を示すものです. この公理 VI の $\forall \varepsilon(a = \varepsilon)$ で概念 $a = \varepsilon$ を満す対象が a だけである事を保証します.

Frege の自然数の構成

さて, Frege は「算術の基本法則」に於て, 算術が論理学から導出可能である事を示す為の議論を進めていますが, この目的遂行の為, 自然数の定義を行わなければなりません. この自然数に関しては「算術の基礎」にて彼独自の記号を極力用いずに議論を進めていますが, この基本法則では, 厳密さと明確さの為に, その自然数の定義を行う為に, 様々な記号を用いています. この記号や函数の導入を行う前に, Frege の基数を簡単に解説しておきましょう.

日常 1, 2, 3, ... と何気に使っている数ですが, 幼児にとっては難しい概念です. 更に, 幼児は非常に実在論的です. 物が無い事 (ないない...) と物が存在する事 (ばあ!) への理解は早いもので, 触れるという事も彼等にとっては非常に重要です. そこで, 顔の鼻や口に数 1 を対応付けたり, 両目に数 2 を対応付けて理解させようと苦勞する訳ですが, それで判ってくれるかどうかは? です. 因に, この Frege の基数も, 物と対応させる思考に連なるものと言えるでしょうか.

例えば, 数 6 を説明する場合, 迂闊に「6 は 5 たす 1 で得られる数」だと言えば, 「5」は何か, 「1」は何か, 更に, 「たす」とは何かという問題が出ますね. それに, 数が何であるかを説明する際に, 下手に数自身に訴えると悪循環に陥ってしまいます³⁵. その上, 実在論を信条とする幼児には判らないでしょう. そこで, 方針を変更して, 「骰子の目」の様に命題が言明する数としてみましょう. 先ず, この定義に数字は何処にも出ていないので変な循環はありません. 更に, 幼児には骰子の模様を見れば良いのです. こうすると数 6 は実体を伴う物として捉えられますね. では, 「甲虫の足」はどうでしょうか. 甲虫は昆虫なので, 足は 6 本あります. 従って, この命題は「骰子の目」と同じ数に関する言明となっています. そして, 重要な事ですが, 「骰子の目」に属する対象, 要するに面の模様と「甲虫の足」に属する対象, 即ち, 足の間に一対一の対応関係を入れる事が可能です. これも非常に現実的に, 甲虫の足に骰子の目の模様を付けてやれば良いので, 幼児にも同数性が理解出来そうです.

では, 「骰子の目」と「蛸の足」はどうでしょうか? 「骰子の目」に属する対象から, 「蛸の足」に属する対象への一意的な写像は出来ませんが, その逆写像は出来ません. この様に, 1, 2, 3, ... と対象を数えなくても, 同じ数の言明に対し, そして, その場合に限って, 概念に属する対象の間に一対一の関係が存在する事が判ります. ここで, 「概念 F に族

³⁵ 男を調べると「女ではない人間」, 女を調べると「男ではない人間」の様に循環したり, 男の事を「人間の雄」, 女の事を「人間の雌」の様な言い換えをする辞書では困りますね.

する諸対象を概念 G に属する諸対象に一对一で対応付ける関係 φ が存在する」場合に、「概念 F が概念 G と等数的である」と呼ぶ事にしましょう。この事は「Hume の原理」(HP とも略記)と呼ばれ、Frege の算術の導出で核となる重要な原理です。さて、この「Hume の原理」によると、「骰子の目」と等数的な概念の間には一対一対応の関数が存在するので、同値関係が入れられます。だから、「骰子の目と等数的な概念」で数 6 が定義出来そうです。更に、この様に個数言明と「Hume の原理」を組合せてやれば、基数の定義さえも出来そうです。

ところが個数言明と「Hume の原理」で数を定義するには大きな問題があります。例えば、「数 Julius Caesar」といった命題はどうでしょうか？これは「Julius Caesar 問題」と呼ばれる問題で、この命題の Julius Caesar が歴史的な人物(ガリアの征服者)であるのか、それとも何かの数なのか、全く判断が出来ません。「Hume の原理」は数を与えられると等しいかどうかの判断が出来ますが、与えられたものが数であるかどうかの判断が出来ないのです。即ち、個数言明と「Hume の原理」で数を定義するには能力不足なのです。

そこで、Frege は個数言明ではなく「概念の外延」、即ち、「クラス」を用いる集合論的な手法を採用します。そして、「クラス」によって Frege は「Julius Caesar 問題」は取り敢えず解決したものと考え、基数を「あるクラスと等数的なクラス」として与えています。従って、数 6 は「骰子の目のクラスと同数のクラス」³⁶と言えます。

この事から、数の定義を行う為には、対象がある概念に帰属する事や、ある概念に属する対象を別の概念に属する対象に一对一で対応付ける写像が存在する事を表現し、その上で「あるクラスと等数的なクラス」を定義しなければなりません。

以降の記号や関係の定義は、この事を目的としています。猶、Frege は各種関係の定義で記号 $\Vdash$ を用いています。この記号 $\Vdash$ は $=_{def}$ に相当する記号で、 $\Vdash \Gamma = \Delta$ で $\Delta =_{def} \Gamma$ の意味になります。

³⁶この様に、基数はクラスのクラスになります

以下, 基本的な記号の定義を述べます³⁷.

基本法則の重要な記号の定義	
記号	定義式
$a \smallfrown u$	$\Vdash \lambda \alpha \left(\begin{array}{c} \text{---} \text{g} \text{---} \\ \text{---} \text{---} \text{g}(a) = \alpha \\ \text{---} \text{---} u = \varepsilon g(\varepsilon) \end{array} \right) = a \smallfrown u$
I_p	$\Vdash \left(\begin{array}{c} \text{---} \text{e} \text{---} \text{d} \text{---} \text{a} \text{---} \text{d} = \text{a} \\ \text{---} \text{---} \text{e} \smallfrown (\text{a} \smallfrown p) \\ \text{---} \text{---} \text{e} \smallfrown (\text{d} \smallfrown p) \end{array} \right) = I_p$
$\rangle p$	$\Vdash \dot{\alpha} \dot{\varepsilon} \left[\begin{array}{c} \text{---} \text{d} \text{---} \text{a} \text{---} \text{d} \smallfrown \varepsilon \\ \text{---} \text{---} \text{a} \smallfrown \alpha \\ \text{---} \text{---} \text{d} \smallfrown (\text{a} \smallfrown \xi) \\ \text{---} \text{---} I_p \end{array} \right] = \rangle p$
$\mathbb{I} p$	$\Vdash \dot{\alpha} \dot{\varepsilon} (\alpha \smallfrown (\varepsilon \smallfrown p)) = \mathbb{I} p$
$\perp$	$\Vdash \dot{\alpha} \dot{\varepsilon} \left(\begin{array}{c} \text{---} \text{r} \text{---} \text{---} \varepsilon \smallfrown (\text{r} \smallfrown p) \\ \text{---} \text{---} \text{r} \smallfrown (\alpha \smallfrown q) \end{array} \right) = p \perp q$

まず, 記号 $\smallfrown$ は対象の概念への帰属を表現する為に用いる記号です ([31]§34). 即ち, $\Delta \smallfrown \varepsilon \Phi(\varepsilon)$ が真である事は, $\Phi(\Delta)$ が真になる事と同値で, 対象 Δ が概念 Φ に帰属する事を意味します. この式を現在の表記を併用して表現するならば, u が概念の場合に $a \smallfrown u =_{def} a \in u$ が対応し, u が概念でない場合には真理値の偽が対応します.

次の I_p は関係 p の多対一性を表現する式です ([31]§37). まず, この式では重積値域に対して $\smallfrown$ を作用させています. ここで二項函数 $\Phi(\xi, \zeta)$ の重積値域は $\dot{\alpha} \dot{\varepsilon} \Phi(\varepsilon, \alpha)$ ですが, ここで, $\Delta \smallfrown (\dot{\alpha} \dot{\varepsilon} \Phi(\varepsilon, \alpha))$ は函数 $\Phi(\xi, \Delta)$ の値域 $\dot{\varepsilon} \Phi(\varepsilon, \Delta)$ に対応します.

そして, $\Gamma \smallfrown (\Delta \smallfrown (\dot{\alpha} \dot{\varepsilon} \Phi(\varepsilon, \alpha)))$ には $\Phi(\Gamma, \Delta)$ が対応します. この様に重積値域の先頭が $\dot{\alpha} \dot{\varepsilon}$ であれば, $\smallfrown$ を挟んで隣り合った α と対象が, その対象を用いて充当され, 最後の $\smallfrown$ で ε が残った対象で充当される仕組みになっています.

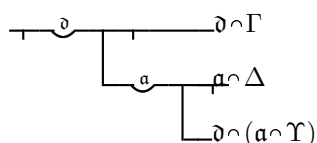
さて, I_p の定義式の意義ですが, これは二つの概念間の p -関係が一意性を持つ事を示すものです. さて, 対象 Γ と Δ , 関係 $p(\xi, \zeta)$ で考えてみましょう. ここで $\Gamma \smallfrown (\Delta \smallfrown p)$ は $p(\Gamma, \Delta)$, 即ち, Γ が Δ と p -関係にある場合, p -関係の一意性は, 任意の Γ, Δ, Λ に対し, $p(\Gamma, \Delta)$ で $p(\Gamma, \Lambda)$ の場合に, $\Delta = \Lambda$ となる事, 即ち, 任意の y に対して $p(x, y)$ を満す x は存在しないか, 存在しても唯一である事です. 現在の数理論理学の式では $\forall x \forall y \forall z [p(x, y) = p(z, y) \rightarrow x = z]$ で置換えられるでしょう.

函数 $\rangle \xi$ は概念間の写像の単射性を表現するものです. この函数は重要なので, 「算術の基本法則」 ([31]§ 38) に沿って解説します.

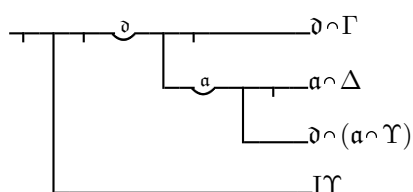
まず, 「 a が概念 F に属する」と 「 a が概念 G に属する対象に対して関係 φ がない」

³⁷ 特殊な記号を用いる為, fge パッケージ [87] を利用しています. このパッケージは TeXlib にも含まれていないので, 自力でインストールしなければなりません.

という二つの命題は全ての a に対して両立しないと言い換えられます. ここで, 概念 F を $\text{---}\xi\wedge\Gamma$, 概念 G を同様に $\text{---}\zeta\wedge\Delta$, 関係 φ を $\text{---}\xi\wedge(\zeta\wedge\Upsilon)$ とすると, 関係 Υ が概念 F と概念 G に帰属する対象を対応付ける事は



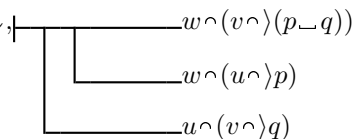
で表記されます. 更に, この対応関係は一意 (多対一) の関係でなければならず, その為には, $I\Upsilon$ が真でなければなりません. このことから,



となります. この式の重積値域を取出し, 項 Υ の函数と見做したものが $\rangle\Upsilon$ になります. そして, $\Gamma\wedge(\Delta\wedge)\Upsilon$ が真であれば, Υ -関係は Γ -概念を Δ -概念に写像すると言います. これで概念 Γ から概念 Δ への単射の存在が表現出来ました. すると, 今度は概念 Δ から概念 Γ への単射の存在が必要になります.

そこで, 逆関係というものを導入します. この記号 $\mathbb{I}$ は逆関係を表現する記号です. 猶, この定義式を現在の表記で一部置換えると, 関係 $p(\xi, \zeta)$ に対し, $\mathbb{I}p =_{def} p(\zeta, \xi)$ となります.

記号 $\sqcup$ は二つの関係の合成で用います. 即ち, 関係 p , 関係 q に対し,



が成立つもので, この関係 p と関係 q を $p \sqcup q$ 関係と呼びます. 猶, Frege は $p \sqcup q$ 関係を直観的に $w \xrightarrow[p]{\rightsquigarrow} u \xrightarrow[q]{\rightsquigarrow} v$ と図示しています.

基数

Frege は基数を概念 F と概念 G の等数性に訴えています. その為には, 概念 F に属する諸対象が概念 G に属する諸対象に一意的に対応し, 更に, その逆も成立しなければなりません. このことから, 概念 F を $\text{---}\xi\wedge\Gamma$, 概念 G を $\text{---}\zeta\wedge\Delta$ とし, 概念 F

Diagram 1: A tree structure with root q . The root has two children: $\xi \cap (\Delta \cap q)$ on the left and $\Delta \cap (\xi \cap q)$ on the right. The left child has a single child $\xi \cap \Delta$. The right child has a single child $\Delta \cap \xi$.

$$\mathfrak{E} \left(\begin{array}{c} \text{---} \text{q} \text{---} \xi \cap (\Delta \cap \text{q}) \\ \quad \quad \quad \Delta \cap (\xi \cap \text{q}) \end{array} \right)$$
$$\models_{\mathcal{E}} \left(\text{Diagram} \right) = \eta u \quad (Z)$$

Hume の原理 ([31]§ 32 参照)

(定理 32)

この式の意義は、逆関係 $\mathbb{I}q$ によって概念 v は概念 u に写され、関係 q によって概念 u は概念 v に写さるのであれば、概念 u の基数 $\#u$ と概念 v の基数 $\#v$ が等しいという事を主張しています。この「Hume の原理」を用いて Frege は自然数を構築してゆくのです。

自然数の構築

さて, Frege はどのようにして自然数を構築したのでしょうか, 実際にその構築を追ってみましょう. 先ず, 数 0 は「それに帰属する対象が存在しない概念に属する数」として定義し, 次の様に表記しています:

$$\Vdash \eta \dot{\varepsilon}(\bigwedge \varepsilon = \varepsilon) = 0 \quad (\Theta)$$

ここで, $\dot{\varepsilon}(\bigwedge \varepsilon = \varepsilon)$ は「自分自身が異なる対象という概念」の外延ですが, この様なクラスは対象を何も含まないので, 空集合 $\emptyset$ になります. ここで集合論の自然数の構成で, 0 に $\emptyset$ が対応していましたが, Frege の定義もこれと似たものとなっています. そして, Frege は基数 0 と独特の表記を用いていますが, これは実際の数 0 と基数 0 が異なる数だからです. これは Frege の基数は個数という量に関する数ですが, 実数は単位量の比として現われる数の為, 基数を実数にそのまま拡張する事が出来ないからです. さて, 次の数 1 はどのような数になるのでしょうか? 数 1 は基数 0 と異なり, 「帰属する対象を一つだけ持つ概念の外延」となります. Frege はその対象が基数 0 に等しいものと定義しています:

$$\Vdash \eta \dot{\varepsilon}(\varepsilon = 0) = 1 \quad (I)$$

この基数 1 の定義も, 集合論に於ける自然数の構成では, 1 に $\{\emptyset\}$ が対応しており, この点でも似ていますね. これで基数 0 と基数 1 が定義されました.

残りの自然数を構築する為に, Frege は今度は「直続」という概念を導入しています. この直続を言葉で説明すると次の様になります.

—— 自然数 n が自然数 m の直続 である事の定義 ——

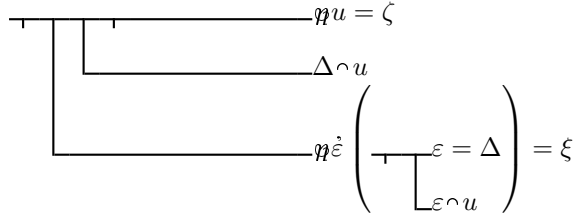
ある概念 F と, その下に属するある対象 x が存在し, 自然数 n が概念 F に帰属する基数であり, 且つ, 概念 F に帰属するが, x と等しくないという概念 G に帰属する基数が m であれば, 自然数 n は自然数 m に直続すると呼ぶ.

要するに自然数 n が自然数 m に直続するとは, $n = m + 1$ となる事です. この事は, 自然数 n が属する概念 F は自然数 m が属する概念 G よりも一つ対象が多い事になります. そうすると, 概念 G の対象は概念 F と一意に対応付けられますが, その時, 概念 F のある対象 x は余ってしまいますね. 直続とは, この性質を述べたものです.

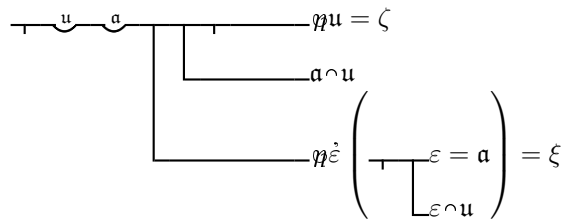
さて, この直続関係を概念記法で表記してみましょう.

先ず, 概念 u を基数 n が帰属する概念, 対象 Δ を基数 m が帰属する概念に属さない対象で, 概念 u に属する対象とします. すると, $\eta \dot{\varepsilon} \left(\bigwedge_{\varepsilon \in u} \Delta \right)$ で, 基数 m が表記出来ます. そして, Δ が概念 u に属する事は $\Delta \in u$ で表記出来ます.

次に, 基数 m が帰属する概念を ξ , 基数 n が帰属する概念を ζ とすると, 概念 ξ と概念 ζ は次の関係を満す事になります.



さて, ここでは n が帰属する概念について述べているので, 概念 u をある概念 u で置換えて Δ もある対象 a で置換える事で次の関係を得ます:

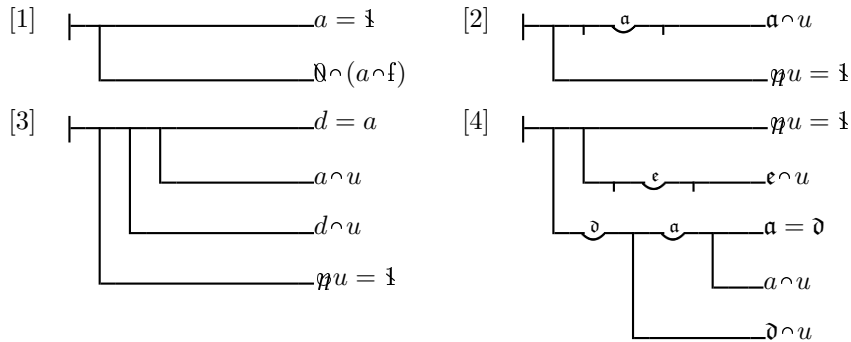


この関係の重積値域の事を直続関係 f として定義します:

$$\Vdash \hat{\alpha} \hat{\epsilon} \left[\begin{array}{c} \text{---} u \text{---} a \text{---} \\ \text{---} \mu u = \alpha \\ \text{---} a \wedge u \\ \text{---} \mu \hat{\epsilon} \left(\begin{array}{c} \text{---} \epsilon = a \\ \text{---} \epsilon \wedge u \end{array} \right) = \epsilon \end{array} \right] = f$$

ここで $\mathbb{Q}$ と $\mathbb{1}$ の関係を考えてみましょう. $\mathbb{Q}$ に対応する概念には対象を何も持ちませんが, $\mathbb{1}$ に対応する概念は $\mathbb{Q}$ を持ちます. 従って, $\mathbb{1}$ は $\mathbb{Q}$ の直続関係, 即ち, $\mathbb{Q} \wedge (\mathbb{1} \wedge f)$ が成立する事が判ります.

この他にも次の事が成立します ([31]§ 44 参照):



先ず,[1] は0と直属する基数は1に限られる事を示しています. 次の [2] は基数が1に等しい概念 u に関しては, その概念に属する対象が存在する事を示しています. そして,[3] は a と d が基数1となる概念 u の下に属する対象であれば, $d = a$ である事を示します.[4] は概念 u に属する対象が全て一致し, 帰属する対象が存在する場合, その概念が基数1である事を示すものです.

さて, Frege の記号を用いずに, 基数を $\langle n \rangle$ と平易に記述しましょう³⁸. そこで, 直続関係を用いる事で, 基数 $\langle 0 \rangle$ から $\langle 1 \rangle$ が定義され, 以下同様に, 基数 $\langle 2 \rangle, \langle 3 \rangle, \dots$ が続々と定義されます. そこで, Frege の方法による自然数の定義を次に纏めておきましょう.

— Frege による自然数の定義 —

- $\langle 0 \rangle$ = 「概念 $\xi \neq \xi$ と同数である」という概念の外延
- $\langle 1 \rangle$ = 「0 と同一」という概念に属する数
- $\langle 2 \rangle$ = 「0 又は 1 と同一」という概念に属する数
- $\langle 3 \rangle$ = 「0 又は 1 又は 2 と同一」という概念に属する数
- $\vdots$
- $\langle n \rangle$ = 「0 又は 1 又は $\dots$ 又は $(n-1)$ と同一」という概念に属する数

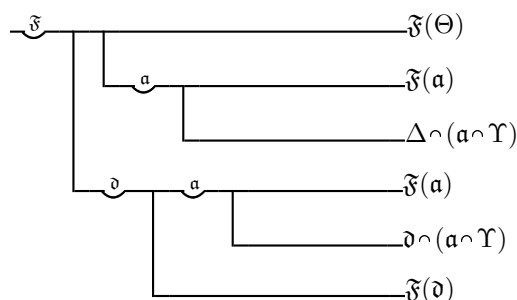
勿論, 直続があれば後続もあります. これは一種の性質の伝播を示すものです. 次に, この後続の定義を述べておきましょう:

— 後続の定義 —

対象 Δ と対象 Θ に対し, Δ と Υ -関係にある各対象が, 概念 $\text{——}F(\xi)$ に属し, この概念 $\text{——}F(\xi)$ に属する対象と Υ -関係となる対象も概念 $\text{——}F(\xi)$ に属し, 対象 Θ も概念 $\text{——}F(\xi)$ に属する場合, 「対象 Θ は Υ -系列に於て対象 Δ に後続する」と呼ぶ.

この事を概念記法で記述すると次の式を得ます:

³⁸ 基数 0, 基数 1 があっても基数 2 以降のフォントが無い為の苦肉の策

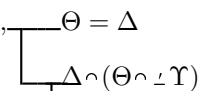


さて、ここで対象 Δ と対象 Θ を変項 ξ と ζ で置換し、更に、関係 Υ を関係 q で置換える事で二項関数が得られます。そして、この関数の重積値域を $\perp q$ と表記します。即ち、以下の定義になります:

$$\vdash \dot{\alpha} \dot{\varepsilon} \left[\begin{array}{l} \dot{\mathfrak{F}} \\ \dot{\mathfrak{F}}(\alpha) \\ \dot{\mathfrak{F}}(\mathfrak{a}) \\ \varepsilon \cap (\mathfrak{a} \cap \Upsilon) \\ \dot{\mathfrak{F}}(\mathfrak{a}) \\ \dot{\mathfrak{d}} \cap (\mathfrak{a} \cap \Upsilon) \\ \dot{\mathfrak{F}}(\mathfrak{d}) \end{array} \right] = \perp q$$

ここで $\Delta \wedge (\Theta \wedge \neg \Upsilon)$ が真であれば「 Θ が Υ -系列に於て Δ に後続する」，或いは「 Δ は Υ -系列に於て Θ に先行する」と呼びます．そして，この関係の事を強先祖関係と Frege は名付けています．

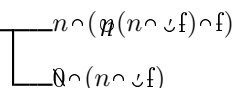
次に、強先祖関係を用いて弱先祖関係と Frege が呼ぶ関係を定めます。まず、 $\text{---}\Theta = \Delta$



は Θ が Υ -系列に於て Δ に後続するか Δ と一致する事の真理値となります。これを Frege は「 Θ が Δ で開始する Υ -系列に所属する」, 或いは, 「 Δ は Θ で終了する Υ -系列に所属する」と呼びます。それから, この関係の外延を考えて弱先祖関係の記号 $\prec$ を次で導入します:

$$\Vdash \dot{\alpha} \dot{\varepsilon} \left(\frac{\perp \alpha = \varepsilon}{\perp \varepsilon \cap (\alpha \cap \perp q)} \right) = \cup q$$

ここで $\mathbb{N} \cap (\Theta \cup f)$ は基数 Θ が基数 $\mathbb{N}$ から開始する基数列に所属する事を意味し、この様な基数 Θ の事を有限基数と呼びます。そして、 $\Theta \cup \gamma$ で「 Θ で終わる基数列に所属する」事を意味し、「有限基数 n で終わる基数列に所属する基数」は $\text{---} \mathbb{N} \cap (\vartheta(n \cup f) \cap f)$



と記述出来ます。

そして、基数0から開始する基数列に所属する対象は自分自身に後続しません。この事は概念記法で次の様に表記出来ます:

$$\begin{array}{|l} \hline b \wedge (b \wedge f) \\ \hline \mathbb{N} \wedge (b \wedge f) \end{array}$$

更に, 有限基数に対して Frege は次の定理 155 を述べています:

$$\begin{array}{|l} \hline b \wedge (\mathbb{N}(b \wedge f) \wedge f) \\ \hline \mathbb{N} \wedge (b \wedge f) \end{array} \quad (\text{定理 155})$$

この式は基数 $\mathbb{N}$ から開始し, 基数 b で終了する基数列に所属する基数は b が有限基数であれば b に直続するという命題です.

そこで, Peano の自然数の公理系との対比してみましょう. そこで, Peano のラテン語の論文の英訳「The principle of arithmetics, presented by a new method」([51], pp.81-103) からの表記を併せて示しておきます.

☆第1公理 (0 の存在) 自然数の第1公理は数0の存在に関する公理です. Peano の本来の記述では $1 \in \mathbb{N}$, 現在の表記では $1 \in \mathbb{N}$ と0ではなく1を用いています. 猶, Peano 自身は0や1の定義を行わず, 無定義述語として用いています. Frege の概念記法では基数 $\mathbb{N}$ の定義 (Θ) が相当するでしょう:

$$\vdash \mathbb{N} \dot{\varepsilon} (\bigwedge \varepsilon = \varepsilon) = \mathbb{N} \quad (\Theta)$$

☆第2公理 (後者の存在性) 自然数の第2公理は後者の存在性です. 本来表記で $a \in \mathbb{N} . \bigcirc . a + 1 \in \mathbb{N}$, 現在の表記では $a \in \mathbb{N} \rightarrow a + 1 \in \mathbb{N}$ となります. これは Frege の次の定理 157 が対応するでしょう:

$$\begin{array}{|l} \hline a \quad b \wedge (a \wedge f) \\ \hline \mathbb{N} \wedge (b \wedge f) \end{array} \quad (\text{定理 157})$$

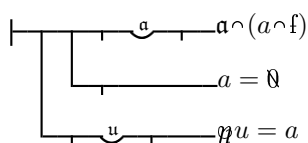
この定理は基数 $\mathbb{N}$ から開始する基数列に所属する任意の対象 b に直続する対象が存在するというものです. これによって基数を次々と構築して行く事が可能になります.

☆第3公理 (0 の先行者の非存在性) 本来の Peano の定義では公理1と同様に1に対するもので $a \in \mathbb{N} . \bigcirc . a + 1 = 1$., 現在の表記では, $a \in \mathbb{N} \rightarrow \neg(a + 1 = 1)$ と, 1を遡る事が出来ないというものです.

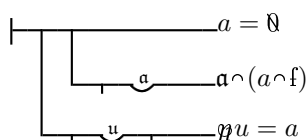
この公理には次の定理 108 が対応します:

$$\vdash \neg c \wedge (\mathbb{N} \wedge f) \quad (\text{定理 108})$$

次の定理 ([31]§ 44 の [6]) も挙げておきます:



この定理の意味は, 基数 0 を例外として, 各基数は基本列に於て, 直接の先行者が一つ存在するというものです. この定理の対偶として次が得られます:

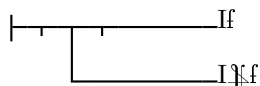


則ち, 基数 a が先行する基数を持たなければ a は 0 と結論付けられます.

☆第 4 公理 (後者関係の一意性) 第 4 公理は後者関係の一意性を示すもので

$a, b \in \mathbb{N}. \cup : a = b. = .a+1 = b+1.$, 現在の表記ならば $a, b \in \mathbb{N} \rightarrow a = b \equiv a+1 = b+1$ と表記されます.

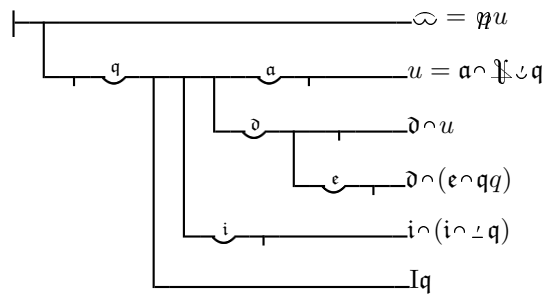
この公理は $\vdash \text{If}$ (定理 78, と $\vdash \text{If} f$ (公理 89, の連言として表現されます:



ここでの If は f-関係が一意的, 即ち, 基本列に於て, 直続する基数は一つ以上存在しない事を意味し, $\text{If} f$ も同様に後続する基数が一つしか存在しない事を意味します.

☆第 5 公理 (数学的帰納法) 数学的帰納法は Peano の本来の表記では

$k \in K \therefore 1 \in k \therefore x \in \mathbb{N}. x \in k : \cup_x. x+1 \in k :: \cup. \mathbb{N} \cup k.$, 現代の表記では $((k \in K \wedge 1 \in k \wedge x \in \mathbb{N} \wedge x \in k) \rightarrow x+1 \in k) \rightarrow \mathbb{N} \subset k$ となります.



これらの命題から得られる Frege の算術は Peano の算術と同型となります。

この様に Frege は一見無限とは無関係そうな「Hume の原理」だけで無限列を構成し、最終的に $\aleph_0$ に到達しているのです。

破綻

この様にざっと「算術の基本法則」を眺めてみました。この Frege の体系は非常に強固なものに思えますが、その強固さに思わぬ脆弱性が潜んでいました。

この「算術の基本法則」は表記の複雑さ故に出版を渋られた事もあるが、「算術の基本法則」を二巻に分け、第 I 巻の売上によって後の第 II 巻を刊行する方針で、やっと 1893 年に第 I 巻を出版します。しかし、第 II 巻は結局、その 10 年後の 1903 年に自費で出版する羽目になっています。その上、第 II 巻の校正中の 1902 年に、Russell から「Russell の逆理」に関する手紙を受け取ります⁴⁰。この逆理は Frege の論理主義の公理 V から導き出せるもので、付録として対処法をなんとか第 II 巻に載せたものの、彼の没後 10 年後に、この対処法でも逆理が導出可能な事が示されています。

この「Russell の逆理」の解説は II 巻の付録にあります。ここは Frege による説明を追ってみましょう。猶、Frege は真理性が疑わしい為に、判断線 $\vdash$ を落した式を用いています。

先ず、 Δ が自分自身に属さないクラスであるという事を、次の式で表現しています。

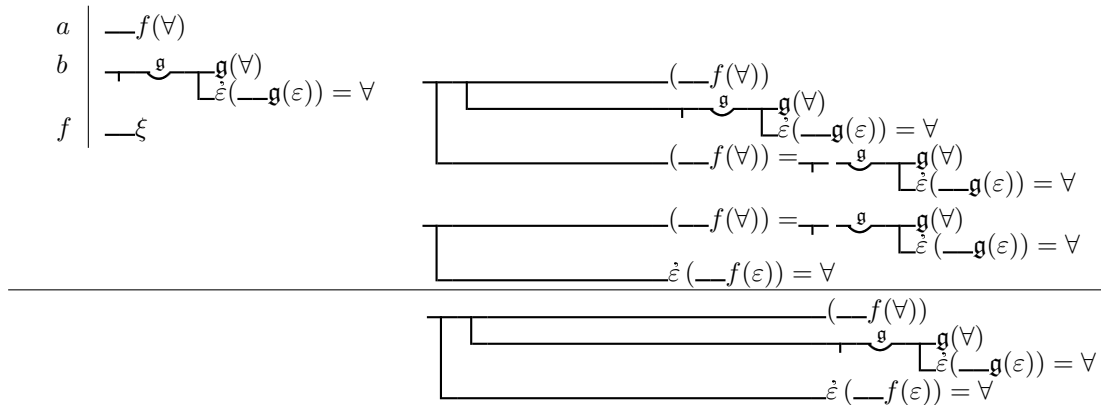
$$\begin{array}{l} \vdash \text{g} \quad \text{g}(\Delta) \\ \vdash \text{g}(\text{g}(\epsilon)) = \Delta \end{array}$$

何故この式で表現出来ているのか、その理由を解説しておきましょう。

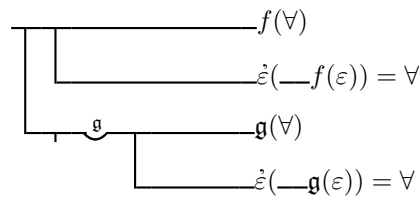
先ず、 $\text{g}(\text{g}(\epsilon)) = \Delta$ で g の外延が Δ に等しい事を意味します。そして、 $\vdash \text{g}(\Delta)$ により、 Δ は g の外延に属さない事を意味します。だから、これらを組合せる事で Δ が

⁴⁰その手紙の内容は [20], pp.204 にあります

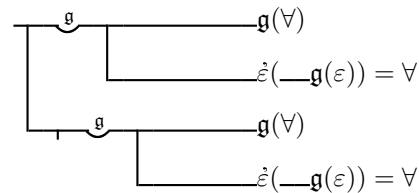
公理 IIIa



ここで、結果の条件を入れ替える事で次の式が得られます:



更に、 f は任意の概念なので、この f に対して全称化を行ってみましょう。
 すると、見事に矛盾が生じています!



ここで問題となるのは排中律と公理 V ですが、Frege は分析を進める事で、公理 V が偽であると最終的に判断しています。

この公理 V は簡単に言えば、「函数 $\Phi(\xi)$ は函数 $\Psi(\xi)$ と同じ値域を持つ」という事と「函数 $\Phi(\xi)$ 函数 $\Psi(\xi)$ とは同じ項に対して常に同じ値を持つ」という事が同値である事を保証する公理です。基数の定義で判る様に、外延 (値域) を根底に置いて自然数の定義を行っている為、公理 V は体系全体に影響を与えてしまいます。

猶、この公理 V に対しては、公理的集合論では制限を加え、Russell の論理主義では型の理論を導入する事で Russell の逆理を排除しています。Frege は値域を持たない概念の存在 (公理的集合論の方法) や型の理論を用いた解決を示唆しますが、Frege の取った

手段はそのどちらでもありませんでした。但し、この解決方法も Frege の没後 10 年後に矛盾が導出され、結局、この逆理に対する根本的な対処が出来ずに Frege の論理主義は破綻してしまいます。但し、近年の研究では外延を用いずに、「算術の基礎」で最初に取り上げられた数の言明と「Hume の原理」を用いる事で、Peano 算術が遂行可能な事が示されています。その事もあって、再評価が進んでいるそうです (田畑 [20])。

4.12.4 Russell の型理論

Frege の論理主義は Russell の逆理で大きく頓挫する事となりましたが、Frege と入れ替わる様に今度は Russell が論理主義を押し進めます。ここでは Russell の「Mathematical logic as based on the theory of types(1908)」([51], pp.150-182) に沿って解説しましょう。

先ず、変項を持たない命題の事を基本命題 (Elementary proposition) と呼びます。項 (term) は命題の主語となり得る個体の事であり、基本命題の項の事を個体 (individual) と呼びます。この項や個体が最低層を構成します。そして、概念 (concept) は命題の述語や関係の事です。

次に、Russell の函数の考え方は、Frege の「不飽和」といった考えとやや異っています。変項は $x, y, z, \dots$ で不定値 (曖昧値) を表現し、意味のある値は $a, b, c, \dots$ で記述します。そして、曖昧値 x に対する函数値を ϕx と表記しますが、函数自体の表記は $\phi \hat{x}$ と表記します。この函数表記は Frege の $\phi(\xi)$ の表記に対応するものと考えても良いでしょう。

ここで基本命題 φ に含まれる個体 a に対する代入操作で命題函数が構築されます。この命題函数を $\varphi/a : x$ と記述し、 φ の事を雛形、 φ/a の事を母体 (matrix) と呼びます。猶、個体 a の他に $b, c, \dots$ を用いて $p/(a, b), p/(a, b, c), \dots$ と母体が構築されます。

さて、Poincaré の言う非述語的な言明による逆理に対しては、次の「悪循環原理」で、その発生を封じ込めようとしています：

悪循環原理

1. ある集まりの全てを含む物は、その集まりの一つの元であってはならない。
2. ある全体を有している集まりが、その全体でのみ定義される元を持つのであれば、その集まりは何如なる全体も持たない。

悪循環原理によって、Russell の逆理の「自分自身を含まない全ての対象」は自分自身への言及がある為、「Russell の逆理」はこの体系から排除されます。

それから個体や命題に型という階層構造を導入します. 先ず, 個体が最下層の 0 階を構成し, 第 1 階は基本命題と個体を明瞭な変数として持つ命題関数で構成されます. そして, 第 2 階の命題関数は第 1 階の命題関数を明瞭な変数として持つ命題関数です. 以降同様に, 第 3 階…と命題関数の階が構成されます. 但し, この階は相対的なものであり, $n-1$ 階, n 階と $n+1$ 階の三種類あれば十分です.

ここで命題関数に含まれる変数の最大階数が n で命題関数が $n+1$ 階であれば, この命題関数を述語関数 (Predicative function) と呼び $\phi!x$ と表記します.

さて, Epimenides の嘘つきの逆理: 「自分 (Epimenides) が主張する命題は嘘である」を φ と置いてみましょう. ここで命題 φ の中の個体の「命題」を第 1 階の命題とすれば, φ を雛形として得られた述語 $\varphi/\text{命題}: x$ は変項として第 1 階の命題を取ることで第 2 階の命題関数となります. 則ち, 「自分が主張する第 1 階の命題は嘘である」という第 2 階の命題になりますが, 彼が主張する第 1 階の命題が嘘であろうがなかろうが, 第 2 階の真理値とは階が違う為に逆理が回避されます.

これで Russell の逆理の排除や嘘つきの逆理も目出たく解決出来た様に見えますが, 悪循環原理の導入によって数学の多くの成果が失われてしまう副作用があります. 更に, 型の導入によって数学的帰納法さえも利用出来なくなります. 例えば, 「0 で性質 F を持ち, n で性質 F を持つのであれば n の後者も性質 F を持つのであれば, 全ての自然数は性質 F を持つ」は, 「0 が性質 F を持つ」, 「 n が性質 F を持つ」と「 n の後継者が性質 F を持つ」は全て 1 階の命題ですが, 「全ての自然数が性質 F を持つ」の「全ての自然数」は「悪循環原理」によって命題「自然数 0」と同じ 0 階には置けず, 1 階の対象となります. その為, 「全ての自然数が性質 F を持つ」は 2 階の命題となり, 階が異なる為に結論付ける事が出来なくなるので, その結果, 帰納法が使えなくなるのです.

この対処方法として Russell は還元可能性公理 (Axiom of reducibility) を導入する事で, この難点を打開しようとしています. この公理を Russell の記号を用いて表記しておきましょう:

還元可能性公理 (還元公理)

$$\exists \varphi : .(x) : \phi x. \equiv_x .\varphi!x.$$

この還元可能性公理の意味は, 「任意の命題関数 ϕ に対し, 同値な述語 φ が存在する」で, 現在の表記では $\exists \varphi [\forall x (\phi(x) = \varphi!x)]$ となるでしょうか. 猶, “:.” や “.” は Peano 流の区切記号で, 式の区切を表現しています.

この公理によって, 任意の命題関数には, その対象よりも 1 階上の集まり (クラス) の存在が保証されます. 但し, この公理の難点は, その述語が存在すると主張するもので,

構成方法を述べたものではない事, 更に, この還元可能性公理自体が実は非述語的な性質を持っている事です. 又, 還元可能性公理は命題関数を満すクラスを, その命題と同値な述語で一時的に定義し, その述語を暫く使って元の命題関数に戻す事を便利良く使っており, 幾分, 「機械仕掛けの神 (Deus ex Machina)」的で, Gödel の言う様に「偽装された集合論」的な側面を持っています. 猶, Gödel の不完全性定理の証明では, 「数学原理」の公理系の中で還元可能性公理を用いずに, 次の「集合の内包公理」を採用しています.

—— 集合の内包公理 ——

$$\exists u[\forall v(u(v) \equiv a)]$$

ここで, v は n 階の変数, u は $n+1$ 階の述語, a は u を含まない $n+1$ 階の命題になります. 猶, 「集合の内包公理」は集合を確定する述語の存在を保証する為, 「還元可能性公理」に包含される公理になります⁴¹.

Russell は彼の論理学を Frege の概念記法ではなく, Peano 流儀の線的な表記を用いています. 次に幾つかの定義を示しておきましょう:

1. 命題関数: 命題関数は変項が x の場合は φx , 変項が $x, y, z, \dots$ であれば, $\varphi(x, y, z, \dots)$ と表記
2. 命題 p の否定は $\sim p$ と表記
3. 二つの命題の論理和は $p \vee q$ と表記
4. 任意の変項に対して命題関数 φ が真の場合, 不定値の x を用いて φx と表記
5. 全ての変項に対して命題関数 φ が真の場合, $(x).\varphi x$ と表記
6. 任意の階の述語を $\varphi!x$ と表記
7. 言明は $\vdash$ で表記

次に論理式で用いる幾つかの記号の定義を, 現在の表記と比較してみましょう:

⁴¹[34] では「還元可能性公理」ではなく, この集合の内包公理を用いています

Russell

$$\text{Df.} \quad \Leftrightarrow \quad (p \rightarrow q) =_{def} (\neg p \vee q)$$
$$\text{Df.} \quad \Leftrightarrow \quad p \wedge q =_{def} f \neg(\neg p \vee \neg q)$$
$$\text{Df.} \quad \Leftrightarrow \quad p \equiv q =_{def} (p \rightarrow q) \wedge (q \rightarrow p)$$
$$\text{Df.} \quad \Leftrightarrow \quad \exists x[\varphi x] =_{def} \neg[\forall x(\neg(\varphi x))]$$
$$\text{Df.} \quad \Leftrightarrow \quad x = y =_{def} \forall \varphi (\varphi!x \supset \varphi!y)$$

この公理系を次に示しておきましょう:

1. 命題 $p \supset q$ は q が真ならば真である
2. $\vdash: p \vee p. \supset p.$
3. $\vdash: q. \supset .p \vee q.$
4. $\vdash: p \vee q. \supset .q \vee p$
5. $\vdash: p \vee (q \vee r). \supset .q \vee (p \vee r)$
6. $\vdash: q \supset r. \supset: p \vee q. \supset .p \vee r$
7. $\vdash: (x).\varphi x. \supset .\varphi y$
8. $\vdash: \varphi y. \supset .(x).\varphi x$
9. $\vdash: (x).\varphi x. \supset .\varphi a$
10. $\vdash: .(x).p \vee \varphi x. \supset . \vee .(x).\varphi x$
11. $f(\varphi x)$ が任意の変項 x に対して真であり, 且つ, $F(\varphi y)$ が任意の変項 y に対して真であれば, $\{f(\varphi x).F(\varphi x)\}$ は任意の変項 x に対して真である

12. 任意の可能な変項 x に対して $\varphi x. \varphi x \supset \phi x$ であれば, 任意の可能な変項 x に対して ϕx は真である

13. $\vdash : .(\exists f) : .(x) : \varphi x. \equiv .f!x.$

14. $\vdash : .(\exists f) : .(x, y) : \varphi(x, y). \equiv .f!(x, y)$

ここで, 13 と 14 が「還元可能性公理」になります.

さて, Russell は「還元可能性公理」に訴える事で, 命題函数の値域を次の様に定めています.

—— 命題函数の外延 ——

$f\{\hat{z}\phi(z)\}. =: (\exists \varphi) : \varphi!x. \equiv_x .\phi x : f\{\varphi!\hat{z}\} \quad \text{Df.}$

$f\{\hat{x}\hat{y}\phi(x, y)\}. =: (\exists \varphi) : \varphi!(x, y). \equiv_{x, y} .\phi(x, y) : f\{\varphi!(\hat{x}, \hat{y})\} \quad \text{Df.}$

この様に命題函数 ϕ の外延 $\hat{z}(\phi z)$ は還元可能性公理が存在を保証する命題函数 ϕ と同値な述語 φ の外延として定義します.

又, 対象 x が外延 $\hat{z}(\phi z)$ に帰属する事も同様に定めます:

—— 対象の帰属 ——

$x \in \hat{z}(\phi!z). = .\varphi!x \quad \text{Df.}$

この定義の意味は, 個体 a が命題函数 ϕ の外延に所属する事は, ϕ と同値な述語函数 φ に対し, $\varphi!a$ が真となる事であると言う事です. これは Frege の $\Delta \cap p$ の定義と似たものですが, ここでも還元可能性公理が介在しています.

更に, 一般の命題函数 ϕ の外延 $\hat{z}(\phi(z))$ に対象 x が帰属する事は,

$$x \in \hat{z}(\phi z). =: (\exists \varphi) : \varphi!y. \equiv_y .\phi y : \varphi!x$$

と還元可能性公理から, $\vdash : x \in \hat{z}(\phi z). \equiv .\phi x.$ が得られます.

ここで, 外延に関する記号の定義をしておきましょう:

—— 外延に関する定義 ——

$cls = \hat{\alpha}\{(\exists\varphi).\alpha = \hat{z}(\varphi!z)\}$	Df.
$Kl = \hat{\alpha}\{(\exists\varphi).\alpha = \hat{z}(\varphi!z.Indivi!z)\}$	Df.
$\alpha \subset \beta. =: x \varepsilon \alpha. \supset_x .x \varepsilon \beta$	Df.
$\exists!\alpha. = .(\exists x).x \varepsilon \alpha$	Df.
$V = \hat{x}(x = x)$	Df.
$\Lambda = \hat{x}\{\sim (x = x)\}$	Df.

記号 cls でクラスを定義しています。これは命題が定める外延の外延です。そして、記号 Kl はクラス cls に包含される個体のクラスになります。

記号“ $\subset$ ”は現在の集合論で用いられる記号“ $\subset$ ”と同じ意味で、外延の包含関係を示します。次の記号“ $\exists!$ ”も現在の数理論理学で用いられる記号“ $\exists!$ ”と同じ意味で、命題関数 ϕx を満す個体が一つのみ存在する事を示します。次の V と Λ は真理値の表現で、各々 Frege の true ($\hat{\varepsilon}(\_ \varepsilon = \varepsilon)$) と false ($\hat{\varepsilon}(\_ \neg \varepsilon = \varepsilon)$) を思い出させるものです。

4.12.5 Hilbert による形式化

Hilbert は既に 1899 年の「幾何学基礎論」[25] で Euclid 幾何学の公理化を行っています。この Euclid 幾何学の公理化に関連した有名な逸話が、「点、線、面でなく、机、椅子、ビールジョッキと言い換えても幾何学が出来る」と Hilbert が語ったというものです。要するに、形式化した体系として幾何学を再構成しても、その本質は残っているという事で、今風に言うならば、点、線や面の意味を知らない計算機でも、形式的な手続を処理して行く事で幾何学が出来ると言い換えても良いでしょう。

やがて、彼の計画に沿って最適と思われる Russell の数学原論 (Principia Mathematica) の考えを取り入れて、命題論理式や述語論理式の形式化を行っています。

ここでは Hilbert の命題の形式化を簡単に眺めてみましょう。先ず、準備として以下の記号や言葉を導入します。

- (1) 命題記号 (基本命題) $a, b, c, \dots$
- (2) 論理記号 $\neg, \rightarrow, \leftrightarrow, \vee, \wedge$
- (3) 命題論理式 (命題) $A, B, C, \dots$

猶、Hilbert・Bernays の数学の基礎 [26] の論理記号は $\neg, \&, \vee, \rightarrow, \sim$ ですが、一部記号を置換えています。

最初の命題記号は、「ミケは猫である」、「 $2 = 1 + 1$ 」や「 $1 < -1$ 」といった変項を含まない命題を表現する記号です。

次の論理記号は論理式の演算子と言えるものです. 実質的な論理記号は, $\neg$ と $\rightarrow$ で, 他の $\leftrightarrow, \vee$ や $\wedge$ は $\neg$ と $\rightarrow$ を用いて表現する事が可能です.

先ず, 最初の論理記号 $\neg$ は否定記号で, 命題論理式 P に対して $\neg P$ とする事で, 命題論理式 P の否定した命題論理式を生成します.

論理記号 $\rightarrow$ は条件と呼ばれ, 命題論理式 $A \rightarrow B$ の意味は「 A ならば B 」に対応します. 猶 $A \rightarrow B$ は通常の「 A ならば B 」の感覚とは異なったもので, $A \rightarrow B$ は A が真で B が偽の場合のみに偽になる論理式です.

そして, 論理記号 $\leftrightarrow$ は, $A \leftrightarrow B$ の様に用いられ, その意味は「 $A \rightarrow B$ かつ $B \rightarrow A$ 」に対応し, 双方の命題が同値である事を意味します.

論理記号 $\vee$ は選言記号と呼ばれ, 命題論理式 $A \vee B$ の意味は「 A または B 」に対応します. 論理式 $A \vee B$ は $\neg A \rightarrow B$ で定義されたものです.

論理記号 $\wedge$ は連言記号と呼ばれ, $A \wedge B$ の意味は「 A かつ B 」に対応します. 論理式 $A \wedge B$ は $\neg(A \rightarrow \neg B)$ で定義されたものです.

さて, これらの用語を用いて, Hilbert は命題論理式を以下の様に帰納的に定義しています.

- (1) 命題記号は命題論理式である.
- (2) A, B を命題論理式とすると, $\neg A, A \wedge B, A \vee B, A \rightarrow B, A \leftrightarrow B$ は命題論理式である.
- (3) (1), (2) で構成されたもののみが命題論理である.

次に, 命題論理式に対しては, 以下の公理系を持たせています.

— Hilbert & Ackermann の公理系 —

公理 1	$A \rightarrow (B \rightarrow A)$
公理 2	$(A \rightarrow B) \rightarrow ((A \rightarrow (B \rightarrow C)) \rightarrow (A \rightarrow C))$
公理 3	$A \rightarrow (B \rightarrow A \wedge B)$
公理 4	$A \wedge B \rightarrow A$
公理 4'	$A \wedge B \rightarrow B$
公理 5	$A \rightarrow A \vee B$
公理 5'	$B \rightarrow A \vee B$
公理 6	$(A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow (A \vee B \rightarrow C))$
公理 7	$(A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A)$
公理 8	$\neg\neg A \rightarrow A$
推論規則 1	二つの論理命題式 $A, A \rightarrow B$ から B が推論出来る

ここで, 公理 8 は二重否定の除去になり, 最後の推論規則 1. は先程の Frege の推論規則にも出てきた前提肯定です.

さて、この公理系で証明は以下の様に定義されます。

1. 公理は証明可能である
2. 証明可能な命題論理式に推論規則 1 を適用して得られる命題論理は証明可能である
3. 1., 2. から得られた命題論理式のみが証明可能である

そして、命題論理式 A が公理からのみ証明可能な場合、 $\vdash A$ と表記します。又、命題論理式 $B_1, \dots, B_n$ から証明可能な場合には、 $B_1, \dots, B_n \vdash A$ と記述します。そして、証明は樹状の階層図として書かれます。

例えば、 $B_1, \dots, B_n \vdash A$ であれば、

$$\frac{B_1, \dots, B_n \quad B_1 \wedge \dots \wedge B_n \rightarrow A}{A}$$

と表記します。

4.12.6 述語論理

次に、述語論理式の定義を行います。その前に幾つかの記号や用語を導入しておきましょう。

1. 対象記号 $c_1, c_2, c_3, c_4, c_5 \dots$
2. 関数記号 $f_1, f_2, f_3, f_4, f_5 \dots$
3. 述語記号 $P_1, P_2, P_3, P_4, P_5 \dots$
4. 自由変項 $a_1, a_2, a_3, a_4, a_5 \dots$
5. 束縛変項 $x_1, x_2, x_3, x_4, x_5, \dots$
6. 論理記号 $\neg, \vee, \wedge, \rightarrow, \forall, \exists$

これらの記号に対し、項を以下の様に定義します。

1. 対象記号と自由変項は項である。
2. $t_1, \dots, t_n$ が項で f が関数記号であれば、 $f(t_1, \dots, t_n)$ も項である。
3. 上記の 1., 2. で得られたものだけが項である。

そして、次に述語論理式の定義に移ります。

1. $t_1, \dots, t_n$ が項で P が述語記号であれば、 $P(t_1, \dots, t_n)$ は論理式である。

2. 二つの論理式 A, B に対し, $\neg A, A \vee B, A \wedge B, A \rightarrow B$ は論理式である.
3. 自由変項 a を持つ論理式 $A(a)$ に対し, 変項 x が論理式 $A(a)$ 中に出現しない変項であれば, $A(a)$ 中の a の出現を x で置換えた論理式を $A(x)$ とすると, $\forall x A(x)$ と $\exists x A(x)$ は論理式になる.
4. 上の 1., 2., 3. で構成されたもののみが論理式である.

そして述語論理式の公理として, 命題論理式の公理系 1. から 8. に次の推論規則を追加します.

——— 述語論理式の公理 ———

公理 9. $A(x) \rightarrow \exists x A(x)$

公理 10. $\forall x A(x) \rightarrow A(t)$

ここで公理 10. の t は命題論理式 $A(x)$ の中に出現しない任意の項です.

ここで言葉の解説をしておきましょう. 先ず, 述語の変項には自由変項と束縛変項の二種類があります. 束縛変項は全称記号 $\forall$, 存在記号 $\exists$ という二つの量子子で束縛された変項の事です. 例えば, 述語 $\forall x. P(x, a)$ が与えられた場合, 変項 x が束縛変項, 変項 a の事を自由変項となります.

全称記号 $\forall$ は $\forall x. P(x)$ で「全ての x に対して $P(x)$ 」と読みます. 又, 存在記号 $\exists$ は $\exists x. P(x)$ で「ある x に対して $P(x)$ 」と読みます. 全称記号と存在記号の双方を合せて, 量子子と呼びます. この全称記号と存在記号には, $\neg \forall x. P(x) \leftrightarrow \exists x. \neg P(x)$ の関係があります. この量子子には直後の変項が影響を及ぼす範囲があります. これを作用範囲 (scope) と呼びます. 例えば, 述語 $\forall x. \exists y. P(x, y) \vee Q(y)$ が与えられた場合, $\forall x$ の作用範囲は $\exists y. P(x, y) \vee Q(y)$ で, $\exists y$ の作用範囲は $P(x, y)$ になります.

述語には α 同値と呼ばれる同値関係があります. これは変項の入れ換えに関する同値関係です. これは, 述語 $P(x)$ が与えられた時, 述語 $P(x)$ に含まれない自由変項 y を用いて変項 x を変項 y で置換える事で得られた述語 $P(y)$ とした場合, $P(x)$ と $P(y)$ を α 同値と呼びます.

4.12.7 標準形について

任意の論理式は量子記号と $\neg, \vee, \wedge$ のみの式に変換する事が可能です. 実際, $P \rightarrow Q$ は $\neg P \vee Q$ に置換えられ, $P \leftrightarrow Q$ はそもそも $(P \rightarrow Q) \wedge (Q \rightarrow P)$ と同値なので, $(\neg P \vee Q) \wedge (\neg Q \vee P)$ に置換えられるからです.

その結果, 任意の論理式は $F_1 \vee \cdots \vee F_m$ や $G_1 \wedge \cdots G_n$ の形の式に変換する事が可能です.

述語 P の自由変項を $x_1, \dots, x_n$ とする時, これらの自由変項を全称量子化子 $\forall$ や存在量子化子 $\exists$ を用いて束縛する事が可能です:

閉形式

全称閉形式 $\forall x_1, \forall x_2, \dots, \forall x_n P$

存在閉形式 $\exists x_1, \exists x_2, \dots, \exists x_n P$

任意の閉形式に対しては, 量子化子を先頭に置いた次に示す標準形と呼ばれる式に変換する事が可能です:

- $Q_1 x_1 Q_2 x_2 \cdots Q_m x_m [(A_{11} \vee \cdots \vee A_{1k_1}) \wedge \cdots (A_{h1} \vee \cdots \vee A_{hk_h})]$
- $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n [(B_{11} \wedge \cdots \wedge B_{1l_1}) \vee \cdots (B_{m1} \wedge \cdots \wedge B_{ml_m})]$

ここで, A_{ij} は $A_{ij1} \wedge A_{ij2} \wedge \cdots \wedge A_{ijm}$ の様に論理演算子 $\wedge$ のみか論理演算子を持たない述語, B_{ij} は $B_{ij1} \vee B_{ij2} \vee \cdots \vee B_{ijn}$ の様に論理演算子 $\vee$ のみか, 論理演算子を持たない述語です.

この変換手順を以下に示しておきましょう:

標準形変換の手順

1. 量子化子 Qx の作用範囲内に変項 x が出現しない場合, 量子化子を削除する
2. 束縛変項の取り換え式の左から順番に行い, 全ての束縛変項に重複が無い様にする
3. 同値な論理式に置換する事で, 論理演算子を $\vee, \wedge, \neg$ のみにする
4. 量子化子の外にある $\neg$ は量子化子の作用範囲内に入れる
5. 量子化子を式の左側に移動させる
6. $\vee$ に対して $\wedge$ を分配, 或いは $\vee$ に対して $\wedge$ を分配する

4.12.8 公理的集合論

さて, Cantor の素朴集合論も形式主義に基いて幾つかの公理系を持つもの (公理的集合論) として再構築されました. ここでは集合の公理系として有名な ZFC-公理系⁴²についてその概要を述べます. ここで用いられている記号の意味は §4.12.5, ZFC-公理系の詳細は田中 [21] を参照して下さい.

—— ZFC-公理系 (ZF-公理系+選択公理) ——

外延公理	$\forall z(z \in x \leftrightarrow z \in y) \rightarrow x = y$
対公理	$\exists z \forall u(u \in z \leftrightarrow u = x \vee z \in y)$
和集合公理	$\exists y \forall z[z \in y \leftrightarrow \exists u(u \in x \wedge z \in u)]$
冪集合公理	$\exists y \forall z(z \in y \leftrightarrow z \subseteq x)$
空集合公理	$\exists x \forall y \neg(y \in x)$
無限集合公理	$\exists x[\emptyset \in x \wedge \forall y(y \in x \rightarrow y \cup \{y\} \in x)]$
正則性公理	$x \neq \emptyset \rightarrow \exists y(y \in x \wedge y \cap x = \emptyset)$
置換の推論図式	変項 v を含まない任意の述語論理式 $\varphi(x, y)$ に対し, $\frac{\forall x \forall y \forall z[\varphi(x, y) \wedge \varphi(x, z) \rightarrow y = z]}{\exists v \forall y[y \in v \leftrightarrow \exists x(x \in u \wedge \varphi(x, y))]}$
選択公理	$\forall x \in u(x \neq \emptyset) \vee \forall x, y \in u(x \neq y \rightarrow x \cap y = \emptyset) \rightarrow \exists v \forall x \in u \exists! t(t \in x \vee t \in v)$

外延公理は二つの集合 x, y の全ての元が一致する事と $x = y$ である事が同値な事, 即ち, 集合はその構成要素で決定される事を意味します.

対公理は, 集合 x, y が与えられた場合, これらを成分に持つ集合 $\{x, y\}$ が存在する事を意味します. 猶, この集合の一意性は外延公理によって保証されます. 猶, 集合 $\{x, y\}$ の元に順序は入りませんが, $\{x, \{x, y\}\}$ という集合を考え, $\langle x, y \rangle$ と表記すると, x と $\{x, y\}$ の間には包含関係が入る為, $\langle x, y \rangle$ の元の x と y に順序を入れる事が可能です. この集合 $\langle x, y \rangle$ の事を対集合と呼びます. この対集合に関しては, $\langle a, b \rangle = \langle c, d \rangle \leftrightarrow a = c, b = d$ が成立します. 又, a を n 階の対象, b を $n+1$ 階の対象とする場合, $\langle a, b \rangle$ は $\langle \{a\}, b \rangle$ とする事で定義が可能になります. この事を用いると, $\langle a, b, c \dots \rangle$ も同様に定義する事が可能です.

和集合公理は, 集合 x, y の和集合 $x \cup y$ の存在を保証します.

冪集合公理は, 集合 x の冪集合 $\mathfrak{P}(x)$ の存在を保証します.

空集合公理は, 何らの元を含まない空集合 $\emptyset$ が存在する事を保証します.

⁴²ZFC=Zermelo-Fränkel+axiom of Choice

無限集合公理は、 $\emptyset$ を含む集合 x の元 y に対し、 $y \cup \{y\}$ を含む集合の存在を保証するものです。この $y \cup \{y\}$ は y の後継とも呼びます。猶、この公理により、順序数の定義で用いた $\{\emptyset, \{\emptyset, \{\emptyset\}\}, \dots\}$ といった集合の存在も保証されます。

正則性公理は、空集合 $\emptyset$ と異なる集合に関して、それ自身と共通要素を持たない要素を含む事を主張します。例えば、 $a = \{a\}$ の様な集合や、 $b_0 = \{b_1\}, b_1 = \{b_2\}, \dots, b_n = \{b_{n+1}\}, \dots$ の様に無限列 $\{b_0, b_1, \dots\}$ の存在を防ぐ公理になっています。もし、このような集合があった場合、 $a_0 \ni a_1 \ni \dots$ の様な底無しの集合が存在する事になって都合が悪いからです。

より簡単な例として、高木 [19] に出ている Finsler の考案によるものを考えてみましょう。

1,2,3,4,5,

この黒枠の中に表示されていない最小の自然数。

さて、上の黒枠の中に直接書かれていない最小の自然数は6ですね。ところが、この6という自然数は”この黒枠の中に表示されていない最小の自然数”として表現されていますね。従って次の自然数の7が最小の自然数になりそうです。しかし、そうすると7が今度は表現される事になりますね。この様に際限が無くなってしまいます。正則性公理は、この様な面倒な命題が発生する集合を認めません。

置換の推論図式は $\forall x \forall y \forall z [\varphi(x, y) \wedge \varphi(x, z) \rightarrow y = z]$ を仮定とし、この仮定の下で、 $\exists v \forall y [y \in v \leftrightarrow \exists x (x \in u \wedge \varphi(x, y))]$ が成立するというものです。

まず、置換の推論図式の仮定から $\varphi(x, y)$ は $F(x) = y$ の様な x の一価関数と看做す事が出来ます。そして、この仮定から得られる結論は、 $y \in v$ に対して、 $F(x) = y$ を満す x で構成される u も集合であると主張するものです。

最後の選択公理は空集合 $\emptyset$ でない集合から元を一つ取出す事が出来るというもの（選択関数の存在）です。この公理はユークリッド幾何学での平行線の公理に幾分似たものです。因に、選択公理の話だけでも一つの本が出来る程の深い話になります。

猶、この選択公理を使って証明出来る逆理で有名なものに Banach-Tarski の逆理があります。

— Banach-Tarski の逆理 —

3次元 Euclid 空間 $\mathbb{R}^3$ において, A, B を内転を持つ任意の有界集合とする. この時, A, B を適当な同数個

$$\begin{cases} A = A_1 \cup A_2 \cup \cdots \cup A_n \\ B = B_1 \cup B_2 \cup \cdots \cup B_n \end{cases}$$

に分割し, 各 A_i と $B_i (1 \leq i \leq n)$ が合同になる様に出来る.

この逆理を適用すると, ゴルフボールの表面を適当に分割し, それらを貼り合せたもので地球が覆える事になります. 従って, 牛の皮であれば砦どころか, 世界征服も可能と, 女王 Dido も大喜びな話⁴³になります.

選択公理と同等なものとしては, 整列性定理と Zorn の補題が挙げられます. 整列性定理は, 任意の集合は, その元の間に適切な順序を定義すると整列集合, 即ち, 任意の空でない部分集合が最小限を持つ集合になるということです.

そして, Zorn の補題は順序を入れた集合に対して非常に有効な補題です. 先ず, 順序付けられた集合に対し, その部分集合の元が順序による上限を持てば, 必ず最大元が存在すると言うものです.

Zorn の補題では, 有限回で終了するかどうか分らない処理で, その処理には上限があり, 更に, 何らかの指標で単調な増大列が構成出来るのであれば, その処理を継続する事で, やがて具体的な値が得られる事を意味します. 又, その指標の程度で近似解が得られる事も保証されるのです.

4.13 帰納的函数

次の (I), (II), (III) の始函数を基に, (IV) と (V) の操作を繰り返して適用する事で構成される函数を原始帰納的函数と呼びます.

⁴³牛の皮で覆えるだけの土地が与えられるという条件で, 牛の皮を細かく切って取り囲んで得た場所から発展したというカルタゴの建国神話

始函数と合成操作

(I)	$f(x) = x'$	
(II)	$f(x_1, \dots, x_n) = q$	(q は定数)
(III)	$f(x_1, \dots, x_n) = x_i$	($1 \leq i \leq n$)
(IV)	$f(x_1, \dots, x_n) = g(h_1(x_1, \dots, x_n), \dots, h_m(x_1, \dots, x_n))$	
(V)	$\begin{cases} f(0, x_2, \dots, x_n) = g(x_2, \dots, x_n) \\ f(y', x_2, \dots, x_n) = h(y, f(y, x_2, \dots, x_n), x_2, \dots, x_n) \end{cases}$	

4.14 λ 計算

λ 式は, 函数 $f(x)$ という表記方法が, 函数 f の表記なのか, それとも x の値なのかを判別し難い為に導入された函数の記述方法です. この問題点は Frege も気付いて独自の表記法を開発していますが (§4.12.3 参照), この Frege とは独立して, Church が λ 式による函数表記を考案しています. Church は, この λ 式を用いて完全な数学の体系を目指しましたが, 数学の体系の構築には失敗します. ところが, この λ 式は, その扱い方の便利さで残ったものです.

さて, 函数 $f(x)$ が函数 f の表記なのか, x に於ける値かどうか不明瞭な事は, 通常の計算では特に問題にならないかもしれませんが. 実際, 数学で式を $f(x) = \sin(x^2 + 1)$ の様に書いている時には, この表記に問題があると言われても何が問題なのか困惑するかもしれませんね. ところが, 函数 f を一種の演算子の様に考えてしまうとどうでしょうか? 具体的には, Basic や Fortran でこの式を表現する場合, 直接 $f(x) = \sin(x**2+1)$ と記述しても, 変数 x が何処か値を割当てられていると, その $f(x)$ にはその値が入ってしまうので, 作用素としての f の表現には失敗していますね. プログラムの場合, サブルーチンやプロシジャといったもので表記する事になるでしょうか, その場合には, 引数を使って表現するでしょう. λ -表記はその方法をより体系的に行える手法です.

その方法は, 先ず, 函数の変数を式の先頭に明示的に取出し, その後に函数の本体を表記するものです. 例えば, $f(x, y) = x \cdot y$ の λ -表記は $\lambda xy. x \cdot y$ になります.

ここで, λ -表記された函数の事を λ 項と呼び, λ の直後にある変数の事を超変数と呼びます. 例えば, $f(x) = x^2 + a$ を λ -表記すると $\lambda x. (x^2 + a)$ となりますが, ここでの変数 a の様に λ 表記の超変数にならない変数の事を自由変数と呼びます. 逆に λ の後にある超変数に対応する本体側の変数の事を束縛変数と呼びます. そして, この束縛変数を函数本体に含まれていない変数で置換する事を α 変換と呼び, λ 項 M が λ 変換で λ 項 N に移る場合に, M と N を α 同値と呼び, $M =_\alpha N$ と表記します.

現在では λ -表記可能な関数の集合は一般帰納的関数と呼ばれるものや, Turing 機械と同じものである事が知られています. そして, 計算可能性と関連して, Church は以下の提唱を行っています.

Church の提唱

- 計算可能な関数は一般機帰納関数と看做す.

この Church の提唱は定理ではありませんが, 現在でも Church の提唱に反する事例は発見されていません.

4.15 Gödel とその後

宿敵 Brouwer を政治的に制圧した Hilbert の Hilbert 計画は思わぬ所で頓挫してしまいます. これは Gödel の不完全性定理によるものです. Gödel の不完全性定理には第一不完全性定理と第二不完全性定理の二種類があり, 第一不完全性定理では, 決定不能な命題の存在性を示し, 第二不完全性定理では, 論理体系の完全性はそれ自身で示す事が出来ないというものです. この Gödel の不完全性定理に関しては, Hilbert が用いた一階述語論理では成立するが, より高階の述語論理では成立しないではないのかという意見 (Zelmer) もある様です. Gödel も当初はその様な考えに同意していた様ですが, 最終的には高階でも不完全性定理が同様に成立すると考えていたそうです.

この決定不能な命題で有名なものに, 連続体仮説があります. これは自然数の基数 $\aleph_0$ よりも大きくて, 実数の基数 $\aleph$ よりも小さな基数が存在しないというものです.

従って, 数学の基礎を巡る論争に関しては, 形式主義も直観主義もどちらも当初の目的が達成出来ておらず, 最終的に勝者にも敗者にもなっていないと言えます.

ところで, 不完全性定理の言う所の決定不能な命題は, どちらかと言えば, 数学の辺境で生じている事で, 普通に数学を研究する上で形式主義の考えは問題が全く無い為に, 形式主義の考えが数学の主流となり様々な方面に影響を与えました.

例えば, 1930 年代のフランスの若手数学者グループ Bourbaki⁴⁴による数学原論は構成主義と呼ばれ, Hilbert の初期の形式主義を修正したものです.

この構成主義では, 集合論を基盤とした数学的構造を全面に出していますが, 1957 年のスプートニクの打ち上げを契機に西側諸国の教育界で吹き荒れた新数学 (New Math, 日本では「数学の現代化」) で, この流儀が大きく採用されています. 因に, この新数学はアメリカでソビエトに対抗する為の技術者育成を目的としたものですが, 集合, 論理, 群やトポロジーを大きく取り上げ, その一方で, Euclid 幾何学の比重が低下する結

⁴⁴ 創立メンバーは, Weil, Cartan, Chevalley 等

果を招く事になりました. この様子を文部省の中学校学習指導要領⁴⁵で眺めてみましょう. 先ず, 新数学以前の昭和 33 年度と新数学に対処した昭和 44 年度の数学の学習指導要領を比較すると, 従来は A:数,B:式,C:数量関係,D:計量,E:図形と分類されていたものが,A:数式,B:函数,C:図形,D:確率・統計,E:集合・論理と内容が一新されています. そして, 図形は双方にあるものの, 昭和 33 年度のものでは Hilbert 流の公理的ではない, 図形の科学⁴⁶とも言える Euclid 幾何学が主体であるのに対し, 昭和 44 年度のものでは解析幾何学的なものに置き代わっています. この様に新数学はあまりにも急進的なものであった為に, 現場での混乱を招き, 更には, 詰め込み教育への反発を起す事となって最終的には消えています. 実際, 昭和 52 年の指導要領を見ると, 最初に集合・論理が消え, それ以降は内容が減って行きます. ここで注意する事としては, Euclid 幾何学は従来の位置に戻る事無く, 比重が低下したままで, 数学全体の内容が削減されている事でしょう⁴⁷

猶, 高等学校の学習指導要領では, 昭和 45 年度で初めて電子計算機と流れ図, 線形計画法が出ています. 線形計画法は昭和 53 年度の学習指導要領からは消えていますが, 計算機は残り, 平成 10 年度の学習指導要領からは, 新たに情報科が増えているのは周知の事です.

4.16 そして計算機

さて, 数学の形式化は命題の意味を考える必要も無い, 非常に機械的な処理を行う事で, その命題の証明が行える事を意味するものです. この事は, 様々な数式の処理に留まらず, 最終的には数学の各種定理の計算機による自動証明の問題にも繋がります. 更に, より一般的には, 計算機の基礎としての数理論理学にも関連して行くのです.

例えば, MIT のテキストの「計算機プログラミングの構造と解釈」⁴⁸の表紙を見てみましょう. ユダヤ教のラビと思われる怪しい老人と若い弟子の間に λ が顕示しているという代物です. 集合論で $\aleph$ というヘブライ文字が用いられている事, λ -表記可能な函数が一般帰納的函数と同値である事, そして, 老人がコンパスと `eval` と `apply` の太極マーク風の小道具を手に行っている考えると, この秘教的な表紙の別の側面が見えてより一層楽しくなるでしょう.

さて, Maxima は MIT の MAC 計画から誕生した MACSYMA を母体になっています. この MACSYMA を記述する為に, MACLisp が開発された側面があり, この MACLisp

⁴⁵<http://www.nicer.go.jp/guideline/old/>

⁴⁶小平邦彦, 幾何学の誘い [15]

⁴⁷赤根也, 新講数学 I,II,III の様な, 異様に贅沢な高校生向けの参考書が消えて久しいですね.

⁴⁸<http://mitpress.mit.edu/sicp/full-text/book/book.html>

の後継が Common LISP になります. そして MACSYMA や LISP にも共に Hilbert 計画の名残があるのです.

第5章 Maxima の処理原理について

Faust

Mir hilft der Geist! Auf einmal seh ich Rat
Und schreibe getrost: Im Anfang war die
Tat!

ファウスト

我を助けよ, 精霊よ! 一瞥で我は悟り
確信をもちて書下す: 原初に業^{わざ}あり!

Göthe, Faust より

この章では, Maxima の述語, 文脈, 宣言と属性の設定, そして, 評価の方法について述べます. これらの処理に関連する事項として, 演算子と LISP に関連した函数についても簡単に述べます.

Maxima の大きな特徴の一つに, 利用者が必要な述語や規則を定義し, その述語や規則を式に適用して式の評価を行うといった処理手順を踏む点です. これは単純に式を代入して函数で処理を行う場合に意識する事は然程ないでしょう. しかし, 函数によっては式に含まれる成分に関する述語を付加したり, 処理に適した宣言を行う事で, より円滑に処理が行えるようになります.

更に, 規則を上手く利用する事で作用素や代数的性質を付加する事が容易になります. この際に, 作用素の定義で利用する Maxima の組込の演算子の特性を熟知していれば, より効果的な規則の適用が行えます.

5.1 Maxima の基礎概念

5.1.1 Maxima の原子

Maxima では数式や数式を処理する為の文が扱えます. これらの数式や文を対象と呼びます. ここで数式や文を構成する最下層の対象の事を原子 (atom) と呼びます.

原子は次の Maxima の対象を包含するものです:

- 記号 (symbol):
アルファベットや数値, 日本語の漢字, 平仮名や片仮名等の各言語の文字と alphabetic 属性を持った ASCII 文字の羅列で構成された対象.
- 文字列 (string):
"123" の様に二重引用符で任意の文字の羅列を括った対象.
- 整数 (fixnum, bignum):
128 の様に 0 から 9 迄の ASCII 文字の数字のみの羅列で構成された対象, 及び, この数字の羅列の先頭に符号として "+" や "-" を配置した対象.
- 浮動小数点数 (float):
12.8 の様に 0 から 9 までの ASCII 文字の数字と一つの小数点 "." で構成された対象, 及び, 1280e-1 の様に整数の末尾や 1.28e+2 の様に, 小数点 "." を一つ含む数字の羅列に e (整数次数) を付加した対象, 及び, これらの対象の先頭に符号として "+" や "-" を配置した対象.
- 多倍長浮動小数点数 (bigfloat)
1280b-1 の様に整数の末尾, 或いは, 1.28b+2 の様に小数点 "." を一つ含む ASCII 文字の数字の羅列に b (次数) を付加した対象. 及び, これらの対象の先頭に符号として "+" や "-" を配置した対象.
- 真理値 (boolean):
true と false, on と off

ここで有理数は Maxima の原子にはなりません. 何故なら, Maxima の有理数は整数の対で式として表現される為です.

以降, Maxima の記号, 文字列, 整数, 実数表現と真理値について, その概要を述べましょう.

Maxima の記号 (symbol)

Maxima の記号は非常に重要な対象です. 記号は Maxima の式の項となるだけでなく, 関数名, 演算子名や関数の変数や被演算子として利用する事が可能です.

この Maxima の記号は次の文字から構成されます:

- ASCII 文字のアルファベット
- 各言語の文字
- 0 から 9 迄の数字
- alphabetic 属性を持つ ASCII 文字

Maxima の文字は `\a` の様に先頭にバックスラッシュ `"\"`, 日本語キーボードでは円記号 `"¥"` を置いた ASCII 文字と各言語の文字を加えた対象です. ここでバックスラッシュ `"\"` を省略する事が可能な文字には, ASCII 文字のアルファベットと alphabetic 属性を持つ文字, 計算機環境に依存しますが, 各言語の文字があります. それ以外の文字を先頭に置く場合, バックスラッシュ `"\"` が外せませんが, 先頭以外に置く場合, 二重引用符や括弧の様に Maxima で特殊な意味を持つ文字以外ならば, バックスラッシュ `"\"` が省略可能です. 例えば, ASCII 文字の数字 0 は記号の先頭に置く場合, `\0ab` の様に記述しますが, 先頭でなければ `x0` の様にバックスラッシュ `"\"` が省略出来ます.

各言語の文字に関しては LISP と OS 等の利用環境に依存する為に一概には言えません. 但し, ここで前提としている OS (UNIX 系, MS-Windows) の日本語環境と CLISP であれば, 記号に日本語が使える筈です. 勿論, 他の計算機環境で利用する事を考慮して, 文字コード (EUC, JIS, SHIFT-JIS, UTF8 等) にも注意しなければなりません.

最後の alphabetic 属性は Maxima の文字に対し, `declare` 函数を用いて利用者が付与可能な属性です. ここで属性の概要は §5.1.4, 詳細は §5.4 を参照して下さい.

この alphabetic 属性を持った文字は先頭からバックスラッシュ `"\"` を外して Maxima の記号として扱う事が可能です. 更に, alphabetic 属性を持った文字は内部変数 `*alphabet*` に割当てられた LISP のリストの一成分になります (内部変数の概要は §5.1.8 参照). 因に `*alphabet*` の初期値として `(_ %)` が割当てられています.

猶, 演算子や Maxima で意味を持つ文字として既に利用されている ASCII 文字に対して alphabetic 属性を与える事は非常に危険な事です. ここで Maxima で意味を持たない文字とは, 何等の属性も与えられていない ASCII 文字と言い換える事が可能です. ここで初期状態で一切の属性を持たないアルファベット以外の ASCII 文字を示しておきます:

—— 属性が指定されていない ASCII 文字 ——

16 進表現	文字	注意事項
34	"	文字列の定義で利用
29	)	matchfix 型演算子 (と組合せて利用
2C	,	列で利用 (alphabetic 属性を絶対与えてはならない!)
5C	\	文字を意味する特殊記号
5D	]	matchfix 型演算子 [と組合せて利用
60	'	無問題
7E	~	外積演算として利用

この中で記号の先頭に置いた時にバックスラッシュ"\"を省略可能な文字は、文字"'"と文字"~"のみです。但し、文字"~"は初期状態で属性を持ちませんが、vect パッケージの外積演算子として用いられる為、その利用には注意が必要です。

Maxima の文字列

Maxima の文字列は任意の文字の羅列を二重引用符(")で括った対象です。因に、1,2,3 の様に文字", "で区切られた Maxima の対象を並べたものを列と呼びます。従って、「文字列」"123" と「文字の列」1,2,3 は別物です。

ここで文字列は Maxima の版によって実体が異なる事に注意が必要です。Maxima-5.13.0 以前は LISP の文字列型とは異った内部データで、大文字と小文字の区別がありませんが、Maxima-5.14.0 からは LISP の文字列型と一致します (§6.11, §6.4.1 参照)。文字列は、演算子の定義や演算子の処理で用いられませんが、この本では、演算子の表記は問題が無ければ二重引用符を外した表記も利用します。

文字列は記号の様に項として利用する事や、値を割当てする事も可能です。但し、このような利用方法は、プログラム言語一般でも正規的な利用方法ではないでしょう。

Maxima の整数と実数の表現

Maxima の整数には、通常の整数を表現する fixnum 型と可変桁の bignum 型の二種類があり、これらは LISP の整数表現と一致します。これらの整数の差異を Maxima で直接認識する事はありませんが、fixnum 型の上限は内部変数 fixnbound に割当てられた LISP の most-negative-fixnum の値で定められており、この値以上の整数は自動的に内部で bignum 型となります。

実数の表現では、浮動小数点数と多倍長浮動小数点数を用います。ここで、浮動小数点数は LISP の浮動小数点型 (float) で、上限は内部変数 flounbound で指定され、その値は LISP の most-negative-double-float 変数の値です。

多倍長浮動小数点数はリストと LISP の hash-table を用いて表現されており、内部データの書式がリストになりますが、有理数と異なり Maxima の原子として扱われます。

真理値

Maxima の真理値は true と false です。on と off もありますが、実体は true と false です。そして、LISP の T に true, NIL に false が対応します。そして、特別な意味を持つ記号として、unknown があります。こちらは述語の判断で「判定不能」である事を示す為に用いられる記号ですが、LISP の真理値ではありません。

5.1.2 有理数と複素数の表現

次に重要な対象である有理数と複素数を紹介しておきましょう:

- 有理数:
二つの整数の間に演算子"/"を置いた対象 (式)
- 複素数:
二つの数 a と b に Maxima の純虚数%i を和の演算子"+"と可換積の演算子"*"を用いて構築した式 $a + \%i * b$

有理数は Maxima 内部では演算子"/"の式として表現されています。そして、複素数も同様に Maxima の式として表現されています。因に、これらの対象に対応する有理数型や複素数型の対象が Common Lisp にありますが、Maxima ではこれらの型をどちらも利用していません (§6.4 参照)。その為、これらは Maxima の原子ではありません。但し、自動簡易化によって有理数や複素数が整数や実数に変換される事もあります。

5.1.3 Maxima の関数と演算子

Maxima の関数や演算子は、形式的に名前と変数、或いは被演算子を表現する記号から構成された対象です:

- Maxima の関数:
名前の直後に置いた括弧“()”の中に対象の列を置いた Maxima の対象
- 演算子:
演算子の型に従い、名前に対する被演算子の位置が定まった Maxima の対象。
関数の様に対象を括弧で囲む必然性は無い。

関数や演算子の名前として使える Maxima のデータ型は記号と文字列です。猶、演算子名は文字列で指定しますが、式中の演算子は文字列から二重引用符を除去した文字の羅列です。

関数や演算子の形式的な表記は、関数、或いは演算子の名前を指定する記号や文字と Maxima の対象で充当される事を示す記号の列で構成されます。猶、これらの記号は、関数の場合、変数、或いは変項、演算子の場合、被演算子と呼ばびます。

そして、形式的な関数単体を関数項、形式的な演算子単体を演算子項と呼ばびます。

例えば、対象 $\sin(x)$ が関数項で、この関数項の括弧“()”の前の記号 $\sin$ が関数名、括弧の中の x が関数項の変数です。

猶、演算子に対しては少し様子が異なります。例えば、形式的表記 $x+y$ に対し、文字列 $+$ が演算子名となり、この演算子が二項演算子属性を持つ為、 $x+y$ が演算子項となります。更に、この演算子項を構成する x と y が演算子 $+$ の被演算子となります (演算子の詳細は §5.3 参照)。

関数や演算子の評価

Maxima は入力した対象に対して自動的に評価を行います。この評価では、対象に含まれる関数や演算子等の属性、更には、関数や演算子固有の処理関数を用いて対象の処理が行われます (§5.8.1 参照)。

この時、属性を持たない対象は、精々、項順序 $>_m$ による並べ換えが行われる程度で、殆どそのままの対象が返される事になります。これは、未定義の関数の関数項、及び、実体の無い演算子の演算子項に対しても同様です。

この関数項や演算子項を処理させる為には、関数や演算子に属性を与えたり、その実体となる処理関数や Maxima に既存の簡易化関数が利用する規則等を定義しなければなりません。

函数や演算子の実体

演算子は各種の演算子属性を付与した函数の側面を持つ為、ここでは函数を中心に解説します。

さて,Maxima の函数は次の四種類に分類可能です:

- 形式的な函数
- Maxima 言語で記述した函数 (§7.1 参照)
- LISP で直接記述され, 利用者が直接利用可能な函数
- LISP で直接記述され,Maxima の裏で内部処理を受け持つ函数

形式的な函数とは, $f(x)$ のように Maxima の函数としての書式を持っているものの、その実体を持たない函数で、この種の函数に対する処理は、その函数に付与された属性による処理のみが実行されます。ここで函数の属性は `declare` 函数等で与えられます (§5.4 参照)。この場合、函数項の処理函数は属性の表現函数となります (§5.4.5 参照)。Maxima から直接利用可能な函数は最初の 3 個の函数です。これらの函数の事を Maxima の函数と呼び、最後の Maxima の裏で動作する函数の事を内部函数と呼びます。この函数は利用者にとっては直接見えない LISP の函数です。猶、Maxima の函数は内部函数を組合せて構成されたもので、Maxima に入力された式は一旦内部表現に変換され、内部表現の型や属性に対応する内部函数で処理されます (§6.6 等参照)。

ここで Maxima の函数と演算子は $((\text{函数名}) \text{変数}_1 \dots \text{変数}_n)$ の内部書式を持っています。但し、LISP の S 式と比べて第一成分の函数に括弧“()”が付いている点で異なります。この事は、内部的に Maxima の函数が他の対象とは一階上の対象である事を視覚的に示しているとも言えます (§6.4 参照)。ここで、演算子と函数の違いは内部表現の書式には直接現われません。演算子と函数の違いは、演算子が演算子としての属性を持っている事です。

ここで Maxima の函数は二つの状態があります。一つは名詞型と呼ばれるもので、この場合、函数項の評価を実行しません。もう一つは動詞型と呼ばれるもので、この場合は函数項の評価を実行します。内部的には名詞型は函数名を `f` とすると、`%f` が名詞型を表現し、`$f` が動詞型を表現します。猶、未定義の函数を入力した場合、Maxima は自動的に名詞型の函数が与えられたものとして Maxima は処理を行います。

最後に、利用者が実体を定義した函数と演算子は大域変数 `functions` に自動的にその名前が登録されます。

5.1.4 属性

属性とはものの性質の事です。この属性には変項が一つの述語が対応します。例えば、「 ξ は奇関数である」の様に具体的な項で充当すべき ξ がある論理式の事です。さて、この述語を「奇関数 (ξ)」としましょう。すると、関数 f が奇関数である事は「奇関数 (f)」が真となる事と同値ですが、この事を $[f, \text{奇関数}]$ と対で表現したものが、Maxima の属性の表現の基本的な考えになります。

因に、この手法は Russell の還元可能性公理 (Axiom of Reducibility): 「任意の命題に対して同値な述語が存在する」を思い出させるものです。この還元可能性公理は集合の内包性と密接に関連するもので、実際、この属性は対象がどのようなクラスに帰属するかを示す方法として有効な手法です。

猶、Maxima で属性を表現する為に、LISP の属性リストを利用する方法、この属性リストを模した Maxima のリスト構造を用いる方法や、関数形で表現する方法が内部で用いられています。

属性は Maxima の内部データとして表現されていますが、属性によっては、利用者が固有の大域変数に纏めた属性を参照したり、変更する事が可能となっています。

ここで対象に属性を付加する具体的な方法として、次の三つの方法があります:

- put 関数を用いて対象に属性を付加する
- declare 関数を用いて対象に属性を付加する
- 内部関数 defprop 関数¹等を用いて対象に属性を付加する

先ず、put 関数を用いる方法は、利用者が任意の対象に属性を与える事に適しています。こちらは最も一般的な方法です。

declare 関数や defprop 関数を用いる手法は、Maxima による式の変換と深く関係します。declare 関数を用いる事で、利用者が関数や演算子に線形性等の性質を付加する事が可能です。更に、演算子に対しては、演算子型や被演算子に対する演算子の束縛力の大きさの指定出来ます。

内部関数 defprop 関数を用いる方法は、利用者が LISP でパッケージを記述した時に主に用いる方法で、式を入力すると自動的に簡易化する内部関数を指定したり、微分公式等の様々な性質や $\text{\TeX}$ の書式に変換する際に用いる雛形が設定可能です (§5.4, §5.8.1, §9.9.1 参照)。

¹正確には LISP のマクロ

5.1.5 Maxima の式

Maxima の式は, 次の方法で構成された Maxima の対象です:

1. 記号, 数値と文字列
2. 1. を出力する関数
3. 1., 2. を演算子や関数で結合する事で得られたもの

1. で述べる様に, $x, 1, 4.123$ といった記号, 数値と文字列が Maxima を構成する最小の単位の式になります. 猶, Maxima の式を構成する記号の事を単純に, 変数, 或いは変項と呼びます.

更に, Maxima の対象が演算子” $:$ ”によって何らかの値を割当てられている場合, 束縛変数, 或いは束縛変項と呼び, 値が何も割当てられていない変数/変項の事を自由変数/自由変項と呼びます.

Maxima にはシステムや各関数を制御する特別な変数があります. この変数を Maxima の大域変数と呼び, 式を構成する対象である変数/変項と区別しておきます (§5.1.8 参照).

猶, 利用者が定義した束縛変数は自動的に大域変数 `values` に登録されます.

2. で述べる様に, `sin(x)` や `evenp(x)` の様な関数や利用者が定義した関数も Maxima の式になります.

最後の 3. は $1+2*3+\sin(x^2+1)/4$ の様な数学の式の事を基本的には意味しますが, 後の演算子で解説する様に, Maxima の演算子は非常に幅広いもので, `[1, 2, 3]` の様なリストも原子から演算子のコンマ” $,$ ”を用いて構築した列に `matchfix` 型の演算子”`[]`”を作用させてリストを生成したものとと言えます. この様に, Maxima の演算子, 特に, 二項演算子は Maxima の式を繋ぐ事で新たな式を生成する接着剤としての作用があります.

5.1.6 Maxima の式の内部表現

Maxima の式は表に見えている式と内部で処理されている式では異なった書式を持っています. Maxima 内部で処理される式の表現の事を内部表現と呼びますが, この内部表現は S 式風の前置式表現で, 式を構成する各項は項順序” $>_m$ ”で並び換えられており, 演算子項や関数項も演算子や関数の属性によって項順序” $>_m$ ”による並び換えが生じる事があります. そして, 関数項や演算子項の名前は”`()`”で括られ, 変項よりも一段高い対象である事が明瞭となる様に表現されています. 又, 変数項 (記号) は先頭に”`$`”が置かれた LISP の記号 (symbol) になります. (§6.4 参照).

5.1.7 Maxima の部分式と項

Maxima の式の部分式は、与を構成し、その単体で Maxima の式となり得る Maxima の対象の事です。更に、項は、和の演算子“+”や差の演算子“-”で区切られた部分式の事を呼びます。

例えば、式 $a+b*c+d$ の部分 $a+b*c$ は与式の部分式であり、 $a, b*c$ と d は与式を構成する項になります。但し、 $a+$ や $b*$ は Maxima の式とならない為、部分式でも項でもありません。

猶、この本では項が $\sin(x)$ の様に函数単体の場合には函数項と呼び、同様に演算子“+”と演算子“-”以外の単一の演算子のみで構成された項の事を演算子項と呼びます。

Maxima に式を入力すると、式の各項は Maxima の項順序“ $>_m$ ”で一意に並び換えられます (§5.2 参照)。その際に式の単純な簡易化 (代入や項の纏め) も式の型や大域変数の設定、函数や演算子の持つ属性に応じて実行されます (§6.6 参照)。

5.1.8 大域変数

Maxima にはシステムや函数の制御を行う特別な変数があり、その性質上、次の二種類の変数に分類されます：

- 大域変数: Maxima 側から利用者が直接利用可能な大域変数
- 内部変数: LISP 側の内部処理で専ら利用される大域変数

大域変数は、函数やシステムの制御を行う為に、利用者が必要に応じて演算子“:”で値を設定する事が可能であったり、大域変数名を入力する事でその値が参照可能な変数です。

内部変数は、システムが専ら利用する事を想定している為に表から見えない変数です。これらの変数は LISP の Special 変数が用いられています。猶、大域変数は Maxima の通常の変数と同様に内部では先頭に\$が付きます。内部変数の命名基準は変数名の先頭と末尾に*が置かれた名前 (慣習的な LISP の Special 変数の命名方法) を採用しています。猶、例外も幾つかありますが、先頭に\$が付く事だけは定義からありません。この内部変数の処理は基本的に LISP 側で処理する事になります (§3, §5.9 参照)。

5.1.9 述語

Maxima の述語は次の方法で構成された Maxima の論理式です:

1. true,false
2. 二つの Maxima の式を比較の二項演算子 ($=, >, <, >=, <=, \#$) で結合した論理式
3. equal 関数項,notequal 関数項のみの論理式
4. Maxima の真理関数を Maxima の式に作用させた論理式
5. 1.,2.,3.,4. の式に論理演算子 (and,or,not) を用いて構築した論理式

最初の 1. は真理値そのものです.

2. は $4 > 1$ や $x < 1$ の様に, 比較の二項演算子を用いる事で二項間の関係を示した論理式です. 猶, 比較の演算子を用いて構成された述語は Maxima に入力しても直ちに解釈はされません.is 関数,maybe 関数や ev 関数等の述語の判断を行う関数で, その真理値が返されます.

3. の equal 関数や notequal 関数は関係の演算子 $=$ と演算子 $\#$ に似た意味を持つ関数ですが, 演算子 $=$ や $\#$ と異なり,assume 関数 で使えます. これらの関数も入力と同時に述語の判断が実行されません.

4. の真理関数は, 文脈等を利用する事で, 被演算子として与えられた述語の判断を行い,true や false 等の真理値を返す Maxima の関数です. 猶, 論理学での真理関数は, 真理値集合から真理値集合への写像の事ですが, ここでの真理関数の定義域は真理値集合に限らない一般的な Maxima の対象で, 何かの判断を伴う関数の事を指しています. 例えば,atomp(x) の様な Maxima 組込の真理集合値を返す関数項,is(x>=0) の様に, 述語を引数とする述語の判断を行う関数項, 或いは利用者が構築した,Maxima の対象を引数とし, 返却値が真理値となる関数項です.

そして 5. は Maxima の真理関数の構成方法を述べたものです. 論理演算子 and, or,not は判断を伴う演算子です. 猶, 演算子 and と演算子 or は被演算子を文字通り字面で判断する為, 被演算子の解釈を的確に行う関数 (例えば ev 関数) と組合せて利用する必要があります.

Maxima の述語は, 文脈と呼ばれる述語の判断を行う上で利用される Maxima の機構を用いて評価が行われます 4. の真理関数を含む論理式の判断は利用者が必要に応じた文脈上で行われます.

5.1.10 文脈

文脈は Maxima の論理式の保管庫として用いられる対象です. 正確には, 文脈は Maxima の記号で, 内部的に `subc` 属性を持った対象です. この `subc` 属性によって, 他の文脈間の親子関係が規定され, 全体として木構造の階層構造が入ります. この `subc` 属性自体は LISP の属性リストで表現され, 子文脈名がその属性値として登録されています (§5.6 参照).

Maxima の述語は `assume` 関数によって, 文脈の `data` 属性の属性値として LISP の属性リストを使って登録されます. ここで文脈に登録可能な論理式は以下の方法で構成されたものに限定されます:

1. 二項間の大小関係 (`>`, `>=`, `<`, `<=`) を示す述語
2. `equal` 関数による二項間の同値性
3. `notequal` 関数による二項間の非同値性
4. 1., 2., 3. の述語の演算子 `not` による否定
5. 1., 2., 3., 4. の述語を演算子 `and` で繋いだもの

ここで文脈に登録された論理式を $P_1, \dots, P_n$ としましょう. ここで Maxima に論理式 P_0 が与えられた時, その論理式の評価は論理式 $P_0 \wedge P_1 \wedge \dots \wedge P_n$ で行います. 即ち, 文脈は変項を ξ とする述語 $\xi \wedge P_1 \wedge P_2 \wedge \dots \wedge P_n$ です. この事が演算子 `or` を含む述語を文脈が受入れられない理由になります. 猶, 述語の評価で用いられる推論は基本的に Modus Ponens(MP) です (§4.12.3, §5.6 参照).

5.1.11 規則

規則によって, 演算子や関数の変換を行わせる事が可能です. ここで規則とは, 或る「式の並び」が与えられた時に, 演算子や関数に対して簡易化関数が行うべき処理を定めたものです.

公式と規則の違いは, 或る対象の固有の属性として与える事が可能な対象が公式ですが, ある特定の Maxima の対象で構成された式に対し, 特有の変換を定めるものが規則となります.

具体的には, $\sin 2x \Rightarrow 2 \sin x \cos x$ の様な式の変換は `sin` 関数の性質として与える事が可能です. ところが, その逆の $2 \sin x \cos x \Rightarrow \sin 2x$ は関数 $f(x) = \sin x \cos x$ の属性として与えられなくもありませんが, 寧ろ, $\sin x \cos x$ の様な「式の並び」が出現した時

に, $\sin 2x$ に変換する事を定めた方が自然ですね. この様に, 公式は「対象の属性」に, 規則は「式の並び」に対して与えられます.

その為, 規則の定義では「式の並び」を定義し, その「式の並び」に対して規則を定める手順となります. ここで, 「式の並び」では決った定項だけではなく, 関数を扱う必要がある為, 様々な指定を行う必要があります (§5.7 参照).

5.1.12 式の自動簡易化

大域変数 `simp` が `true` であれば, Maxima は自動的に式の簡易化を行います.

ここで, 自動簡易化では和`+`, 可換積`*`, 商/以外の演算子 (以降, 簡単に演算子と略記します), 及び関数の属性を用います. この簡易化で用いられる内部関数の種類を次で纏めておきましょう:

—— 演算子と関数の自動簡易化で用いられる内部関数の種類 ——

一般的な性質	⇔	内部変数 <code>*oper-list</code> に登録された属性の表現関数
関数固有の性質	⇔	<code>operators</code> 属性に対応する内部関数

線形性の様な一般的な演算子や関数の性質は, `declare` 関数を用いて属性として, 関数や演算子付加させ, その属性を用いて与式を自動的に簡易化させる事が可能です. この仕組は, 与えた属性に沿って式を変換する関数 (属性の表現関数と呼びます) と属性のリストを成分とする内部変数`*opers-list` を Maxima が持っており, 内部関数 `simplify` で, 入力された式の演算子項 (演算子と被演算子で構成された部分式) や関数項 (全ての変数が充足された関数) に対し, その演算子や関数の持つ属性と`*opers-list`に登録された属性の照合を行い, 属性が一致すれば, その属性の表現関数を項に用させます (§5.4.5 参照).

関数固有の性質による簡易化は, 内部関数 `defprop` によって関数毎に `operators` 属性として与えられた内部関数を関数項に作用させる事で行います. 具体的には, 内部関数 `simplify` にて, 関数項に対し, その関数固有の簡易化関数を項に作用させて関数項の簡易化を実施しています (§5.8.1 参照).

猶, 演算子項や関数項だけではなく, より一般的な式の自動簡易化を行う方法に, `tellsimp` 関数や `tellsimpafter` 関数による規則を用いた方法があります (§5.7.6 参照).

これは内部関数 `simplify` に演算子や関数の処理関数を追加する方式と言えるものです. 但し, この処理を行う為には, 規則を適用する為の「式の並び」を予め定義しておく必要があります (§5.7 参照)

式の簡易化では文脈を Maxima は利用します. この文脈による評価では, 大小関係と同値性を用いて, 符号処理や MP による推論が行われます (§5.6 参照).

5.1.13 まとめ

Maxima は, 与えられた対象が持つ属性, 対象が置かれた文脈に蓄積された述語を前提に, 大域変数によって制御された Maxima の函数群を用いて処理を行います.

この処理は結局, 述語の処理に他ならず, その意味で Maxima は Hilbert 計画直系の子孫 (「ロボット式数学」[19]) の側面を強く持っています.

更に, この属性を活用する特徴によって, Maxima は非常に古い数式処理システムでありながら, 全体的に問題に対応する為の柔軟さが得られているのです.

5.2 順序

5.2.1 Maxima の変数順序

Maxima の記号の集合を A_m と表記し, そして, 集合 A_m に入る順序を $>_m$ と記述します. すると, この順序 $>_m$ はどのようなものでしょうか? この順序について解説しましょう.

まず, 順序 $>_m$ によると, アルファベットでは大文字が小文字よりも大きくなっていますが, 同じ大文字では Z が A よりも大きくなっており, 通常のアルファベットの順番と逆になっています.

そして, alphabetic 属性を与えた Maxima の文字は, アルファベットの大文字の Z よりも大きく, `ordergreat` 関数の最後の引数よりも小さくなります.

0 から 9 迄の数値に対しては $>_m$ は通常的大小関係 $>$ と同値です.

`ordergreat` 関数や `orderless` 関数で入れた順序に無関係な文字と記号に関しては, ASCII コードの大きいものの順位が高くなります. これは, 文字の大小関係で内部の LISP の `great` 関数を用いて判別する為です.

変数が一文字であれば $>_m$ による比較は分かり易いものです. しかし, 変数は通常, abc の様に複数の文字で構成されます. この複数の文字に対しても, 順序 $>_m$ で順序付ける事が可能です. この考え方を簡単に説明しましょう. 二つの変数 $x_1x_2\cdots x_n$ と $y_1y_2\cdots y_m$ が与えられた時, $n = m$ でなければ, 短い変数の側に空白を入れて等しい長さにしたものを考えます. この時, $i = 1, \dots, k-1$ では, $x_i = y_i$, k 番目の文字 x_k と y_k で初めて異なるとします. すると, 文字 x_k と y_k の順序は $>_m$ の順序で決める事が出来ます. そこで, 全体もその順序に従うものとしましょう. すると, $x_k >_m y_k$ であれば, $x_1x_2\cdots x_n >_m y_1y_2\cdots y_m$ となります.

具体的には, 二つの変数 abc と $aaaz$ が与えられた場合, 頭は同じ文字 a なので二番目以降の文字で大小関係を定めます. この例では二番目の文字に対し, $b >_m a$ となるので $abc >_m aaaz$ となります.

最後に Maxima の項順序 $>_m$ で最も順位が高い変数が主変数 (mainvar) として `declare` 関数を用いて宣言された変数です. CRE 表現 (§6.2.2 参照) への変換等の多くの関数で主変数が用いられています. 与式中にて mainvar として宣言された変数が存在しなければ, 式中の変数で順序 $>_m$ による最高位の変数が (局所的な) 主変数として用いられます.

この Maxima の変数順序を次に纏めておきましょう:

Maxima の変数順序 $>_m$					
宣言された主変数	$>_m$	ordergreat の引数 ₁	$>_m$	...	$>_m$
ordergreat の引数 _h	$>_m$	alphabetic 宣言文字 ₁	$>_m$	...	$>_m$
alphabetic 宣言文字 _k	$>_m$	頭文字が Z の変数	$>_m$	...	$>_m$
頭文字が A の変数	$>_m$	頭文字が z の変数	$>_m$	...	$>_m$
頭文字が a の変数	$>_m$	orderless の引数 _n	$>_m$	...	$>_m$
orderless の引数 ₁	$>_m$	宣言されたスカラー	$>_m$	宣言された定数	$>_m$
Maxima の定数	$>_m$	9	$>_m$	...	$>_m$
0					

順序 $>_m$ は全順序となる為、 $(A_m, >_m)$ は全順序集合になります.

5.2.2 項式項に対する順序

次に Maxima の式を構成する項に対する順序 $>_m$ について解説します. Maxima の式は演算子+や-で区切られた部分式に分解されます. この部分式の事を項と呼びます. 項に対しては, 変数順序 $>_m$ を自然に拡張した項順序 $>_m$ を Maxima は持っています. そして, 順序 $>_m$ は辞書式順序と呼ばれる順序です.

ここで変数順序 $>_m$ の項への拡張方法を述べます. その為, 最初に扱う式を多項式に限定して解説します.

先ず, Maxima の項を構成する変数を $>_m$ の順序に従って大きなものから並べて列を構成します. 例えば, 項の変数が x_1 から x_9 の場合, $x_9 >_m x_8 >_m \dots >_m x_1$ となるので, 変数の列 $x_9, \dots, x_1$ を得ます. この項内部での変数の置換は Maxima で自動的に実行されます.

さて, 二つの項が入力されると, 各項の変数は順序 $>_m$ で並び換えられます. 次に二つの変数の列から項の次数リストを構築します. 次数リストは二つの項を構成する全ての変数を順序 $>_m$ で変数を並び換え, それから項毎に, 変数に対応する次数を左から順に並べた整数のリストです. もし, 片方の項に無い変数があれば, 次数 0 を入れます. 例えば, 二つの項 $x_1 x_2^2 x_8^3$ と $x_1 x_2^2 x_3 x_9$ が与えられた場合, 二つの項を構成する変数 $x_9, \dots, x_1$ の順序に従って変数置換を行います. その結果, $x_1 x_2^2 x_8^3$ は, $x_8^3 x_2^2 x_1$ に, $x_1 x_2^2 x_3 x_9$ は $x_9 x_3 x_2^2 x_1$ になります. それから, 各項の変数の次数で変数を置換えますが, 一方の項のみに現れる変数があれば, その変数が現れていない項の該当変数は 0 とします. 先ず, 項 $x_8^3 x_2^2 x_1$ は x_9 と x_3 が抜けているので $x_9^0 x_8^3 x_3^0 x_2^2 x_1$ になり, $x_9 x_3 x_2^2 x_1$ は x_8 が無いので, $x_9 x_8^0 x_3 x_2^2 x_1$ となります. この様に項を正規化してから, 変数の次数を左から順に並べたリストを構築します. その結果, 5 成分の

次数リスト $(0, 3, 0, 2, 1)$ と $(1, 0, 1, 2, 1)$ が得られます. 次に, 項の大きさの比較では, これらのリストの先頭から通常的大小関係 $>$ を使って決めます. この場合, $(0, 3, 0, 2, 1)$ の先頭の値が 0 であるのに対し, $(1, 0, 1, 2, 1)$ の先頭が 1 となるので $(1, 0, 1, 2, 1)$ の方が大となります. この次数リストの大小関係をそのまま項の大小関係とします. この事から $x_1 x_2^2 x_3 x_9 >_m x_1 x_2^2 x_8^3$ となります.

この方法で構築した順序 $>_m$ は $x >_m y$ であれば, 任意の 0 と異なる項 a に対して, $ax >_m ay$ が成立する事が判ります. この性質を満す順序の事を項順序と呼びますが, この事から順序 $>_m$ が項順序である事が判ります.

猶, 大域変数 `simp` が true の場合, Maxima は項順序 $>_m$ に沿って式の項の並び換えを自動的に実行します:

```
(%i16) expr1:x1*x2^2*x8^3+x1*x2^2*x3*x9;
              2          2  3
(%o16)      x1 x2  x3 x9 + x1 x2  x8
(%i17) expr2:x1*x2^2*x3*x9+x1*x2^2*x8^3;
              2          2  3
(%o17)      x1 x2  x3 x9 + x1 x2  x8
(%i18) :lisp $expr1;
((MPLUS SIMP)((MTIMES SIMP) $X1 ((MEXPT SIMP) $X2 2)((MEXPT SIMP) $X8 3))
 ((MTIMES SIMP) $X1 ((MEXPT SIMP) $X2 2) $X3 $X9))
(%i18) :lisp $expr2;
((MPLUS SIMP)((MTIMES SIMP) $X1 ((MEXPT SIMP) $X2 2)((MEXPT SIMP) $X8 3))
 ((MTIMES SIMP) $X1 ((MEXPT SIMP) $X2 2) $X3 $X9))
(%i18) :lisp (equal $expr1 $expr2)
T
```

この例では, 同値な式の項の順番を変更して Maxima に入力していますが, 項の順序が変更されて同じ式で返されています.

更に, `:lisp $expr1` と `:lisp $expr2` で式 `expr1` と式 `expr2` の内部表現を表示させていますが, `expr1` と `expr2` の内部表現は同一です. 猶, `:lisp` は後に続く文字列を LISP に直接渡して結果を Maxima に戻す演算子です.

この内部表現で注目して頂きたいのは, 順序 $>_m$ で小さなものが, リストの左側に置かれ, 大きなものが右側に置かれる事です. この内部表現に対して, 式の表示は項順序の大きなものから左側に並べられます. こうする事で数式の通常書き方に準拠したものになっています.

5.2.3 局所的な順序の変更

Maxima の項順序 $>_m$ を局所的に変更する事が可能です. 項順序 $>_m$ を変更する函数としては `ordergreat` 函数と `orderless` 函数があります:

Maxima の順序変更を行なう函数

```
ordergreat(〈記号1〉, ..., 〈記号n〉)
orderless(〈記号1〉, ..., 〈記号n〉)
unorder()
```

`ordergreat` 函数により, 第一引数である記号 $\langle \text{記号}_1 \rangle$ が最大で, 以降, 第二引数, 第三引数...と右側に行くに従って小さくなり, 第 n 番目の記号 $\langle \text{記号}_n \rangle$ が最小となる様に新たな項順序 $>_m$ が再構築されます. `orderless` 函数の場合, 第一引数の $\langle \text{記号}_1 \rangle$ が最小で, 第二, 第三と右に行くに従って大きくなり, 第 n 番目の引数 $\langle \text{記号}_n \rangle$ が最大になる様に Maxima の順序 $>_m$ が再構築されます.

この `ordergreat` 函数や `orderless` 函数で Maxima に入れた順序は `unorder()` を実行すれば解除されます.

猶, `ordergreat` 函数や `orderless` 函数で再構築した順序 $>_m$ は, 一度 `unorder` 函数で元に戻さなければ, これらの函数による順序の再構築は出来ません:

```
(%i13) ordergreat(c,b);
(%o13)                                     done
(%i14) ordergreat(b,z);
Reordering is not allowed.
-- an error.  Quitting.  To debug this try debugmode(true);
(%i15) unorder();
(%o15)                                     [b, c]
```

この例では, 最初に $c >_m b$ という順序を入れています. それから, $b >_m z$ という順序を入れようとしてエラーが返されています. 最後に `unorder()` を実行して順序 $>_m$ を元に戻しています.

5.2.4 順序に関連する関数

Maxima の項順序は, 逆アルファベット順の辞書式順序と呼ばれる順序が入っています. そこで今度は Maxima で変数順序や項順序がどの様に入っているか, 実際に調べてみましょう. 二つの与えられた項の順序を調べる関数として Maxima には `ordergreatp` 関数と `orderlessp` 関数の二つの真理関数があります.

— 順序を確認する真理関数 —

```
ordergreatp(< 項1 >, < 項2 >)
orderlessp(< 項1 >, < 項2 >)
```

これらの関数は二つの引数を取る真理関数で, その真理値集合は $\{\text{true}, \text{false}\}$ です. これらの関数は内部で LISP の `great` 関数を用いています.

まず, `ordergreatp` 関数は, $\langle \text{項}_1 \rangle$ の筆頭項が $\langle \text{式}_2 \rangle$ の筆頭項よりも大となる場合に `true` を返し, それ以外の場合は `false` を返します.

次の `orderlessp` 関数は $\langle \text{式}_1 \rangle$ の筆頭項が $\langle \text{式}_2 \rangle$ の筆頭項よりも小となる場合に `true` を返し, その他の場合に `false` を返します.

```
(%i33) ordergreatp(abc,a);
(%o33)                                     true
(%i34) ordergreatp(abc,ax);
(%o34)                                     false
(%i35) ordergreatp(x^2,y^2);
(%o35)                                     false
(%i36) ordergreatp(z^2,y^2);
(%o36)                                     true
(%i37) ordergreatp(z,y^2);
(%o37)                                     true
(%i38) ordergreatp(z^3,z^2);
(%o38)                                     true
(%i39) ordergreatp(z^2*x*y^2,z^2*x*t^3);
(%o39)                                     true
```

最初の例では, 変数 `abc` と変数 `a` の順序を比較しています. Maxima の項順序 $>_m$ では, アルファベットに関しては逆アルファベット順で大きさが決り, 変数の比較では, 左端の文字から両者を比較し, 最初に大きなものがあつた方を大とします. この例では, `abc`

と a では頭は同じですが, a には他の文字が無い為, $abc >_m a$ となります. 次の, abc と ax の場合, 頭の a は同じでも, b と x では x の方が大となる為, $ax >_m abc$ となります. 次の x^2 と y^2 の比較では, 変数 y の方が x よりも大の為, y^2 の方が大になります. 同様に, z^2 と y^2 の比較では, 変数 z の方が y よりも大の為, z^2 が大になりますが, z と y^2 の比較でも, 次数とは無関係に z の方が y^2 よりも大になります. そして, 同じ変数の場合は次数の大きな方が大になり, $z^2 \times y^2$ と $z^2 \times t^3$ の場合は, 頭から比較した時に, $y >_m t$ となる事から, $z^2 \times y^2$ の方が $z^2 \times t^3$ よりも大になります. この事から, Maxima の変数順序は斉次ではなく, 辞書式順序に基く順序である事が理解出来るかと思えます.

Maxima では変数の順序を `ordergreat` 関数や `orderless` 関数で多少変更する事が可能ですが, 基本的な順序は辞書式順序のみです. SINGULAR の様な数式処理では, 複数の順序を目的に応じて選択する事が出来ませんが, Maxima は違います. この点から Maxima は古風な数式処理システムと言えるでしょう.

5.2.5 関数を含めた順序

Maxima では通常の多項式に加えて `exp` や `sin` 等の Maxima 組込の初等関数や利用者定義の関数に対しても順序を入れる事が可能です.

Maxima の組込の関数に関しては, 先ず, 同じ引数の二つの関数に対しては関数名で順序付けられます. 次に, 同じ関数を別の式に作用させる事で得られる二つの式に対しては, 関数を作用させる前の式に対して順序付ける事が出来ます.

組込の関数と多項式の項に対しては, 基本的に関数項の方が多項式項よりも大になりますが, 関数項に主変数が含まれず, 多項式項に主変数が含まれている場合, 多項式の項の方が順序 $>_m$ の上位になります.

利用者定義の関数の場合, Maxima 側で値を解釈する為に `ordergreatp` 関数や `orderlessp` 関数の値はその状況に応じて変化します.

以下にその例を示します.

```
(%i77) neko(x):=if x<0 then x^2 else cos(x)^3;
                                     2          3
(%o77)          neko(x) := if x < 0 then x  else cos (x)
(%i78) assume(p0>0);
(%o78)          [p0 > 0]
(%i79) ordergreatp(cos(p0),neko(p0));
(%o79)          false
```

```

(%i80) assume(p1<0);
(%o80) [p1 < 0]
(%i81) ordergreatp(cos(p1),neko(p1));
(%o81) true
(%i82) ordergreatp('neko(x),atan(x));
(%o82) true
(%i83) ordergreatp(neko(x),atan(x));
Maxima was unable to evaluate the predicate:
x < 0
#0: neko(x=x)
-- an error. Quitting. To debug this try debugmode(true);
(%i84)

```

この例で示す様に利用者函数に単引用符を付けなければ Maxima で解釈が実行され、その結果で ordergreatp 函数の結果が決ります。これに対して、単引用符を付けて名詞型で比較すると単純に函数名で比較されます。この様に初等函数や利用者定義函数が名詞型の場合は引数も含めた函数名で変数順序 $>_m$ による比較が実行されます。

5.3.1 Maxima の演算子について

5.3.1 Maxima の演算子について

Maxima の演算子には、数学で用いられる四則演算等を模した算術演算子、更には、変数に式を割当てゐる演算子等があります。ここで、演算子は利用者が新規に定義したり、既存の演算子の性質を変更する事も可能です。猶、新規に属性を定義する場合、演算子の名前として利用可能な Maxima の対象は、Maxima の記号、文字列に限定されます。Maxima の演算子は、演算子名と被演算子の関係、則ち、演算子の属性から次の 6 種類に分類可能です:

- **prefix 型** ⇔ 前置表現の演算子
微分演算子 " $\frac{d}{dx}$ " の様に一つの引数に対し、引数の前に配置される演算子
- **infix 型** ⇔ 内挿表現の演算子
等号の演算子 "=" の様に二つの引数の間に配置される演算子で、結合律を考慮する必要の無い演算子。
- **nary 型** ⇔ 内挿表現の演算子
和の演算子 "+" の様に二つの引数の間に配置される演算子で、結合律を考慮する必要のある演算子。
- **matchfix 型** ⇔ 外挿表現の演算子
[1,2,3] の "[]" の様に対象を囲む演算子
- **postfix 型** ⇔ 後置表現の演算子
階乗の演算子 "!" の様に演算子の後に引数を置く演算子
- **nofix 型** ⇔ 無引数の演算子
引数を取らない演算子

猶,Maxima の属性の詳細に関しては §5.4 を参照して下さい.
 それでは, 適当な文字列を演算子として宣言してみましょう.

[illegible]


```
(%i28) x :/ mike y;
(%o28)                                     x :/ mike y
```

この例では、函数としての実体が無い為に評価がこれ以上実行されません。演算子による処理を行う為には、演算子の実体を定義しなければなりません。この時、定義した演算子に実体を与える方法と定義した函数を演算子として宣言する方法の二種類がありますが、どの方法も結果は同じです。

そこで、先程の演算に実体を与えたり、函数を演算子にしてみましょう。

```
(%i29) mike x:=2*x+1;
(%o29)                                     mike x := 2 x + 1
(%i30) x :/ y := (x+sin(x))/y;
(%o30)                                     sin(x) + x
                                     x :/ y := -----
                                               y
(%i31) pochi(x,y):=x^y;
(%o31)                                     y
                                     pochi(x, y) := x
(%i32) nary("pochi");
(%o32)                                     pochi
(%i33) mike 3;
(%o33)                                     7
(%i34) 5 :/6;
(%o34)                                     sin(5) + 5
                                     -----
                                               6
(%i35) 4 pochi 2;
(%o35)                                     16
```

この例で示す様に、演算子として宣言した函数の実体を定義する場合、Maxima の函数の定義と同様に演算子”:=”を使います則ち、演算子”:=”の左辺にメタ変数を用いた演算子項を置き、右側にメタ変数の処理内容を函数の定義と同様に記述します。(§7.2 参照)。

この例から判る様に Maxima の演算子の実体は演算子属性を持った函数です。

5.3.2 演算子の束縛力

数学で用いる算術演算子には被演算子を引き止める力があります. 例えば, 数式 $1 + ab^2c - d$ が与えられると, 演算子と被演算子の関係から, $(1 + a(b^2)c) - d$ と解釈しています. Maxima では, 演算子の被演算子に対する引き込む作用を束縛力 bp(Binding Power) と呼び, 200 までの整数値で表現しています. 更に, 演算子と被演算子の配置関係から束縛力は二つの属性, 即ち, 左束縛力 lbp(Left Binding Power) と右束縛力 rbp(Right Binding Power) に分けて指定出来る演算子もあります.

束縛力は利用者が自由に設定する事が出来ます. Maxima の組込演算子の属性 lbp と rbp の値はファイル nparse.lisp で設定されています. 例えば, 和の演算子 "+" は両方共に 100, 可換積演算子 "*" は lbp のみ 120, 冪演算子は lbp が 140 で rbp が 139, 最後に差の演算子は lbp が 100 で rbp が 134 となっています. 左右の束縛力の値が大きなもの程, 演算子と被演算子の結び付きが強くなります. 例えば, 可換積が 120 ですが, 一方で冪は 140 となっています. これは冪の方が被演算子を引き付ける力が大きな事を意味し, それ故に b^2*c は $(b^2)*c$ と解釈される事になります. そして, 束縛力を指定しない場合, 初期値の 180 が設定されます. この事を実際に確認してみましょう.

```
(%i1) prefix("tama")$
(%i2) :lisp (get '$tama 'lbp);
NIL
(%i2) :lisp (get '$tama 'rbp);
180
```

この例では前置表現の演算子 tama を定義し, その左右の束縛力を取出しています. 束縛力の設定は LISP の属性を用いている為, LISP の get 関数で調べられます. この例では, 演算子 tama が前置表現の演算子の為, 左束縛力が未設定 (NIL) で, 右束縛力が 180 である事が判ります.

今度は後置式表現の演算子 mike を定義して束縛力を調べてみましょう.

```
(%i4) postfix("mike")$
(%i5) :lisp (get '$mike 'lbp);
180
(%i5) :lisp (get '$mike 'rbp);
NIL
```

この例から前置式演算子と束縛力の設定が逆になっている事が判ります.

この束縛力は演算子以外の右小括弧（や左小括弧）といった文字にも設定されています。ここで小括弧の束縛力は最大で 200 となっています。則ち、小括弧“()”は外や内部の束縛力を遮断する作用を持つ事になります。従って、処理に不安がある場合、小括弧で纏めれば演算子の影響は括弧の内外に及びません。

二項演算子に対しては左右の束縛力の釣合加減で結合律の成立に影響が出ます。

```
(%i5) infix("><",100,120)$
(%i6) (a >< b):=a^b;

                                     b
(%o6)          (a >< b) := a

(%i7) a><b><c;

                                     b c
(%o7)          (a )

(%i8) infix("><",120,100)$
(%i9) a><b><c;

                                     c
                                     b
(%o9)          a
```

この例では infix 型の内挿型演算子"><"を最初に左束縛力を 100, 右束縛力を 120 で定義しています。その為、 $a><c$ は右束縛力の方が強い為に、 $(a><b)><c$ と解釈されます。次に左束縛力を 120, 右束縛力を 100 と逆にした場合、演算子と左側の被演算子との繋がりが強くなった為に、 $a><c$ は $a><(b><c)$ で処理されます。この様に、結合律を満たす二項演算子を定義する場合は、infix 型を利用するのであれば左右の束縛力を同程度にするか、最初から nary 型で定義すべきです。

束縛力は内挿式、前置式演算子の実体の定義でも重要です。何故なら、定義で用いる演算子":="にも束縛力があるからです。演算子":="の束縛力は左が 180. 右が 20 となっています。その為、演算子の束縛力が弱い、即ち、演算子の右束縛力が 180 よりも小さい場合は、演算子":="の側に被演算子が引き寄せられる事になります。その為、演算子":="の左側の演算子項全体を小括弧で括らなければなりません。

この様に Maxima の式を記述する際に、少しでも式中の演算子の束縛力に疑問があれば、演算子の束縛力の作用範囲を明確にする為にも、小括弧“()”の併用を薦めます。

5.3.3 演算子の型

演算子の被演算子と処理結果には型があります. これらの型の指定は, 演算子の宣言を行う関数で行えます.

先ず, nary 関数と matchfix 関数で被演算子と返却値の型が設定される属性を示します:

- 被演算子の型が設定される属性: `argpos`
- 返却値の型が設定される属性: `pos`

nary 関数と matchfix 関数以外の関数では演算子の左右の被演算子を独立して設定する事が可能です. 次に左右の被演算子と返却値の型が指定される属性を示します:

- 左被演算子の型が設定される属性: `lpos` (left part of speech)
- 右被演算子の型が設定される属性: `rpos` (right part of speech)
- 返却値の型が設定される属性: `pos`

上記の属性に指定可能な型を次に示します:

演算子の型		
型	概要	属性値
<code>expr</code>	数式全般	<code>algebraic</code>
<code>clause</code>	論理式全般	<code>logical</code>
<code>any</code>	任意の Maxima の式	<code>untyped</code>

`expr` 型は Maxima の数式に対応する型です. `clause` 型は Maxima の述語や論理式に対応する型で, `any` 型は `expr` 型や `clause` 型を含む Maxima の任意の式となります.

更に, これらの `expr` 型, `clause` 型, `any` 型の属性として, `english` があり, その属性値は夫々 `algebraic`(代数的), `logical`(論理的), `untyped`(未分類) となっています.

```
(%i1) :lisp (get '$expr 'english);
algebraic
(%i1) :lisp (get '$clause 'english);
logical
(%i1) :lisp (get '$any 'english);
untyped
```

猶, 演算子宣言で演算子の型を指定しなければ, 左右の被演算子の型と返却値の型に

\$any が自動的に設定されます. 猶, これらの属性は LISP の属性を用いて実現されている為, LISP の put 関数で無理矢理に設定が出来ます.

```
(%i4) prefix("mike")$
(%i5) :lisp (get '$mike 'pos)
$ANY
(%i5) :lisp (put '$mike '$clause 'pos)
$CLAUSE
(%i5) :lisp (get '$mike 'pos)
$CLAUSE
(%i5) mike a := freeof(a,x)$
(%i6) if mike (x^2+1) then print("test1")$
test1
(%i7) :lisp (put '$mike '$expr 'pos)
$EXPR
(%i7) if mike (x^2+1) then print("test1");
Incorrect syntax: Found algebraic expression where logical
expression expected if mike (x^2+1) then
```

最初に定義した前置式演算子 mike の属性は any ですが, LISP の put 関数を用いて返却値の型を指定する属性 pos の値を clause に変更します. 次に mike 本体を定義し, 次の if 文で演算子 mike を使いますが, 演算子 mike の返却する型が clause 型の為に問題はありません. 次に LISP の put 関数を使って, 演算子 mike の返却値の属性を expr 型に変更します. すると二度目の if 文では演算子 mike が expr 型を返す為にエラーになります.

5.3.4 演算子の属性を宣言する関数

Maxima では利用者が記号や文字列を演算子として利用する事が可能です. 又, 関数も演算子としての属性を持たせれば, その属性を持った演算子として利用出来ます. その上, 適当な対象に演算子としての属性を付加してしまえば, それだけで演算子として利用が出来ます. 又, 演算子が返す値や演算子としての実体を持っていなくても, atvalue 関数で特定の値に対して, 決った値を返したり, 勾配を設定したりする事も可能です. ここで演算子を宣言する関数には, infix, nary, prefix, postfix, matchdeclare, nofix があります.

これらの函数を解説する前に、幾つかの表記を導入します。先ず, `lbp` と `rbp` が左束縛力と右束縛力, `lpos` と `rpos` が左右の被演算子の型, `pos` が返却値の型を表現します。更に, `matchfix` 函数と `infix` 函数に関しては、束縛力を左右に分けずに束縛力を `bp`, 被演算子の型を `argpos` とします。

infix 型演算子

infix 函数の構文

<code>infix(a)</code>	対象 <code>a</code> を <code>infix</code> 型演算子として宣言
<code>infix(a, lbp, rbp)</code>	対象 <code>a</code> を <code>infix</code> 型の演算子として左右の束縛力を含めて宣言
<code>infix(a, lbp, rbp, lpos, rpos, pos)</code>	対象 <code>a</code> を <code>infix</code> 型の演算子として左右の束縛力と被演算子の型を含めて宣言

`infix` 函数は `infix` 型の内挿表現の演算子の宣言を行います。`infix` 型の演算子は数式 $a = b$ の中の演算子 $=$ の様に被演算子が演算子の左右に配置される演算子で、左右の束縛力を変更しておく必要のある演算子であるか、`lhs` 函数や `rhs` 函数を用いて被演算子を取り出す必要のある式を構築しなければならない場合に用いる演算子の型です。

nary 函数

nary 函数の構文

<code>nary(a)</code>	対象 <code>a</code> を <code>nary</code> 型演算子として宣言
<code>nary(a, bp)</code>	対象 <code>a</code> を <code>nary</code> 型演算子として束縛力を含めて宣言
<code>nary(a, bp, argpos, pos)</code>	対象 <code>a</code> を <code>nary</code> 型演算子として束縛力と被演算子の型を含めて宣言

`nary` 函数は `nary` 型の内挿表現演算子の宣言で用います。`nary` 函数で宣言可能な演算子は左右の束縛力が等しく、`rhs` 函数や `lhs` 函数を用いて左右の被演算子を取り出す必要が無いものに限定されます。この演算子の左右束縛力は初期値として 180 が設定されます。この演算子は結合律を満す演算子の宣言に向いています。

infix 型と nary 型の違い

infix 型と nary 型は共に内挿表現の演算子の型ですが, 性質が異なる属性です. この infix と nary の内部表現の違いを実際に確認しておきましょう.

```
(%i1) infix("<>")$
(%i2) nary("><")$
(%i3) eq1:1<>2<>3<>4$
(%i4) eq2:1><2><3><4$
(%i5) :lisp $eq1
(($<> SIMP) (($<> SIMP) (($<> SIMP) 1 2) 3) 4)
(%i5) :lisp $eq2
(($>< SIMP) 1 2 3 4)
```

ここで, infix 型演算子”<>”を使った式の内部表現が二項演算子の痕跡を残しているのに対し, nary 型演算子”><”を使った式の内部表現が平になって二項演算子の痕跡が無い事に注意して下さい. この内部表現の違いの為に infix 型の演算子に対してのみ lhs 関数と rhs 関数が機能するのです.

この性質の違いは, infix 演算子は結合律の成立を前提とした演算ではありませんが, nary 演算子では結合律の成立を前提としたものだからです.

例えば, 演算子”<”は infix 型の演算子で述語を構成しますが, 式 $a_1 < a_2 < a_3 < a_4$ は正しくない構文となりエラーになります. 勿論, $((a_1 < a_2) < a_3) < a_4$ と纏めると入力が可能ですが, 今度は, is 関数や ev 関数で, この式の評価が上手く出来ません. 正しい式の構文は, $a_1 < a_2$ and $a_2 < a_3$ and $a_3 < a_4$ の様に, 論理演算子 and を用いて結合しなければなりません.

nofix 関数**nofix 関数の構文**

```
nofix(a)      対象 a を無引数の演算子として宣言
nofix(a,pos)  対象 a を無引数の演算子として出力の型を含めて宣言
```

nofix 関数は無引数演算子の宣言を行います. 無引数演算子は演算子が何等の引数を持たない事を明示する為に使われます.

prefix 関数

prefix 関数の構文

<code>prefix(a)</code>	対象 <code>a</code> を前置表現の演算子として宣言
<code>prefix(a,rbp)</code>	対象 <code>a</code> を前置表現の演算子として右束縛力を含めて宣言
<code>prefix(a,rbp,rpos,pos)</code>	対象 <code>a</code> を前置表現の演算子として右束縛力と被演算子の型を含めて宣言

`prefix` 関数は前置表記演算子の宣言を行います. 前置表記の演算子は一個の引数のみを持ち, その引数は演算子の直後に置かれるものです.

postfix 関数

postfix 関数の構文

<code>postfix(a)</code>	対象 <code>a</code> を後置表現の演算子として宣言
<code>postfix(a,lbp)</code>	対象 <code>a</code> を後置表現の演算子として左束縛力を含めて宣言
<code>postfix(a,lbp,rpos,pos)</code>	対象 <code>a</code> を後置表現の演算子として左束縛力と被演算子の型を含めて宣言

`postfix` 関数は後置表記演算子の宣言を行います. 後置表記の演算子は一つの引数のみを持ち, 引数は演算子の前に置かれます.

matchfix 関数

matchfix 関数の構文

<code>matchfix(a,b)</code>	被演算子を対象 <code>a</code> と対象 <code>b</code> で挟む外挿式の演算子を宣言
<code>matchfix(a,b,argpos,pos)</code>	引数の型と結果の型を含めて宣言

`matchfix` 関数は任意個数の引数を二つの対象で囲む演算子を宣言します. 猶, 演算子名は左側の対象で代表されます.


```

%i5) matchfix("@-", "-@")$
%i6) @- a,b,c,d,e,f -@:=a*b*c+d*e^f;
                                     f
(%o6)          @-a, b, c, d, e, f-@ := a b c + d e
%i7) @- 1,2,3,4,5,6 -@;
(%o7)                                     62506
%i8) dispfun("@-");
                                     f
(%t8)          @-a, b, c, d, e, f-@ := a b c + d e

(%o8)                                     done

```

この例では matchfix 演算子の定義を行い, dispfun 関数で定義内容を表示させています.

演算子宣言関数で定められる属性

上述の演算子を宣言する関数で定められる属性を以下に示しておきます. 猶, ここでの左右の束縛力は値はデフォルト値です.

演算子宣言関数で定められる属性					
演算子	左束縛力	右束縛力	左変数型	右変数型	返却値型
postfix	180		any		any
prefix		180		any	any
infix	180	180		any	any
nofix					any
nary			any	any	any
matchfix			any	any	any

これらの関数によって変数型と返却型は any に設定されます. matchfix 関数で宣言された演算子の被演算子型は他の関数と異なり, 左右の変数型属性 lpos や rpos ではなく argpos 属性で指定し, その属性値は any になります.

5.3.5 演算子属性の削除

演算子属性は kill 関数や remove 関数で削除出来ます.remove 関数は演算子の属性のみを削除し,kill 関数は演算子を全体を削除します.

```
(%i10) nary("tama")$
(%i11) a tama b:=a+b^2$
(%i12) properties("tama");
(%o12) [function, operator, noun]
(%i13) remove("tama",op);
(%o13) done
(%i14) properties("tama");
(%o14) []
(%i15) prefix("mike")$
(%i16) mike x:=x!+1$
(%i17) kill("mike");
(%o17) done
(%i18) properties("mike");
(%o18) []
```

この例では内挿式演算子 tama を定義し,remove 関数を使って tama の演算子属性を第二引数に op を指定する事で削除しています. 実際に properties 関数で調べても tama の属性がありません. 又,kill 関数は演算子名を指定するだけで全てを削除しています. この remove 関数は properties 関数で調べた特定の属性だけを削除出来ます.

```
(%i19) nary("tama")$
(%i20) a tama b:=a+b^2$
(%i21) remove("tama",function)$
(%i22) properties("tama");
(%o22) [operator, noun]
(%i23) 3 tama 4;
(%o23) 3 tama 4
```

この例では, 第二引数 function にした為に, この演算子に割当てられた関数が削除されただけで, 演算子としての属性は残しています. その為, `3 tama 4;` と入力してもエラーにはなりません.

5.3.6 算術演算子

二項算術演算子として次の演算子が定義されています:

二項数式演算子

演算子	属性	例	概要
+	prefix	$a + b$	a と b の和
-	prefix	$a - b$	a と b の差
*	nary	$a * b$	a と b の可換積
/	infix	a / b	a の b による商
**		$a ** b$	冪 a^b
^		$a ^ b$	冪 a^b , $a ** b$ と同じ
.	infix	$a . b$	a と b の非可換積
^^		$a ^^ b$	a と b の非可換積の冪乗

これらの演算子の持つ属性を以下に示しておきます:

二項算術演算子の持つ属性

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値の型
+	100	100	不要	expr	expr
-	100	134	不要	expr	expr
*	120	不要	expr	不要	expr
/	120	120	expr	expr	expr
^	140	139	expr	expr	expr
**	140	139	expr	expr	expr
.	130	129	expr	expr	expr
^^	140	139	expr	expr	expr

これらの演算子項は通常の数式と殆ど同じ表記ですが、演算子“-”, 演算子“/”は Maxima 内部では表示された式とは別表現です (§6.4 参照).

例えば, Maxima の式 $x-y$ は Maxima 内部では $x+(-1)*y$, x/y は $x*y^{-1}$ に対応する内部表現となっています. この内部表現を強制的に本来の形に変換する関数 `dispform` もあります. 更に,-に関しては, 大域変数 `negsumdispflag` を `false` に変更する事で, 内部表現通りに表示させる事も可能です.

非可換積の冪“^^”と通常の可換積の冪“^”は別物です. 例えば, $a^{^3}$ は $a . a . a$ を意味し, a^3 は $a * a * a$ を意味します. 猶, 可換積“*”と非可換積“.”, そして, そ

れらに対応する冪乗 a^b と $a^{(b)}$ が Maxima の式中に混在しても構いません. 又, 演算子 a^b を含む項の表示では, 右側の被演算子が通常の冪, 例えば, x^3 の様に上付きで表示されるのに対し, 演算子 $a^{(b)}$ は $x^{(n)}$ の様に $\langle \rangle$ で括られて表示されます.

```
(%i1) a^b;
```

```
<b>
```

```
(%o1)
```

```
a
```

ここで非可換積 $a.b$ と非可換冪 $a^{(b)}$ は行列で用いられます. この場合, 非可換積 $a.b$ が通常の行列の積を意味し, 可換積 $a*b$ はスカラー積や同じ大きさの行列に対する成分毎の積を取る演算子になります.

```
(%i54) A:matrix([1,2],[3,4]);
```

```
[ 1  2 ]
```

```
(%o54)
```

```
[      ]
```

```
[ 3  4 ]
```

```
(%i55) B:matrix([2,1],[4,3]);
```

```
[ 2  1 ]
```

```
(%o55)
```

```
[      ]
```

```
[ 4  3 ]
```

```
(%i56) A*B;
```

```
[ 2  2 ]
```

```
(%o56)
```

```
[      ]
```

```
[ 12 12 ]
```

```
(%i57) A.B;
```

```
[ 10  7 ]
```

```
(%o57)
```

```
[      ]
```

```
[ 22 15 ]
```

可換積の冪 a^b と非可換積の冪 $a^{(b)}$ の計算結果が長くて式が表示し切れない場合, 記号 `expt` が可換積の冪, `ncexpt` が非可換積の冪の表記で利用されます.

非可換積 $a.b$ と非可換冪 $a^{(b)}$ には挙動を制御する大域変数が存在します. 次に主な大域変数を示しておきます:

— 非可換積演算子に関連する大域変数 —

変数名	初期値	true の場合の作用の概要
dot0nscsimp	true	0 と scalar 属性を持たない対象の非可換積を可換積に変換.
dot0simp	true	0 と scalar 属性を持つ対象の非可換積を可換積に変換.
dot1simp	true	1 とその他の項の非可換積を可換積に変換.
dotassoc	true	非可換積項に結合律を適用.
dotconstrules	true	定数と項の非可換積を可換積で置換. この大域変数は dotocimo, dotonscsimp, dot1simp に影響.
dotdistrib	false	非可換積項に分配律を適用.
dotexptsimp	true	同じ式による非可換積を非可換積の冪乗に変換.
dotident	1	非可換冪の 0 乗で返される値を設定.
dotscrules	false	群環の元と scalar 属性を持つ対象の非可換積を可換積に変換.

ここでスカラーと群環の元との非可換積に関連する大域変数は, scalar 属性を付与した対象には制御が効きますが, 通常の数値は大域変数とは無関係に可換積に置換されます.

```
(%i6) 2 . 3 . x . y;
```

```
(%o6) 6 (x . y)
```

猶, 非可換積を Maxima で記述する場合, $a \cdot b$ の様に空白を空けるべきです. これは引数が数値の場合は必須です. 何故なら, 1.2 は 1 と 2 の非可換積でなく, 浮動小数 1.2 を意味し, $1. \cdot 2$ や $1 \cdot 2$ の様に中途半端に空行が入った場合, どちらも浮動小数点数と整数を空白で繋いだ式と解釈される為, 無意味な Maxima の式になるからです. 大域変数 `dotassoc` を `false` にすると, 複数の非可換積で勝手に括弧を外されない為, 非可換積に対する規則を定義した場合に, その適用が容易になります.

— 後置式の数式演算子 —

! $n!$ n が整数の場合, n の階乗を計算. 一般の場合 $\Gamma(x+1)$
 !! $n!!$ n が奇数 (偶数) ならば, n 以下の奇数 (偶数) の積

後置式演算子は引数を取ります. Maxima では "!" と "!!" が後置式の演算子となります. これらの演算子の数学的側面の詳細に関しては, §8.1 の階乗や gamma 関数の項

目を参照して下さい。

後置式表現演算子の属性

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値型
!	160		expr		expr
!!	160				

5.3.7 論理演算子

Maxima の論理演算子は Maxima の述語を構成する演算子です (§5.5 参照). 否定”not”, 論理和”or”, 論理積”and”は述語を評価する演算子ですが, その他の演算子は評価を伴わない二項間関係を表現する演算子です.

論理演算子

演算子	属性	例	概要
not	prefix	not a	述語 a を否定
and	nary	a and b	述語 a と b の論理積
or	nary	a or b	述語 a と b の論理和
=	infix	a = b	a と b が等しい
#	infix	a # b	a と b が等しくない
>=	infix	a >= b	a は b 以上
>	infix	a > b	a は b より也大
<=	infix	a <= b	a は b 以下
<	infix	a < b	a は b より小

C の `&&` や `||` に相当するものが, 演算子”and”と演算子”or”です. これらの演算子”and”と演算子”or”は述語を強引に評価します. この評価では文字通りの字面による解釈が実行される為, false が返される場合には注意が必要です.

二項間の関係を示す演算子は, Maxima 独特の演算子”#”と”=”を除くと, C や FORTRAN で利用されるものとの違いはありません.

ここで, 注意が必要なのは演算子”=”です. Maxima では変数への値の割当に演算子”:”を用い, 等しい事を示す関係子に演算子”=”を用い, この演算子”=”が C の”==”に対応します. この割当の演算子と等号の演算子は何気に混同し易いので注意して下さい.

次に論理演算子の属性値の一覧を示します:

論理演算子の属性値

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値の型
not		70	clause	clause	clause
and	65		clause		clause
or	60		clause		clause
=	80	80	expr	expr	clause
#	80	80	expr	expr	clause
>=	80	80	expr	expr	clause
>	80	80	expr	expr	clause
<=	80	80	expr	expr	clause
<	80	80	expr	expr	clause

5.3.8 割当の演算子

割当の二項演算子

演算子	属性	例	概要
:	infix	a : b	a に b の値を割当てて
::	infix	a :: b	a に b の値を割当てて
::=	infix	a ::= b	本体が b のマクロ a を定義
:=	infix	a:=b	本体が b の関数 a を定義

大域変数 `setcheck` に割当てた変数リストに含まれる変数に演算子 ":" と "::" を用いて割当を行う場合、変数名と割当てられた値の表示が行われます。

```
(%i11) setcheck:false;
(%o11)                                     false
(%i12) a1:20;
(%o12)                                     20
(%i13) setcheck:['a1','b1'];
(%o13)                                     [a1, b1]
(%i14) a1:10;
a1 SET TO 10
(%o14)                                     10
```

```
(%i15) a1:20;
a1 SET TO 20
(%o15)
```

20

割当の二項演算子の属性

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値の型
:	180	20	any	any	any
::	180	20	any	any	any
::=	180	20	any	any	any
:=	180	20	any	any	any

もし、演算子の実体を割当の関数で定義する際に、演算子の右束縛力が 180 よりも小であれば、演算子項の右側のメタ変数が割当の演算子に吸い取られます。この場合、割当の演算子の左側に配置した演算子項全体を小括弧で括る必要があります。

```
(%i7) infix("tama",111,111)$
(%i8) x tama y:= x+y*2;
Improper function definition:
y
-- an error. Quitting. To debug this try debugmode(true);
(%i9) (x tama y):= x+y*2$
(%i10) 2 tama z;
(%o10) 2 z + 2
```

この例では、左右の束縛力を 111 に設定した為、最初の演算子の実体の定義で、演算子項の右メタ変数 y が演算子 $:=$ に引き寄せられてエラーになっています。そこで、左側の演算子項 $x\ \text{tama}\ y$ 全体を小括弧で括る事で割当の演算子 $:=$ の束縛力を遮断して、演算子 tama の実体を定義しています

5.3.9 その他の演算子

式を構成する多くの構成要素が実は演算子として扱われています。その結果、演算子としての束縛力がプログラムの構築でも影響します。

先ず、括弧や引用符、そしてコンマの束縛力の表を示します：

括弧等の演算子としての束縛力

演算子	左束縛力	左変数型	返却値型
]	5		
[	200	any	any
)	5		
(	200		
'	190		
"	190		
,	10	any	any

ここで括弧に関しては多少解説が必要でしょう. 先ず, 文字 [と文字 (は matchfix 型の演算子名として利用され, 共に束縛力が 200 ある為, [に対応する] と (に対応する) は, 外部からの演算子の束縛力を完全に遮断する事になります.

if 文を構成する対象も同様に束縛力を持ちます:

if 文に関連する演算子の束縛力

演算子	左束縛力	右束縛力	右変数型	返却値型
if		45	clause	any
then	5	25		
else	5	25		
elseif	5	45	clause	any

最後に do 文を構成する対象も同様に束縛力を持ちます.

do 文に関連する演算子の束縛力

演算子	左束縛力	右束縛力	右変数型	返却値型
for	25	200	any	any
from	25	95	any	any
step	25	95	expr	any
next	25	45	any	any
thru	25	95	expr	any
unless	25	45	clause	any
while	25	45	clause	any
do	25	25	any	any

5.3.10 演算子に関連する函数

lhs 函数と rhs 函数

infix 属性を持つ内挿演算子から被演算子を取り出す函数として, lhs 函数と rhs 函数があります. lhs 函数が infix 属性の演算子を挟んで左側の被演算子を取り出し, rhs 函数が右側の被演算子を取り出します.

lhs と rhs

```
lhs(<infix 演算子を最上層に含む式>)
rhs(<infix 演算子を最上層に含む式>)
```

これらの函数は, infix 属性を持つ演算子のみに有効です.

```
(%i55) nary("<>")$
(%i56) A:((x+1)^2) <> ((z+a)^3)$
(%i57) lhs(A);rhs(A);

              2              3
(%o57)      ((x + 1) ) <> ((z + a) )
              0
(%o58)
(%i59) infix("<>")$
(%i61) lhs(A);rhs(A);

              2
(%o61)      (x + 1)
              3
(%o62)      (z + a)
```

この例では最初に nary 函数で演算子”<>”を宣言していますが, この演算子に対しては lhs 函数が上手く動作していません. 次に, infix 函数を用いて同名の演算子を宣言します. すると, 今度は lhs 函数と rhs 函数の両方が作用している事が判ります. この理由は内部表現の違いによるもので, nary 型の演算子は結合律の成立を前提とした演算子ですが, infix 型の演算子はそうではなく, その結果, 二項演算子の形を常に保っているからです.

op 関数

op 関数は一つの演算子や関数を含む式から演算子や関数を取り出す関数です.

op 関数の構文

op(< 式 >)

猶, この関数の実体は part 関数 (§6.4.10 参照) で, 第二引数として 0 を指定したもので, $\lambda x. \text{op}(x) \equiv \lambda x. \text{part}(x, 0)$ の関係があります.

```
(%i18) op(a+b+c+d);
(%o18)                                     +
(%i19) op(a*(b+c+d));
(%o19)                                     *
(%i20) op(sin(a*(b+c+d)));
(%o20)                                     sin
(%i21) part(sin(a*(b+c+d)),0);
(%o21)                                     sin
(%i22) part(a*(b+c+d),0);
(%o22)                                     *
(%i23) part(a+b+c+d,0);
(%o23)                                     +
```

この様に, op 関数の結果と, part 関数の第二引数に 0 を指定した関数による結果が一致します.

5.4 属性

一般的に属性は変項が一つだけの述語の事です。そして、属性から集合 (クラス, 或いは外延) が一つ定められます。そして、集合が定められる事で、その集合に属する対象に対して固有の操作を指定したり、その集合毎に持ち得る値 (属性値) を定める事が可能です。

Maxima では、函数や演算子の線形性や対象が偶数であるといった数学上の性質を属性として表現し、その属性を函数で表現しています。特に線形性の様に、式の変換を伴う性質を表現する函数の事を、ここでは属性の表現函数と呼びます。そして、対象を評価する際に、Maxima は文脈や属性を調べて、その属性に対応する属性の表現函数等を対象に作用させ、式の簡易化を行います。

ここで属性が設定可能な Maxima の対象は記号と文字列です。但し、記号や文字列は Maxima の原子であるだけでなく、函数や演算子の名前として利用可能です。従って、函数や演算子の性質が表現可能となるのです。更に、属性の設定では目的に応じて様々な Maxima の函数が用いられます。

属性とその設定函数に関しては次の様に大きく分類出来ます:

- 一般的な属性の設定:
put 函数,qput 函数
- 属性を付与する事で副作用を伴う属性の設定:
declare 函数
- 函数の値, 勾配,Taylor 展開, 依存関係等を属性で設定:
atvalue 函数,gradef 函数,deftaylor 函数,depends 函数等

これらの函数で属性が付与された対象は大域変数 props に割当てられたリストに登録されます。この大域変数 props に登録された対象に付与された属性は properties 函数で表示出来ます。

Maxima では属性を文脈でも利用します。属性はその性格上、特定の文脈に限定されるものではなく、大域的なものになります。この文脈と属性の関連に関しては、§5.6 で解説します。

5.4.1 put 関数による一般的な属性指定

put 関数と qput 関数は利用者が自由に属性を指定する事が可能な関数です. 基本的には利用者が集合を一つ定め, その集合に於ける対象の値を対象に与える為に用いる事が可能です. 従って, 一つの属性に対して一つの属性値のみが設定可能です.

これらの関数を用いて属性を与えた対象は属性が付加された対象は大域変数 props では [user properties] に分類されています.

一般的な属性に関連する関数

```
put(<対象>, <属性値>, <属性>)
qput(<対象>, <属性値>, <属性>)
get(<対象>, <属性>)
rem(<対象>, <属性>)
```

put 関数と qput 関数は <対象> の <属性> に <属性値> を付加します.

qput 関数は内部で put 関数を用いている関数ですが, 引数の評価を行わない点で put 関数と異なります (qput=quote put).

get 関数は put 関数と qput 関数で与えた属性値を <対象> と <属性> を指定する事で取得する関数です.

rem 関数は put 関数や qput 関数で与えた属性を削除する関数です. rem 関数は <対象> の <属性> と属性値を削除しますが, 大域変数 props に割当てられたリストから対象を削除しません.

これら put 関数, qput 関数, get 関数と rem 関数の実体は内部関数 prop1 です.

```
(%i2) put("ミケ", "3 歳", "年齢")$
(%i3) get("ミケ", "年齢");
(%o3)
3 歳
(%i4) props;
(%o4) [nset, {, }, kron_delta, trylevel, maxmin, nummod, conjugate, desolve,
eliminate, adjoint, invert, ミケ]
(%i5) properties("ミケ");
(%o5)
[[user properties, 年齢]]
```

5.4.2 declare 関数 について

declare 関数は Maxima の記号と文字列に対し、予め定義された属性 (features) を付加する関数です。

declare 関数の構文

```
declare(< 対象1>, < 属性1>, < 対象2>, < 属性2>, ...)
declare(< 対象 >, [< 属性1>, ..., < 属性n>])
declare([< 対象1>, ..., < 対象m>], [< 属性1>, ..., < 属性n>])
```

declare 関数で < 対象_i> に < 属性_i> が付加されます。declare 関数の属性をリストで与えた場合、属性リストに対応する対象に、その属性リストに含まれる全ての属性が付加されます。又、対象をリストで与えた場合、対象のリストに含まれる全ての対象に属性が指定されます。

```
(%i10) declare(a1,[integer,odd])$
(%i11) declare([b1,c1,d1],[integer,odd])$
(%i12) featurep(d1,odd);
(%o12)                                     true
(%i13) featurep(c1,integer);
(%o13)                                     true
```

この例では、最初に declare 関数を使って、一々、記号 a1 が整数である事、奇数である事を宣言しています。次の例では記号 b1,c1 と d1 が整数で、しかも奇数である事を一度に宣言しています。この様に、複数の属性を設定する場合は属性をリストで与え、複数の対象に同じ属性を与える場合、対象をリストで与える事が可能です。猶、この例では featurep 関数を用いて検証を行っています。

複数の属性を一つの対象に設定する場合、設定する属性は無矛盾でなければなりません。例えば、偶数、且つ奇数であると宣言する事は出来ませんが、属性が矛盾しなければ問題ありません。

```
(%i11) declare(n1,odd)$
(%i12) declare(n1,even)$
Inconsistent Declaration: declare(n1,even)
-- an error. Quitting. To debug this try debugmode(true);
(%i13) declare(n2,integer)$
(%i14) declare(n2,even)$
```

declare 関数による属性の付与は、その属性の表現関数の存在を前提としており、実際に式の評価で、この属性の表現関数を用いて式の変換が遂行されます。ここで、declare 関数による影響は、文脈 (§5.6 参照) で利用される assume 関数による影響と比較し、特定の文脈上に留まらずに大域的に影響を及ぼします。これは declare 関数による属性の付与は対象に対して行われる為で、文脈に対して対象と属性が関連付けられる為ではないからです。しかし、その一方で、declare 関数を用いた文脈上でのみ、`facts();` で内容が表示されます。従って、文脈 A 上で declare 関数で付与した属性は文脈 A 上でのみ `facts();` と入力して確認可能ですが、declare 関数で設定した属性を利用する事は何処の文脈上でも可能です。

5.4.3 declare 関数で付与可能な属性

declare 関数で付与可能な属性の詳細は §5.4.6 で述べますが、ここでは内部処理の違いから属性を分類して示しておきます：

declare 関数で付与可能な属性	
属性	概要
• <code>evfun, evflag, special, nonarray</code>	⇔ 属性を付与し、対象を属性に対応する大域変数に登録
• <code>noun</code>	⇔ 属性を付与し、対象の名詞化
• <code>constant, nonscalar, scalar, mainvar</code>	⇔ 属性に応じて処理
• <code>alphabetic</code>	⇔ alphabetic 属性を対象に付与
• 大域変数 <code>opproperties</code> 内の属性	⇔ 属性に応じて処理
• 大域変数 <code>features</code> 内の属性	⇔ 属性に応じて処理
• <code>feature</code>	⇔ 属性を大域変数 <code>features</code> に登録

ここで、大域変数 `opproperties` に含まれる属性の一覧を示します：

大域変数 <code>opproperties</code> に含まれる属性			
<code>linear</code>	<code>additive</code>	<code>multiplicative</code>	<code>outative</code>
<code>evenfun</code>	<code>oddfun</code>	<code>commutative</code>	<code>symmetric</code>
<code>antisymmetric</code>	<code>nary</code>	<code>lassociative</code>	<code>rassociative</code>

大域変数 `opproperties` に含まれる属性は、基本的に演算子や関数が持つ属性で、属性の表現関数を持つものです。この表現関数は対象の評価を行う際に利用されます。

次に、大域変数 `features` に含まれる属性の一覧を示しておきます：

大域変数 features に含まれる属性

integer	noninteger	even	odd
rational	irrational	real	imaginary
complex	analytic	increasing	decreasing
oddfun	evenfun	posfun	commutative
lassociative	rassociative	symmetric	antisymmetric
integervalued			

大域変数 features に含まれる属性は feature と呼びます。この属性は一部、大域変数 `opproperties` にも含まれる属性がありますが、基本的に数学上の一般的な性質を述べたもので、文脈と組合せて利用可能な属性になります。

内部的には、大域変数 features に含まれる属性に対して Maxima は、対象と属性を結び付ける内部関数 `kind` による表現を導入しています：

kind

```
kind(< 対象 >, < 属性 >)
```

猶、ここでの大域変数 features と feature は `status` 関数で扱う内部変数 `*features*` とは別物です。

5.4.4 属性の追加

`declare` 関数では属性として feature を指定する事で、利用者が属性を追加する事が可能です。この属性は、`declare` 関数を用いて利用する事が当然可能です。

declare 関数による属性の追加

```
declare(< 新属性 >, feature)
```

`declare` 関数で属性を宣言すると、大域変数 features に割当てられたリストに `< 新属性 >` が追加されます。

ここで対象が大域変数 features に関連する属性を持つかどうかは真理関数の `featurep` 関数を使って調べられます：

真理関数 featurep

```
featurep(< 対象 >, < 属性 >)
```


この関数は真理値集合を $\{\text{true}, \text{false}\}$ とし、対象が指定した属性を持つ時に `true` を返します。猶、`featurep` 関数は第二引数が `complex` であれば、第一引数が何であろうと `true` を返却する仕様となっています。

ここでは新しい属性として四元数 (quaternion) を追加してみましょう。

```
(%i1) features;
(%o1) [integer, noninteger, even, odd, rational, irrational,
      real, imaginary, complex, analytic, increasing, decreasing,
      oddfun, evenfun, posfun, commutative, lassociative,
      rassociative, symmetric, antisymmetric, integervalued]
(%i2) declare(quaternion,feature)$
(%i3) features;
(%o3) [integer, noninteger, even, odd, rational, irrational,
      real, imaginary, complex, analytic, increasing, decreasing,
      oddfun, evenfun, posfun, commutative, lassociative,
      rassociative, symmetric, antisymmetric, integervalued,
      quaternion]
(%i4) declare(q1,quaternion)$
(%i5) featurep(q1,quaternion);
(%o5)                                     true
```

この例で、`declare(quaternion,feature);` で `quaternion` 属性を宣言すると、`declare` 関数は大域変数 `features` に `quaternion` 属性を追加しています。次に、`declare(q1,quaternion);` で対象 `q1` が `quaternion` 属性を持つ対象であると宣言し、`featurep` 関数で対象 `q1` が `quaternion` 属性を持つ事を確認しています。

これで `quaternion` 属性を Maxima に入れました。後は `quaternion` 属性を持つ対象を処理する Maxima の関数、即ち、属性の表現関数を定義すれば良いのです。

5.4.5 属性の表現関数

属性の表現関数の解説の前に、重要な変数の解説をしておきます。先ず、大域変数 `opproperties` の実体は内部変数 `opers` に割当てられたリストです：

```
(%i1) opproperties;
(%o1) [linear, additive, multiplicative, outative, evenfun,
      oddfun, commutative, symmetric, antisymmetric, nary,
```

```
lassociative, rassociative]
(%i2) :lispopers
($RASSOCIATIVE $LASSOCIATIVE $NARY $ANTISYMMETRIC $SYMMETRIC
$COMMUTATIVE $ODDFUN $EVENFUN $OUTATIVE $MULTIPLICATIVE
$ADDITIVE $LINEAR)
```

この様に大域変数 `opproperties` の内容は内部変数 `opers` の成分を逆に並べたものです。そして、内部変数 `opers` は内部変数 `*opers-list` に登録されたリストから属性のみを抜き出したものです:

```
(%i2) :lisp *opers-list
(($RASSOCIATIVE . RASSOCIATIVE) ($LASSOCIATIVE . LASSOCIATIVE)
($NARY . NARY1) ($ANTISYMMETRIC . ANTISYM)
($SYMMETRIC . COMMUTATIVE1) ($COMMUTATIVE . COMMUTATIVE1)
($ODDFUN . ODDFUN) ($EVENFUN . EVENFUN) ($OUTATIVE . OUTATIVE)
($MULTIPLICATIVE . MULTIPLICATIVE) ($ADDITIVE . ADDITIVE)
($LINEAR . LINEARIZE1))
```

ここで、属性 `commutative` に注目して下さい。この属性は LISP 側では `$commutative` が対応します。そして、この `$commutative` はリスト `($COMMUTATIVE . COMMUTATIVE1)` が対応します。このリストの第二成分 `commutative1` が属性の表現函数となります。この事を `commutative1` の動作を LISP の函数 `trace` を用いて確認してみましょう:

```
(%i4) infix("(^_^)")$
(%i5) declare("(^_^)",commutative)$
(%i6) :lisp (trace commutative1)
WARNING: TRACE: redefining function COMMUTATIVE1 in top-level, was defined in
      /usr/local/maxima-5.13.0/src/binary-clisp/asum.fas
;; Tracing function COMMUTATIVE1.
(COMMUTATIVE1)
(%i6) Y (^_^ ) P - P (^_^ ) Y;
1. Trace: (COMMUTATIVE1 '((|$(^_^)|) |$y| |$p|) 'NIL)
1. Trace: COMMUTATIVE1 ==> ((|$(^_^)| SIMP) |$p| |$y|)
1. Trace: (COMMUTATIVE1 '((|$(^_^)|) |$p| |$y|) 'NIL)
1. Trace: COMMUTATIVE1 ==> ((|$(^_^)| SIMP) |$p| |$y|)
(%o6)
0
```

この例では, infix 型の演算子 ($\wedge$) を定義し, この演算子項を含む式を入力しています. すると, 内部関数の `commutative1` によって被演算子の交換が行われている事が判ります.

この変換が生じる理由は, 入力式の自動簡易化を担当する内部関数 `simplifya` があり, この内部関数は大域変数 `simp` の値が `true` の場合に, 入力式に含まれる関数や演算子の大域変数 `features` に登録された属性を調べ, 合致する属性が存在する場合は, 内部変数 `*opers-list` から, その属性に対応する内部関数を用いて式の簡易化を実行する仕様となっています.

この例で解説すると, 式が入力されると内部関数 `simplifya` で式は項毎に分解されます. それから, 演算子 ($\wedge$) の各演算子項が全て演算子属性を持つ事から, 各々が内部関数 `oper-apply` に引渡されます. そして, 内部関数 `oper-apply` で内部変数 `*opers-list` に登録された属性と演算子項の属性を照合し, 一致する属性があれば属性の表現関数を用いて演算子項の処理を行います. ここでは属性は `commutative` 属性のみの為, 用いられる内部関数も `commutative1` だけです. 但し, 入力式には演算子項が二つあり, 共に `commutative` 属性を持つ為に二度 `commutative1` が呼出され, 最初の演算子項の被演算子が項順序 $>_m$ で並び換えられています.

このような演算子の自動簡易化を用いたければ, 属性とその属性の表現関数を定義し, 双方を `*opers-list` に登録するか, `tellsimp` 関数を用いた規則を用いて内部関数 `simplifya` に簡易化の手順を教える必要があります (§5.7.6 参照). 但し, 内部変数 `*opers-list` を直接操作する Maxima の関数は存在しない為, この操作は全て LISP 側からの操作になります.

5.4.6 declare 関数に用意された属性

システム変数に関連する属性

システム変数に関連する属性を次に示します:

システム変数に関連する属性

<code>declare(<f>, evfun)</code>	対象 <f> に evfun 属性を付加
<code>declare(<a>, evflag)</code>	大域変数 <a> に evflag 属性を付加
<code>declare(<a>, nonarray)</code>	<a> を配列ではないと宣言
<code>declare(<a>, special)</code>	<a> を special 変数として宣言

これらの属性では属性を付与する対象の Maxima 内部の名前から \$ を削除して属性に対応する内部変数等に登録する副作用があります.

evfun 属性と evflag 属性は LISP の属性リストで与えられる属性で、これらの属性を持つ関数、演算子や大域変数は ev 関数を用いた評価で利用されます。ここで、ev 関数の引数として与えられた evflag 属性を持つ大域変数は、ev 関数による式の評価で内部的に true が設定されて処理が実行されます。同様に evfun 属性を持つ関数を ev 関数の引数とすると、evflag 属性を持った引数とその関数を与式に適用して処理を行います (§5.8.2 参照)。

special で対象が LISP の special 変数であると宣言します。この宣言はあまり表に出る事はありません。special 変数は関数の最適化等でも用いられますが、この変数に属性を指定して、割当てられる値に制限を加える事も可能になります。この方法は、define_variable 関数で解説します。

記号の型に関連する属性

記号の型に関連する属性を次に示します:

記号の型に関連する属性

declare($\langle a \rangle$, scalar)	$\langle a \rangle$ をスカラーとして宣言
declare($\langle a \rangle$, nonscalar)	$\langle a \rangle$ をスカラーではないと宣言
declare($\langle a \rangle$, constant)	$\langle a \rangle$ を定数として宣言
declare($\langle a \rangle$, alphabetic)	$\langle a \rangle$ を記号として宣言
declare($\langle f \rangle$, noun)	$\langle a \rangle$ を名詞型として宣言
declare($\langle a \rangle$, mainvar)	$\langle a \rangle$ を主変数として宣言

scalar 属性や nonscalar 属性の付与は行列やベクトルの処理で大きく影響します。

alphabetic 属性を対象に付与する事で Maxima の記号になります (§4.12.3 参照)

noun 属性を付与する事で、内部で nounify 関数を用いて対象は名詞型になります。

mainvar(主変数) 属性の付与は多項式の処理に大きく影響します。何故なら、多項式は順序 $>_m$ で最高位の変数の多項式として表現される為です:

```
(%i1) f1:(x+y)^4$
```

```
(%i2) f1,expand;
```

```
(%o2)          4      3      2 2      3      4
              y  + 4 x y  + 6 x  y  + 4 x  y + x
```

```
(%i3) f1,declare(x,mainvar),expand;
```

```
(%o3)          4      3      2 2      3      4
              x  + 4 y x  + 6 y  x  + 4 y  x + y
```

```
(%i4) ans1:%o2$
```

```
(%i5) ans2:%o3$
```

先ず, $y >_m x$ の為, 最初の式の展開は y の多項式となります.

次の `f1,declare(x,mainvar),expand;` で, x を主変数として式を展開します (§5.8.2 参照). その結果は変数 x の多項式となります. この表示の違いは式の内部表現自体が異っている事に由来します:

```
(%i6) :lisp $ans1
```

```
((MPLUS SIMP) ((MEXPT SIMP) $X 4) ((MTIMES SIMP) 4 ((MEXPT SIMP) $X 3) $Y)
((MTIMES SIMP) 6 ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 2))
((MTIMES SIMP) 4 $X ((MEXPT SIMP) $Y 3)) ((MEXPT SIMP) $Y 4))
```

```
(%i6) :lisp $ans2
```

```
((MPLUS SIMP) ((MEXPT SIMP) $Y 4) ((MTIMES SIMP) 4 ((MEXPT SIMP) $Y 3) $X)
((MTIMES SIMP) 6 ((MEXPT SIMP) $Y 2) ((MEXPT SIMP) $X 2))
((MTIMES SIMP) 4 $Y ((MEXPT SIMP) $X 3)) ((MEXPT SIMP) $X 4))
```

```
(%i6) ans1+ans2;
```

```
(%o6) y4 + 4 x y3 + 6 x2 y2 + 4 x3 y + 2 x4 + 4 y3 x + 6 y2 x2 + 4 y x3 + y4
```

```
(%i7) ev(%,simp);
```

```
(%o7) 2 x4 + 8 y x3 + 12 y2 x2 + 8 y3 x + 2 y4
```

実際,ans1 が変数 Y で式を纏めているのに対し,ans2 が変数 X で纏めている事が判ります. その上,ans1+ans2 の結果は, 最悪な事に ans1 と an2 を単純に繋げ,ans1 の末端と ans2 の先頭の x^4 を足し合せただけの結果となっています. この場合,ev 関数を用いて式を再評価する必要があります. 実際,ev 関数を用いて簡易化させる事で本来の結果が得られます.

数値属性

Maxima の数値属性は大域変数 `features` に登録された属性で, 内部変数 `opers` に登録されていないものが該当します:

数値属性

<code>declare($\langle a \rangle$, integer)</code>	$\langle a \rangle$ を整数として宣言
<code>declare($\langle a \rangle$, noninteger)</code>	$\langle a \rangle$ を非整数として宣言
<code>declare($\langle a \rangle$, even)</code>	$\langle a \rangle$ を偶数として宣言
<code>declare($\langle a \rangle$, odd)</code>	$\langle a \rangle$ を奇数として宣言
<code>declare($\langle a \rangle$, rational)</code>	$\langle a \rangle$ を有理数として宣言
<code>declare($\langle a \rangle$, irrational)</code>	$\langle a \rangle$ を無理数として宣言
<code>declare($\langle a \rangle$, real)</code>	$\langle a \rangle$ を実数として宣言
<code>declare($\langle a \rangle$, imaginary)</code>	$\langle a \rangle$ を純虚数として宣言
<code>declare($\langle a \rangle$, complex)</code>	$\langle a \rangle$ を複素数として宣言

Maxima には, これらの属性を基に式の簡易化を行う関数が多くあります.

更に, これらの数値属性の間には次の関係が成立します:

数値属性間の関係

関係	Maxima 内部での定義
<code>integer $\equiv$ even $\vee$ odd</code>	<code>par((\$even \$odd) \$integer)</code>
<code>real $\equiv$ rational $\vee$ irrational</code>	<code>par((\$rational \$irrational) \$real)</code>
<code>complex $\equiv$ real $\vee$ imaginary</code>	<code>par((\$real \$imaginary) \$complex)</code>
<code>integer $\rightarrow$ rational</code>	<code>kind(\$integer \$rational)</code>

属性の包含関係は内部関数 `par` と内部関数 `kind` を用いて表現されます.

例として, `integer` の属性を調べてみましょう:

```
(%i17) properties(integer);
(%o17) [database info, par(even, integer), kind(integer, rational), feature]
```

内部関数 `kind` と内部関数 `par` は似た関数ですが, 内部関数 `par` は属性間の包含関係のみを表現し, 第二引数の属性が二つの属性に分割可能な場合に, 属性間の包含関係の表現で用いられるのに対し, 内部関数 `kind` は, 属性間に対しては, その包含関係を表現します. 更に, 対象に付与した属性を表現する為にも用いられ, 含意“ $\rightarrow$ ”の意味で用いられています.

函数や演算子の属性

函数や演算子の属性は内部変数 `opers` と大域変数 `features` に登録された属性です。

これらの属性は演算子や函数の線形性といった作用に関連した属性や函数の単調増加といった函数自体の特徴に関連する属性に大きく二つに分類出来ます。

猶, 演算子の場合は引数が演算子の型によってまちまちとなりますが, 演算子と函数の内部表現の引数の配置は違いが無い為, ここでは函数を例に解説します。更に, 「演算子/函数 f 」と表記する代りに, ここだけ「対象 f 」と表記しますが, f 等の名称を後に伴わない「対象」は通常の Maxima の対象を意味します。

作用に関連する属性

<code>declare(<f>, additive)</code>	対象 $\langle f \rangle$ の加法性
<code>declare(<f>, multiplicative)</code>	対象 $\langle f \rangle$ の乗法性
<code>declare(<f>, outative)</code>	対象 $\langle f \rangle$ と積の可換性
<code>declare(<f>, linear)</code>	対象 $\langle f \rangle$ の線形性
<code>declare(<f>, commutative)</code>	対象 $\langle f \rangle$ の対称性
<code>declare(<f>, symmetric)</code>	対象 $\langle f \rangle$ の対称性
<code>declare(<f>, lassociative)</code>	対象 $\langle f \rangle$ の左分配律
<code>declare(<f>, rassociative)</code>	対象 $\langle f \rangle$ の右分配律

additive 属性

対象 f の第一引数での加法性を付与し, その結果, 次の変換が行われます:

$$f(x_1 + y_1, \dots) \Rightarrow f(x_1, \dots) + f(y_1, \dots)$$

猶, `sum` 函数に対しては効果がありません。

multiplicative 属性

対象 f の第一引数での乗法性を与えます。その結果, 次の変換が行われます:

$$f(x_1 * y_1, \dots) \Rightarrow f(x_1, \dots) * f(y_1, \dots)$$

猶, `multiplicative` 属性は, `additive` 属性と同様, 式の内部表現に依存する為, `product` 函数に対して無効になります。

outative 属性

対象 f の第一引数での可換積 $*$ と対象 f の交換性を付与します. その結果, 次の変換が行われます:

$$f(a * x_1, b * x_2, \dots) \Rightarrow a * f(x_1, b * x_2, \dots)$$

ここで f の作用域の外に出される対象は数値 (整数, 有理数, 浮動小数点数, 多倍長浮動小数点数) と constant 属性を持つ対象に限定されます. この outative 属性を持つ函数として, sum 函数, integrate 函数と limit 函数があります. この宣言は内部函数 defprop を用いて行われています.

linear 属性

linear 属性は additive 属性と outative 属性の両方を持たせた属性で, 対象 f の第一引数に対する線形性を付与します. その結果, 次の変換が生じます:

$$f(a * x_1 + b * y_1, \dots) \Rightarrow a * f(x_1, \dots) + b * f(y_1, \dots) \quad a, b \text{ は定数}$$

commutative(=symmetric) 属性

対象 f の任意の引数の順序に結果が依存しない性質を付与します. その結果, 対象 f の引数全体に対し, 項順序 $>_m$ による変換が生じます:

$$f(x_1, x_2, \dots, x_n) \Rightarrow f(x_{\sigma(1)}, x_{\sigma(2)}, \dots, x_{\sigma(n)})$$

ここで n を対象 f の引数の総数, σ は $x_{\sigma n} >_m \dots >_m x_{\sigma(2)} >_m x_{\sigma(1)}$ を満す n 次の置換群 $\mathfrak{S}_n$ の元となります. 猶, 対称性を表現する symmetric 属性は commutative 属性そのものを用いて表現しています.

antisymmetric 属性

歪対称性を表現する属性で, 次の式の変換が発生します:

$$f(x_1, x_2, \dots, x_n) \Rightarrow (-1)^{\#(\sigma)} f(x_{\sigma(1)}, x_{\sigma(2)}, \dots, x_{\sigma(n)})$$

ここで n を対象 f の引数の総数, σ は $x_{\sigma n} >_m \dots >_m x_{\sigma(2)} >_m x_{\sigma(1)}$ を満す n 次の置換群 $\mathfrak{S}_n$ の元, $\#(\sigma)$ は σ を互換の積に分解した時の互換の総数となります.

lassociative 属性と rassociative 属性

これらの属性は Maxima の式の内部表現を理解していなければ判り難い属性です. これらの属性は演算子を左がら次々と作用させられるか, 右から作用させられるかという性質に対応するものです.

ここで次の Maxima の式の模式的表現を用いて解説してみましょう. 猶, Maxima の式の内部表現の詳細は §6.4 を参照して下さい. 先ず, 対象 $f(x_1, \dots, x_n)$ の模式的内部表現を次に示しておきます:

対象 f の模式的内部表現	
Maxima の表示	Maxima の模式的内部表現
$f(x_1, \dots, x_n)$	$((f) x_1 \dots x_n)$

この様に, Maxima の式の内部表現は LISP の S 式の様な前置式表現となります².

先ず, lassociative 属性が対象 f に付与された場合, 次の式の変換が生じます:

$$f(x_1, x_2, \dots, x_n) \Rightarrow f(\dots f(f(x_1, x_2), x_3) \dots, x_n)$$

この変換式を模式的な内部表現で見ると次の式になります:

$$((f) ((f) \dots ((f) ((f) x_1 x_2 x_3) \dots x_{n-1}) x_n))$$

この様に, 対象 f の引数のリスト $(x_1, \dots, x_n)$ を左側から括弧で対として括って, 各リストの先頭に対象 f を作用させる形になります.

ここで, 対象 f を可換積 $*$ で置換えると, 変換式後の式は $((\dots ((x_1 * x_2) * x_3) \dots) * x_n)$ となり, この変換は左結合律を意味するものとなります.

rassociative 属性を対象 f に付与した場合, 次の変換が生じます:

$$f(x_1, x_2, \dots, x_n) \Rightarrow f(x_1, f(x_2, \dots f(x_{n-2}(f(x_{n-1}, x_n))) \dots))$$

この変換式を模式的な内部表現で見ると次の式になります:

$$((f) x_1 ((f) x_2 \dots ((f) x_{n-2} ((f) x_{n-1} x_n) \dots)))$$

この場合は対象 f の引数のリスト $(x_1, \dots, x_n)$ を右側から括弧で対として括って, 各リストの先頭に対象 f を作用させる形になります. ここで対象 f を可換積 $*$ で置換えると, 与式は $x_1 * (x_2 * (\dots (x_{n-2} * (x_{n-1} * x_n)) \dots))$ となり, 上述の変換は右結合律を意味するものとなります.

²正確には, $((\$f \text{ simp}) x_1 \dots x_n)$ となるでしょう

更に, declare 函数を用いた対象の属性に, 函数の単調増加性, 単調減少性, 奇函数, 偶函数, 正值函数, 解析的函数等の性質を指定する事も可能です.

—— 対象 f の属性 ——

declare($\langle f \rangle$, analytic)	対象 $\langle f \rangle$ を解析的函数として宣言
declare($\langle f \rangle$, increasing)	対象 $\langle f \rangle$ を単調増加函数として宣言
declare($\langle f \rangle$, decreasing)	対象 $\langle f \rangle$ を単調減少函数として宣言
declare($\langle f \rangle$, oddfun)	対象 $\langle f \rangle$ を奇函数として宣言
declare($\langle f \rangle$, evenfun)	対象 $\langle f \rangle$ を偶函数として宣言
declare($\langle f \rangle$, posfun)	対象 $\langle f \rangle$ を正值函数として宣言

次に Maxima 組込の函数に対する函数属性を示しておきます:

—— Maxima の組込函数が持つ函数属性 ——

atan	⇒	increasing ∧ oddfun
delta	⇒	evenfun
sinh	⇒	increasing ∧ oddfun
cosh	⇒	increasing
tabh	⇒	increasing ∧ oddfun
coth	⇒	oddfun
csch	⇒	oddfun
sech	⇒	posfun
li	⇒	complex
cabs	⇒	complex
log	⇒	increasing

函数と函数属性は内部函数 kind を用いて述語付けられます.

```
(%i18) properties(atan);
(%o18) [deftaylor, database info, kind(atan, increasing), kind(atan, oddfun),
        rule, noun, gradef, transfun, transfun, transfun, transfun]
```

この例では atan 函数の属性を properties 函数を用いて調べています. ここで, atan 函数の属性として, 内部函数 kind を用いた increasing と oddfun の二つの属性があります. 猶, 対象がこれらの属性を持つかどうかの判断は内部函数 kindp で行えます:

```
(%i28) :lisp (kindp '%atan '$increasing)
```

```
T
(%i28) :lisp (kindp '$integer '$rational)
T
(%i28) :lisp (kindp '$real '$complex)
T
```

atvalue 函数

atvalue 関数の構文を次に示します:

atvalue 函数

atvalue(⟨ 函数 ⟩, ⟨ 变数 ⟩ = ⟨ 式₁ ⟩, ⟨ 式_A ⟩)
atvalue(⟨ 函数 ⟩, [⟨ 变数₁ ⟩ = ⟨ 式₁ ⟩, ⋯, ⟨ 变数_n ⟩ = ⟨ 式_n ⟩], ⟨ 式_A ⟩)

atvalue 関数は第一引数に函数項や函数項の微分 (名詞型), 第二引数に第一引数の項で用いた変数の値, 第三引数に第二引数で指定した値で函数を充足した時に函数が取るべき値を指定する函数です. そして, 対象には atvalue 属性として, この値が割当てられ, 大域変数 props に対象が登録されます.

第二引数は単変数に値を指定する場合は〈変数〉=〈値〉となりますが、多変数の場合は、〈変数_{*i*}〉=〈値_{*i*}〉を成分とするリストとなります。ここで、〈変数_{*i*}〉=〈値_{*i*}〉の=は等号ではなく、ev 関数で用いる=と同様に、関数に含まれる左辺の変数を左側の対象で充足させる事を意味する記号です。

例えば, $\text{atvalue}(f(a), a=a^2, g)$ の意味は, 函数項 $f(a)$ の変数 a を a^2 で置換する事で得られた函数項 $f(a^2)$ の取る値が g である事を示します.

atvalue 関数で設定された値を printprops 関数を使って表示する際に、関数の疑似変数を表記する為に、記号@1.@2... が用いられます。

```
(%i10) atvalue(f(x),x=x^2,g);
(%o10)
g
(%i11) [f(x),f(1)];
(%o11)
[f(x), f(1)]
(%i12) atvalue(f(x,y,z),[x=a,y=b],g(z));
(%o12)
g(@3)
(%i13) f(a,b,10);
```

```
(%o13)                                     g(10)
```

ここで, `atvalue` 属性をもう少し調べてみましょう. `atvalue` 属性が対象に設定されている事は `properties` 関数で判り, 具体的な内容は `printprops` 関数で表示可能です:

```
(%i14) props;
(%o14) [nset, {, }, kron_delta, trylevel, maxmin, nummod, conjugate, desolve,
        eliminate, adjoint, invert, f]

(%i15) properties(f);
(%o15)                                     [atvalue]
(%i16) printprops(f, atvalue);
        f(a, b, @3) = g(@3)

        2
        f(x ) = g

(%o16)                                     done
```

この `atvalue` 属性は Maxima 内部では `atvalues` 属性となっています. この属性値は内部関数の `mget` 関数を用いて得られます:

```
(%i6) atvalue(f1(x1,x2,x3,x4,x5), [x1=a,x5=b], a*b*g(x2,x3,x4));
(%o6)                                     a b g(@2, @3, @4)
(%i7) :lisp (mget '$f1 'atvalues)
(((0 0 0 0 0) ($A ##### ##### ##### $B)
  ((MTIMES SIMP) $A $B (($G SIMP) &@2 &@3 &@4))))
```

`atvalues` 属性の属性リストは三成分のリストの並びの繰り返しで構成されています. この例では, 第一成分が関数項の変数の座を示す 0 で構成されたリストで, 第二成分が各変数の値, そして第三成分が返却される値になります. 猶, 変数の値で ##### とあるのは, Maxima で変数に値が割当てられていない事を示す内部変数 `unbound` の値です. 猶, 先程の `f` の様に複数の `atvalue` 属性が設定されている場合, 次の様なリストが得られます:

```
(%i7) :lisp (mget '$f 'atvalues)
(((0 0 0) ($A $B #####) (($G SIMP) &@3)) ((0) ((MEXPT SIMP) $X 2)) $G))
```

この場合,3 変数の関数 f の `atvalues` 属性と 1 変数の関数 f の `atvalues` 属性が並んでいる事が判ります.

at 関数

at 関数の構文

```
at(<式>, <変数>=<式1>)
at(<式>,[<変数1>=<式1>,<変数n>=<式n>])
```

`at` 関数は与えられた式の変数に指定された値を代入した式を返す関数です. この関数は代入処理の際に,`atvalue` 属性を調べて式の処理を行います. 代入を行う `subst` 関数とは, この点で異なります:

```
(%i28) atvalue(xa(x,y),x=1,cos(y)*gamma(y));
(%o28)                cos(@2) gamma(@2)
(%i29) at(xa(x,y),[x=1,y=3]);
(%o29)                2 cos(3)
(%i30) subst([x=1,y=3],xa(x,y));
(%o30)                xa(1, 3)
```

この例では最初に `atvalue` 関数で $xa(1,y)=\cos(y)*\gamma(y)$ となる事を `atvalue` 属性を用いて表現し,`at` 関数で $[x,y] \rightarrow [1,3]$ の場合の処理を行っています. この時,`atvalue` 属性を参照するので正しく処理が行えます. 一方で,`subst` 関数では直接値を入れるだけなので, 期待する値は得られていません.

gradef 関数と depends 関数

勾配と従属性の設定

```
gradef(<関数名>(<変数1>,...,<変数m>),<式1>,...,<式n>)
gradef(<記号>,<変数>,<式>)
depends(<関数>,<変数1>,...,<関数n>,<変数n>)
```

`gradef` 関数は関数の各変数 $\langle x_i \rangle$ による 1 階微分と $\langle 式_i \rangle$ を結び付ける関数です. 先ず,`gradef` 関数の第一引数に $f(x)$ の様な関数項を与えた場合, 関数名 f に `gradef` 属


```
(%o13) [g(x), y(z), h(x)]
```

ここで Maxima の組込関数も `gradef` 属性を用いて微分が定義されていますが, 大域変数 `gradefs` には登録されていません.

```
(%i104) properties(sin);
(%o104) [deftaylor, rule, noun, gradef, transfun, transfun, transfun, transfun]
(%i105) printprops(sin,gradef);
      d
      -- (sin(x)) = cos(x)
      dx

(%o105) done
```

この例では `sin` 関数の属性を `properties` 関数で表示させています. ここで表示される属性で, `gradef` が微分の公式の関係式が登録されている属性なので, `printprops` 関数を使って `gradef` 属性を表示させると, `sin` 関数の微分の公式が現れます. 組込の関数では, この属性の設定で内部関数の `defprop` 関数を用いています.

Taylor 級数を定める関数

deftaylor 関数

```
deftaylor(< 関数名 >(< 変数 >), < 式 >)
deftaylor(< 関数名1 >(< 変数1 >), < 式1 >, ..., < 関数名n >(< 変数n >), < 式n >)
```

`deftaylor` 関数は指定した関数の Taylor 展開式を指定する関数で, 関数には `deftaylor` 属性が付加され, Taylor 展開式は `deftaylor` 属性値として設定され, 更に, 関数名が大域変数 `props` に割当てられたリストにも登録されます.

```
(%i1) defaylor(f1(x),sum(x^(2*i+1)/i!,i,1,inf));
(%o1) [f1]
(%i2) taylor(f1(x),x,0,10);

      5      7      9
      3      x      x      x
x  +  --  +  --  +  --  +  . . .
      2      6      24

(%o2) T/
```

5.4.8 属性の表示

属性の表示に関連する関数としては, `properties` 関数, `propvars` 関数と `printprops` 関数があります.

属性の表示に関連する関数と大域変数

```
properties(< 対象 >)
propvars(< 属性 >)
props
```

`properties` 関数は < 対象 > に関連する全ての属性を記載したリストを表示します. 次の例では, `put` 関数で対象に属性を指定し, `properties` 関数で対象に指定した各種属性を確認し, `get` 関数を使って属性値を取り出しています:

```
(%i37) put(Mike, "2004/07/4", birthday)$
(%i38) put(Mike, "10[Kg]", Weight)$
(%i39) put(Mike, "White-Black-Red", Color)$
(%i40) properties(Mike);
(%o40)          [[user properties, Color, Weight, birthday]]
(%i41) get(Mike, Color);
(%o41)          White-Black-Red
```

`propvars` は内部で `properties` 関数を用いた関数で, 大域変数 `props` に割当てられたリストから, < 属性 > を持つ対象のリストを返します. 例えば, `propvars(atvalue)` で `atvalue` 関数で値が設定された記号のリストを返します:

```
(%i23) atvalue(f(x), x=0, 0)$
(%i24) atvalue(g(x), x=1, 0)$
(%i25) propvars(atvalue);
(%o25)          [f, g]
```

大域変数 `props` には `declare` 関数, `atvalue` 関数や `matchdeclares` 関数等で属性が指定された記号が追加されたリストが割当てられています.

```
(%i1) props;
(%o1) [nset, {, }, kron_delta, trylevel, maxmin, nummod, conjugate, desolve,
      eliminate, adjoint, invert]
```


5.4.9 remove 関数による属性の削除

remove 関数は declare 関数等で指定された属性の削除を行います.

remove 関数の構文

```
rem(<対象>,<属性>)  
remove(<記号1>,<属性1>,...,<記号n>,<属性n>)  
remove([<記号1>,...,<記号m>],[<属性1>,...,<属性n>])  
remove("<記号>",operator)  
remove(<記号>,transfun)  
remove(all,<属性>)
```

remove 関数は対象と属性を指定する事で、対象に付与された属性と属性値が削除されます.

ここで remove 関数で削除可能な属性を示しておきましょう:

属性の削除のみ

- assign
LISP の putprop 関数によって属性リストを持たされた対象から属性リストを削除します.
- atvalue
関数や記号に付与された atvalues 属性を削除します. 猶,atvalues 属性は atvalue 関数 で付与されます.
- autoloader
関数名に付与された autoloader 属性を削除します.autoloader 属性は setup_autoloader 関数 を用いて付与される属性です.
- evfun,evflag,special,nonarray,mainvar,constant, nonconstant,scalar,nonscalar
対象に割当てられた属性を単純に削除します. これらの属性は全て declare 関数 で与えられたもので, 関連する大域変数が存在せず, 属性値も true のみである為に属性の削除に留まります. これらの属性は declare 関数で与えられます
- matchdeclare
matchdeclare 属性を対象から削除します. 猶,matchdeclare 属性は matchdeclare 関数 で与えられる属性です.

- `mode,modeddeclare`
`modeddeclare` 関数 (= `mode.declare` 関数) で与えた `mode` 属性を削除します.
- `numer`
対象の `numer` 属性を削除します. この属性は, 属性値が内部関数 `mputprop` で与えられ, 大域変数 `numer` が `true` の場合に属性値が自動的に取り出されます. 猶, 大域変数 `values` に登録された対象に対しても, その値を削除する作用があります.
- `transfun`
対象の `transfun` 属性を削除します. この属性は `translate` 関数で与えられる属性で, `putprop` 関数で属性 `translated` として与えられています.

大域変数の書き換えを伴う

- `alphabetic`
内部変数 `*alphabet*` に登録された Maxima の記号から `alphabetic` 属性を削除し, Maxima の文字にします. その結果, 内部変数の `*alphabet*` に割当てられたリストから対象が削除されます.
- `array`
大域変数 `arrays` に登録された対象から, 割当てられた配列本体を削除します. その結果, 大域変数 `arrays` から対象が削除されます.
- `alias`
大域変数 `aliases` に登録された対象の `alias` 属性を削除します. この時, 大域変数 `aliases` から対象が削除されます. ここで `alias` 属性は `alias` 関数で与えられます.
- `function`
大域変数 `functions` に登録された演算子や関数の本体を削除し, 形式的な演算子や関数にします. この時, 大域変数 `functions` から対象は削除されます. この `function` 属性は演算子 `:=` や `define` 関数で与えられます.
- `macro`
大域変数 `macros` に登録された対象の `macro` 属性と `macro` の実体を削除します. この `macro` 属性は演算子 `::=` を用いて与えられます.

- operator と op
演算子としての属性を削除します. ここで function 属性を持つ演算子は演算子としての属性を削除されるだけなので, 通常の函数になります.
- features
大域変数 features に登録された対象の feature 属性を削除します. この結果, 対象は大域変数 features に割当てられたリストから削除されます. 猶, features 属性は declare 函数で与えられます.
- gradef, grad, atomgrad
登録された函数の勾配を削除します. 猶, 大域変数 graddefs に割当てられたリストに登録された対象は, このリストから削除されます. ここで atomgrad 属性を持った対象は graddefs に登録されていません. 猶, これらの属性は gradef 函数で与えられます
- defatylor と taylordef
Taylor 展開式を削除し, その結果, 大域変数の deftaylor に割当てられたリストから対象が削除されます. deftaylor 属性は deftaylor 函数で与えられます.
- depends, dependency, depend
大域変数 dependencies に登録された対象から, dependency 属性を削除します. その結果, 大域変数 dependencies から対象が削除されます. 猶, dependency 属性は depends 函数や gradef 函数で与えられます.
- rule
大域変数 rules に登録された対象から, rule 属性を削除します. その結果, 対象は大域変数 rules から削除されます. rule 属性は defrule 函数で与えられます.
- value
大域変数 values に登録された対象から割当てた値を削除します. その結果, 対象は大域変数 values から削除されます. value は正確には属性ではありません. 演算子: による割当て対象が大域変数 values に登録されます.

猶, remove 函数は与えられた属性が存在しない時でもエラーを返しません. remove 函数の返却値は常に done です.

5.5 述語

5.5.1 Maxima の述語について

Maxima の述語は Maxima の対象の性質を表記し,Maxima の自動簡易化や函数による判断で true や false 等の真理値を返す Maxima の式です (§5.1.9 参照).

述語の具体的な例を示しておきましょう:

```
(%i124) 3>5;
(%o124)                                     3 > 5

(%i125) 4#1;
(%o125)                                     4 # 1

(%i126) x^2+y^2+z^2>0;
(%o126)                                     2      2      2
                                     z  + y  + x  > 0

(%i127) diff(f(x),x,2)+k*f(x)=0;
(%o127)                                     2
                                     d
                                     --- (f(x)) + k f(x) = 0
                                     2
                                     dx
```

最初の二つは数値を比較する述語です. この様に Maxima で入力された述語が直ちに評価されるとは限らず, 比較の演算子の演算子項は ev 函数等の判断を行う為の函数を必要とします.

次の二つの例は, 変数や函数を伴う述語の例になります. この様に変数を持つ述語の場合, 通常, 変数の値等の条件によって述語の意味が異なるのが通常です. 勿論, $x - x = 0$ の様に変数 x の値に依存せずに真偽が定まる述語も存在しますが, この様な述語の事を恒真式 (tautology) と呼びます. 又, 述語 $x^2 - 3x + 2 = 0$ は $x = 1$ 又は $x = 2$ の場合に限って真になりますが, この様に変項の値で真となる述語の事を充足可能な述語と呼びます.

この命題 $x^2 - 3x + 2 = 0$ の様に述語が真であるかどうかは, その述語に含まれる変数の値域に依存しますが, Maxima では文脈を用いて判断を行います. (§5.1.9, §5.5 参照).

ここで文脈で利用可能な演算子とそうでない演算子があります:

- 文脈で利用可能な比較の演算子:

$>=, >, <=, <$

- 文脈で利用不可の比較の演算子:

$=, \#$

文脈で利用出来ない演算子 $=$ と演算子 $\#$ を表現する時は, `equal` 関数と `notequal` 関数の二つの関数を用いて述語を構成します:

equal 関数と notequal 関数

```
equal(< 式1>, < 式2>)
notequal(< 式1>, < 式2>)
```

`equal` 関数と `notequal` 関数は各々が演算子 $=$ と $\#$ と似ていますが, 内部表現が異なります. この事を簡単な例を使って確認しましょう. ここで, 内部表現の確認の為に, `:lisp` 演算子を用いています. 又, Maxima の式の内部表現に関しては §6.4 を参照して下さい.

```
(%i1) eq1:x^2=y^3;
                                2    3
(%o1)      x  = y
(%i2) eq2:x^2#y^3;
                                2    3
(%o2)      x  # y
(%i3) eq3:equal(x^2,y^3);
                                2    3
(%o3)      equal(x , y )
(%i4) eq4:notequal(x^2,y^3);
                                2    3
(%o4)      notequal(x , y )
(%i5) :lisp $eq1
((MEQUAL SIMP) ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 3))
(%i5) :lisp $eq3
(($EQUAL SIMP) ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 3))
(%i5) :lisp $eq2
((MNOTEQUAL SIMP) ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 3))
```

```
(%i5) :lisp $eq4
(($NOTEQUAL SIMP) ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 3))
```

ここでの例に示す様に, 演算子=を用いた述語 eq1 の内部表現のヘッダが MEQUAL であるのに対し, equal 函数を用いた述語 eq3 のヘッダが \$EQUAL になっています. 同様に演算子#を用いた述語 eq2 のヘッダが MNOTEQUAL であるのに対し, notequal 函数を用いた述語 eq4 のヘッダは \$NOTEQUAL になっています. この様に, 同じ意味の述語でありながら内部表現が異なる為, これらの式は本質的に別物であると言えます.

猶, equal 函数による述語と notequal 函数による述語は演算子 not で互いに移り合えます.

```
(%i5) not equal(x^2,y^3);

                                2   3
(%o5)                        notequal(x , y )
(%i6) not notequal(x^2,y^3);

                                2   3
(%o6)                        equal(x , y )
```

5.5.2 述語の判断

前述の様に, 比較の演算子 (>, <, >=, <=, =) と equal 函数や notequal 函数のみで構成された述語の判断を, Maxima は自動的に実行しません. その一方で, and 函数, or 函数, not 函数を含む述語が入力されると, これらの述語の評価を強引に自動的に行います:

```
(%i1) 4>=1;
(%o1) 4 >= 1
(%i2) 1>3 and 4>=1;
(%o2) false
(%i3) not(3>1);
(%o3) false
(%i4) x>1;
(%o4) x > 1
(%i5) not(x>1);
(%o5) x <= 1
(%i6) (x+1)^2-expand((x+1)^2)=0 and 4>1;
(%o6) false
```

```
(%i7) (x+1)^2-expand((x+1)^2)=0 or 4<1;
(%o7)                                     false
(%i8) equal((x+1)^2-expand((x+1)^2),0) and 4>1;
(%o8)                                     true
(%i8) notequal((x+1)^2-expand((x+1)^2),1) and 4>1;
(%o8)                                     true
```

先ず, 最初の $4 > 1$ では評価が実行されていませんが, 演算子 `and` を作用させる事で述語が `true` であると判断されています. 同様に, 述語に `not` 演算子を作用させると, 真理値が決定可能な述語に対しては真理値を返し, 決定不能な述語に対しては, 単純に, その述語を否定した述語に変換して返しています.

演算子 `and` や演算子 `or` による述語の判断は文字通りに字面だけで判断を行います. これは演算子 `and` と演算子 `or` が真理値に対する演算が主であって, 式の評価を基にした述語の判断を行う演算子ではない為です.

実際, 述語 $(x+1)^2 - \text{expand}((x+1)^2) = 0$ and $4 > 1$ や $(x+1)^2 - \text{expand}((x+1)^2) = 0$ or $4 < 1$ の評価の様に, 演算子 `and` や演算子 `or` の左側の述語は式を展開すれば真であるにも関わらず, 字面だけで判断して `false` を返しています.

これに対して, `equal` 函数を用いた場合, 多少は真面に式の評価が実行されている事が判ります. これは `equal` 函数や `notequal` 函数を用いた場合, これらの函数の属性表現函数にて, `ratsimp` 函数による式の評価が含まれている為です.

ここで, 演算子 `and` と演算子 `or` の `true`, `false` の真理値と, 特殊な意味を持つ `unknown`, そして項 x に対する値を見てみましょう:

——— 演算子 `and` と `or` (`true` と `false` に対して) ———

<code>true</code>	<code>and</code>	<code>true</code>	$\Rightarrow$	<code>true</code>
<code>true</code>	<code>and</code>	<code>false</code>	$\Rightarrow$	<code>false</code>
<code>false</code>	<code>and</code>	<code>false</code>	$\Rightarrow$	<code>false</code>
<code>true</code>	<code>or</code>	<code>true</code>	$\Rightarrow$	<code>true</code>
<code>true</code>	<code>or</code>	<code>false</code>	$\Rightarrow$	<code>false</code>
<code>false</code>	<code>or</code>	<code>false</code>	$\Rightarrow$	<code>false</code>

演算子 and と or(true,false,unknown,x に対して)

true	and	unknown	⇒	unknown
false	and	unknown	⇒	false
true	and	x	⇒	false
false	and	x	⇒	false
true	or	unknown	⇒	true
false	or	unknown	⇒	unknown
true	or	x	⇒	true
false	or	x	⇒	x

演算子 and は false が含まれていれば false を返し, unknown 以外の項があれば, false を返し, true と unknown の場合のみ unknown を返す演算子です. これに対し, 演算子 or は基本的に false 以外の値が式中に含まれていれば, その値を返す演算子です.

今度は文脈を用いた述語の判断の例を示しましょう:

```
(%i1) assume(x1>1 and y1<0);
(%o1) [x1 > 1, y1 < 0]
(%i2) x1>0 and -y1>0;
(%o2) true
(%i3) x1>0 and -y2>0;
(%o3) - y2 > 0
(%i4) x1>0 and abs(y2)=y2;
(%o4) false
```

この例では, assume 関数を用いて二つの述語 $x1 > 0$ と $y < 0$ を文脈に登録しています. 猶, 文脈は特に指定しない場合, initial を Maxima は用いて式の評価を行います. ここで, 文脈に登録された述語を $P_1, \dots, P_n$ とし, 評価すべき述語を P_0 とすると, 述語 $P_0 \wedge P_1 \wedge \dots \wedge P_n$ の判断を Maxima は行います (§5.1.10, §5.6 参照).

ここで assume 関数で有効な述語は, 論理演算子 or, 比較の演算子 = や #, 真理関数を含まない述語に限定されます. 猶, 比較の演算子 = と # は使えませんが, equal 関数や notequal 関数で代用する事が可能です. しかし equal 関数と notequal 関数は演算子 = と # と内部表現が異なる為, これらの式を混在させて評価する事は危険が伴います.

```
(%i1) assume(equal(x^2,1));
2
(%o1) [equal(x , 1)]
```

```
(%i2) x^2=1,pred;
(%o2)                                     false
(%i3) equal(x^2,1),pred;
(%o3)                                     true
(%i4) equal(x^2-x,1-x),pred;
(%o4)                                     true
(%i5) equal(x^2-1,0),pred;
(%o5)                                     true
(%i6) x^2-1=0,pred;
(%o6)                                     false
(%i7) x^2=1,pred;
(%o7)                                     false
```

この例では, `equal` 函数を用いて, x^2 と 1 が等しいという述語を設定しています. 意味的には $x^2=1$ ですが, `ev` 函数を用いた評価では, `equal` 函数を用いた述語では有効に作用しても, 演算子 `=` を用いた述語では, 評価される述語に文脈が反映されていません. 更に, `assume` 函数を用いて文脈に二つの式が等しい事を表現する為に, `equal` 函数ではなく比較の演算子を用いて表現する事は出来ません.

```
(%i14) assume(x1>=1 and x1<=1);
(%o14) [x1 >= 1, inconsistent]
(%i15) assume(x2>=1 and x2<=5);
(%o15) [x2 >= 1, x2 <= 5]
```

この例では, $x_1=1$ を示す為に, 述語を $x_1 \geq 1$ and $x_1 \leq 1$ を文脈に登録しようとしています. この場合, `and` 演算子の被演算子を `assume` 函数は文脈に登録して行きますが, $x_1 \leq 1$ が前の述語 $x_1 \geq 1$ との関連で妥当ではないとして排除しています. 最後の例は, `=` の関係でなければ, 受理される例です.

ここで文脈を用いて述語の評価を行う函数として, `is` 函数, `maybe` 函数と `ev` 函数があります. `is` 函数や `maybe` 函数は基本的に式の変形操作を行わない為, $(x+1)^2 - x^2 - 2x - 1 = 0$ and $3 > 0$ の様に式の展開等の操作が必要な式には対処出来ずに `false` を返します.

```
(%i27) is((x+1)^2-expand((x+1)^2)=0);
(%o27)                                     false
(%i28) maybe((x+1)^2-expand((x+1)^2)=0);
(%o28)                                     false
```

その為、より詳細な評価を行う場合には、ev 函数を用いる必要性があります。実例として、 $(x+1)^2-x^2-2x-1=0$ and $3>0$ の ev 函数による評価例を以下に示しておきます。

```
(%i7) (x+1)^2-x^2-2*x-1=0 and 3>0;
(%o7) false
(%i8) expr1:'((x+1)^2-x^2-2*x-1=0 and 3>0);
          2      2
(%o8)      (x + 1)  - x  - 2 x - 1 = 0 and 3 > 0
(%i9) ev(expr1,expand,pred);
(%o9) true
```

この ev 函数で述語を評価する場合には引数として pred を必ず指定します。他に expand,ratsimp 等のオプションを指定すると良いでしょう。

猶、 $\text{diff}(f(x),x)-f(x)=0$ and $f(x)=\exp(x)$ の様な述語の処理は、ev 函数でもまともに処理出来ません。実際、この述語は正しい記述ではありません。少し考えると判る事ですが、この式の意味する所は、 $f(x) = \exp(x) \rightarrow \text{diff}(f(x),x) - f(x) = 0$ なので、 $\text{ev}(\text{diff}(f(x),x)-f(x)=0, f(x)=\exp(x), \text{pred})$ で評価すべき式です。

そこで、以下の様に強引に演算子を定義して評価する事も可能です。

```
(%i21) infix("->");
(%o21) ->
(%i22) a ->b:=block(ev(b,a,expand,diff,pred));
(%o22) a -> b := block(ev(b, a, expand, diff, pred))
(%i23) (f(x)=sin(x)) -> '(diff(f(x),x,2)+f(x)=0);
(%o23) true
(%i24) true -> ((x+1)^2-x^2-2*x-1=0);
(%o24) true
(%i25) false -> ((x+1)^2-x^2-2*x-1=0);
(%o25) true
```

ここでの最後の二つの例は、この微分の話に限定して、数理論理学の「 A ならば B 」, 即ち、 $A \rightarrow B$ の論理演算子 $\rightarrow$ と同じ結果になる事を確認したものです。

5.5.3 述語を評価する函数

Maxima で述語を評価出来る函数としては、is 函数や maybe 函数、それに ev 函数が挙げられます。is 函数と maybe 函数は与えられた述語を評価して、真理値を返却する函

数ですが, `ev` 関数は与えられた式の展開や微分といった評価も可能な非常に高機能な関数です. これらの構文を以下に示しておきましょう.

—— 述語を評価する関数 ——

```
is(< 述語 >)
maybe(< 述語 >)
ev(< 述語 >, pred)
ev(< 述語 >, < 変数1 > = < 値1 >, ..., < 変数n > = < 値n >, pred)
ev(< 述語 >, < 変数1 > = < 値1 >, ..., < 変数n > = < 値n >,
    < オプション1 >, ..., < オプションn >, pred)
```

`is` 関数と `maybe` 関数は与えられた述語を文脈上で処理を行う関数です. これらの関数が返却する真理値集合は $\{\text{true}, \text{false}, \text{unknown}\}$ になります.

`is` 関数と `maybe` 関数は殆ど同じ処理を行います. 特に, 大域変数 `prederror` の値が `true` の場合, `is` 関数と `maybe` 関数は同じ働きをしますが, `prederror` の値が `false` であれば, `is` 関数は述語の評価の結果が `true` にならなかった時にエラー表示を行う仕様になっています.

`maybe` 関数は述語が正しい場合には `true`, 偽の場合には `false`, そして, 判定不能の場合には `unknown` を返す関数です.

`is` 関数と `maybe` 関数は与えられた述語を多少簡易化して解釈しますが, 式の展開, 微分や値の代入といった操作は一切行いません. その為, これらの関数を利用する場合には, 予め式を可能な限り目的に合せて変換しておく事と, 様々な仮定を文脈に登録しておく必要があります.

式の変換や変数の代入といった操作が行え, 述語の評価が行える関数が `ev` 関数です. この関数は引数に `pred` を指定すると, 第一引数として与えた述語の判定を行います. この時, 述語の持つ各変数に値を設定して判別する事, 式の展開や微分といった操作も可能です. この `ev` 関数の詳細については §5.8.2 を参照して下さい.

5.5.4 Maxima の真理関数

Maxima には, Maxima の式に対して `true`, `false` といった真理値を返す関数があります. ここでは, 真理値を返す関数の事を Maxima の真理関数と呼びます. 猶, 数理論理学での真理関数は真理値集合から真理値集合への関数の事で, この本の真理関数よりも狭義の関数になります.

この真理関数は Maxima には非常に多く存在し,Maxima の式がある属性を持つかどうかを判定する関数の場合は,LISP 風に末尾に p が付く事で,名前から判別可能な関数もあります.

ここでは複数の引数を取る特徴のある関数に関して解説します.

zeroequiv 関数

与えられた式が 0 に等しいかどうかを判別する関数 zeroequiv を説明しましょう.

— zeroequiv 関数の構文 —

```
zeroequiv(< 式 >, < 変数 >)
```

この zeroequiv 関数の真理値集合は {true,false,dontknow} になります. zeroequiv 関数は与えられた式が 0 に等しいかどうかを判別するだけの関数です判定不能の場合にのみ,dontknow を返します. 勿論, この関数の能力はあまり高くはありません. その為,zeroequiv に引渡す式は予め簡易化しておく必要があります.

```
(%i6) zeroequiv (sin(2*x) - sin(x)*cos(x), x);
(%o6) false
(%i7) zeroequiv (sin(2*x*y) - sin(x*y)*cos(x*y), x);
(%o7) dontknow
```

freeof 関数と lfreeof 関数

式中の変数の有無を調べられる関数として,freeof 関数と lfreeof 関数があります.

— freeof 関数と lfreeof 関数の構文 —

```
freeof(< 変数1 >, ..., < 変数n >, < 式 >)   < 変数i > が < 式 > に存在しない場合
lfreeof([< 変数1 >, ..., < 変数n >], < 式 >)   < 変数i > が < 式 > に存在しない場合
```

この freeof 関数は < 変数_i > が < 式 > の中に現われなければ true を返し, それ以外は false を返します. ここで, < 変数_i > は変数原子, 添字付けられた名前, 関数, 或いは二重引用符""で括られた演算子が扱えます.

猶,freeof 関数は,sum 関数や product 関数の内部で利用される疑似変数に対して適用する事は出来ません. 疑似変数を < 変数_i > に指定した場合, 式に疑似変数が含まれていても,true を返します.

lfreeof 関数も動作は freeof と殆ど同じものです. 但し, 変数をリストで与える点と演算子が扱えない点で異なります.

subvarp 関数

非常に形式的な関数として, subvarp 関数があります. この関数は引数が添字付けられている場合に true を返し, それ以外は 0 を返す関数です.

subvarp 関数

subvarp(< 式 >)

この関数は非常に形式的な関数で, 単純に与えられた式の形が, $a[i]$ の形式, 内部表現では $((\$A \text{ SIMP ARRAY}) \$I)$ の書式である場合に true を返し, それ以外では false を返す式です.

```
(%i1) subvarp(a[1]);
(%o1)                                     true
(%i2) subvarp(a[i]);
(%o2)                                     true
(%i3) subvarp(x^2+y+1);
(%o3)                                     false
(%i4) subvarp("いろはにほへと");
(%o4)                                     false
(%i5) a[1]:10;
(%o5)                                     10
(%i6) subvarp(a[1]);
(%o6)                                     false
(%i7) a[2]:b[100];
(%o7)                                     b
                                           100
(%i8) subvarp(a[2]);
(%o8)                                     true
```

この例で示す様に, subvarp 関数は入力式の内部表現のみに影響されます.

every 関数と some 関数

some 関数

```

every(< 真理関数 >, < リスト >)
every(< 真理関数 >, < 集合 >)
every(< 真理関数 >, < リスト1 >, ..., < リストn >)
some(< 真理関数 >, < リスト >)
some(< 真理関数 >, < 集合 >)
some(< 真理関数 >, < リスト1 >, ..., < リストn >)

```

every 関数は第一引数の関数が第二引数以降の関係を全て満す場合に true を返す関数です, この every 関数は $\forall$ に対応する関数です.

これに対し, some 関数は第一引数の関数が第二引数以降の関係を一部満す場合に true を返す関数で, some 関数は $\exists$ に対応する関数になります.

```

(%i1) every(integerp, {1,2,3,4});
(%o1)                                     true
(%i2) every(integerp, [1,2,3,4,4/5]);
(%o2)                                     false
(%i3) some(integerp, [1,2,3,4,4/5]);
(%o3)                                     true

```

every 関数も some 関数も引数が二つであれば, 第二引数はリストや集合の双方が利用可能です. 但し, 引数が 3 個以上の場合, every(f, a, b, ..., z) や some(f, a, b, ..., z) では, $f(a_i, b_i, \dots, z_i)$ を評価を行う為, 第二引数以降のリストは全て同じ長さのリストでなければなりません. 又, 第一引数に演算子を指定する場合, 演算子は二重引用符で括らなければなりません.

```

(%i6) some(">", [1,2,3,4,4/5], [0,0,0,0,0]);
(%o6)                                     true
(%i7) every(">", [1,2,3,4,4/5], [0,0,0,0,0]);
(%o7)                                     true

```

猶, 大域変数 prederror が false の場合, every 関数と some 関数の返す真理値集合は {true, false, unknown} ですが, true に切替えた場合, unknown は false に包含される為, {true, false} の二値になります.


```
(%i42) unknown(a pochi b);
(%o42)                                     false
```

identity 関数

identity 関数の構文

```
identity(< 式 > )
```

identity 関数は引数を実評価でそのまま返却する関数です。

実際、(defun \$identity (x) x) で定義された関数です。猶、この identity 関数は内部で some 関数や every 関数と組合せて多く用いられています。

charfun 関数

charfun 関数の構文

```
charfun(< 述語 > )
```

charfun 関数は与えられた述語を実評価して、述語が true の場合に 1 を返し、false の場合に 0 を返し、それ以外は名詞型を返す関数です。

```
(%i77) charfun(3>1);
(%o77)                                     1
(%i78) charfun(4<1);
(%o78)                                     0
(%i79) charfun(x>1);
(%o79)                                     charfun(x > 1)
```

5.5.5 引数が一つの真理関数

Maxima には他にも多くの真理関数があります。特に多いのが引数が一つの関数です。この様な関数は非常に多くあり、引数として真理関数に与えられる Maxima の式の属性は今後もパッケージの増加と共に増えるでしょう。その為、ここでは基本的と思われる真理関数と、真理関数が真を返す為に、その対象が満すべき条件を纏めた一覧を示す事に留め、詳細は各対象の解説で行う事とします。

型に関連する真理関数

atomp($\langle$ 式 $\rangle$)	原子の場合
numberp($\langle$ 式 $\rangle$)	数値の場合
bfloatp($\langle$ 式 $\rangle$)	bfloat 型の場合
floatnump($\langle$ 式 $\rangle$)	浮動小数点数の場合
integerp($\langle$ 式 $\rangle$)	整数の場合
ratnump($\langle$ 式 $\rangle$)	有理数の場合
ratp($\langle$ 式 $\rangle$)	CRE 表現, 或いは taylor 級数型の場合
taylorp($\langle$ 式 $\rangle$)	taylor 級数型の場合
constantp($\langle$ 式 $\rangle$)	定数の場合
scalarp($\langle$ 式 $\rangle$)	数, 定数やスカラとして宣言された変数, 数, 定数, そして, 行列やリストを含まない式の場合
nonscalarp($\langle$ 式 $\rangle$)	非スカラーの場合
matrixp($\langle$ 式 $\rangle$)	行列の場合
diagmatrixp($\langle$ 式 $\rangle$)	対角行列か二次元配列の場合
lstringp($\langle$ 式 $\rangle$)	LISP の文字列の場合
stringp($\langle$ 式 $\rangle$)	Maxima の文字列の場合

整数の性質に関連する真理関数

oddp($\langle$ 式 $\rangle$)	奇数の場合
evenp($\langle$ 式 $\rangle$)	偶数の場合
primep($\langle$ 式 $\rangle$)	素数の場合

文字に関連する真理関数

lowercasep($\langle$ 式 $\rangle$)	小文字の文字列の場合
uppercasep($\langle$ 式 $\rangle$)	大文字の文字列の場合
digitcharp($\langle$ 式 $\rangle$)	数字の場合

5.5.6 その他の関数

compare 関数

```
compare(< 式1>, < 式2>)
```

compare 関数は与えられた式を比較する演算子で、述語 $\langle \text{式}_1 \rangle \langle \text{演算子} \rangle \langle \text{式}_2 \rangle$ が真となる演算子を返す関数です。

内部的には、どちらか一方が実数でなく、両方の式が等しければ演算子= を返し、等しくない場合には notcomparable を返します。双方が実数の場合、 $\langle \text{式}_1 \rangle - \langle \text{式}_2 \rangle$ を計算して、その符号で大小関係を返し、不定の場合は、#を返し、大小関係が判らない場合や、差が計算出来ない場合には unknown を返します。

```
(%i67) compare(x^2,0);
(%o67)                                     >=
(%i68) compare(1/(1+x)^2,0);
(%o68)                                     >
(%i69) compare(1/(1+x),0);
(%o69)                                     #
(%i70) compare(1/(1+%i*x),0);
(%o70)                                     notcomparable
(%i71) compare(x,y);
(%o71)                                     unknown
```

5.6 文脈

5.6.1 文脈の概要

Maxima の文脈 (context) は問題を考える上で必要とされる様々な仮定や事実を積重ねたものです。Maxima で式の計算や簡易化といった処理は、この文脈を利用して行います。

例えば、4 の平方根は 2 になりますが、 x^2 であればどうでしょうか？ x は正か負の実数か、ひょっとすると複素数かもしれません。この様に $\sqrt{x^2}$ を考えるだけでも、 x に関して様々な情報が必要になります。まず、実数であれば $|x|$ となりますね。更に、 $x \geq 0$ であれば答は x になります。

纏めると、この処理では x に対して次の仮定を用いています：

- x は実数である。
- $x \geq 0$ である。

さて、ここでの仮定をもう少し吟味してみましょう。

まず、最初の仮定は対象 x が帰属するクラスを表現したものです。ここでもし x を複素数であると仮定すれば後の仮定に大きく影響しますね。その意味では、この仮定は大域的な仮定ですね。そして、このような性質は、その他の対象との関係を述べたものではなく、その対象の属性として表現されるものです。

次の仮定 $x \geq 0$ は二つの対象 x と 0 との関係を述べたものです。この関係は $x < 0$ としても $|x| > 1$ に置換えても最初の仮定 (属性) に影響を与えるものではありませんね。このことから、文脈は対象の性質を表現する属性と、対象の持つ何等かの関係を表現する述語で構成される事が理解されるでしょう。

次に、式を評価する上で仮定を切り替える事もよくある事です。例えば、先程の $\sqrt{x^2}$ の評価で、仮定 $x \geq 0$ を仮定 $x < 0$ に変更して式の評価を行う事が対応します。これは文脈の切替で機械的に行えますね。例えば、文脈 A を $x \geq 0$ 、文脈 B を $x < 0$ としましょう。すると、仮定 $x \geq 0$ で式の評価を行う場合には文脈 A を指定し、仮定 $x < 0$ で評価を行いたければ文脈 B を指定すれば良いのです。

では、式 $\sqrt{x^2 y^2}$ の評価はどのように考察しているのでしょうか？通常、次の 4 つの場合に分けて考察していますね：

1. $x \geq 0 \wedge y \geq 0$
2. $x \geq 0 \wedge y < 0$

$$3. x < 0 \wedge y \geq 0$$

$$4. x < 0 \wedge y < 0$$

従って、文脈もこの場合分けに応じた四種類の文脈が与えられます。ここで実際に、この様な式が与えられた時、最初から四種類の場合分けで考察するでしょうか？試行錯誤して考察するのであれば、最初に $x \geq 0$ と $x < 0$ に分類し、今度は、 $y \geq 0, y < 0$ に分けて考えるでしょう：

$$1. x \geq 0 \begin{cases} y \geq 0 & \Rightarrow \text{文脈 1} \\ y < 0 & \Rightarrow \text{文脈 2} \end{cases}$$

$$2. x < 0 \begin{cases} y \geq 0 & \Rightarrow \text{文脈 3} \\ y < 0 & \Rightarrow \text{文脈 4} \end{cases}$$

この事は、文脈には階層構造を持った方が実用的である事を示唆しています。実際、Maxima の文脈は階層構造を持ったものです。

Maxima の文脈は、Maxima の対象としては記号であり、この記号に対して述語や属性を登録した対象になります、そして、文脈の階層構造は記号の `subc` 属性として与えられています。

猶、文脈に登録した述語や属性を $P_1, \dots, P_n$ とすると、標準形 $P_1 \wedge \dots \wedge P_n$ で式の評価を行います (§4.12.7 参照)。その為、登録する述語は $A_1 \vee \dots \vee A_m$ の形式は許されません。基本的に $\vee$ を反映させる為には、文脈を分けて考える必要があります。

5.6.2 文脈に登録可能な述語

Maxima の文脈に登録可能な述語は次の二項関係を表現した述語に限定されます：

—— 文脈に登録可能な述語例 ——

数学の記号	Maxima の演算子/函数	例
=	<code>equal</code>	<code>equal(x,y)</code>
≠	<code>notequal</code>	<code>notequal(x,y)</code>
≥	<code>>=</code>	<code>x>=y</code>
>	<code>></code>	<code>x>y</code>
≤	<code><=</code>	<code>x<=y</code>
<	<code><</code>	<code>x<y</code>

ここで、内部的に $\leq$ の演算子項と $<$ の演算子項は被演算子の左右の入替えによって $\geq$ の演算子項と $>$ の演算子項で置換えられて登録されます。

更に、同値性と非同値性を示す演算子として、Maxima には演算子 $=$ と演算子 $\#$ がありますが、これらの演算子を用いた述語は Maxima の文脈には登録出来ない仕様となっています。その為、文脈に登録する為に演算子項 $x=y$ を関数項 `equal(x,y)`、演算子項 $x\#y$ を関数項 `notequal(x,y)` に置換える必要があります。猶、Maxima では関数項 `not(equal(x,y))` は `notequal(x,y)` に、関数項 `not(notequal(x,y))` は `equal(x,y)` に自動的に置換えられます。

5.6.3 述語の登録

文脈上で述語操作を行う関数を次に纏めておきます：

—— 文脈上の述語に関連する関数 ——

```
assume(〈述語1〉, 〈述語2〉, …)
forget(〈述語1〉, …, 〈述語n〉)
forget([〈述語1〉, …, 〈述語n〉])
facts(〈事項〉)
facts(〈文脈〉)
facts()
```

Maxima に事実や仮定を教える関数は `assume` 関数です。`assume` 関数によって、述語は文脈の `data` 属性の属性値として蓄積されます。猶、`assume` 関数は適切でない式や述語が入力された場合、`assume` 関数はエラー表示やリストを返します。特に、演算子 or を含む述語に対して演算子 or を含む述語の処理が出来ないと表示し、同値演算子 $=$ や非同値の演算子 $\#$ を含む述語が入力された場合には `equal` 関数や `notequal` 関数を使う様にと表示します。

更に、既に登録された述語が入力された場合、リストの形式で対応する述語を `redundant` にして返します。式が変数や文字列のみの場合は、`meaningless` を返し、真理関数を含む述語に対しては `inconsistent` を返します。

有効な述語は `assume` 関数によって現行の文脈上に設定されます。これに対し、教えた事実や仮定を忘れさせる、即ち、文脈に設定した述語を消去する場合、`forget` 関数を用います。そして、或る事項に関して、どの様な述語が含まれているかを調べたければ、`facts` 関数を用います。`facts` 関数は文脈を指定した場合、その文脈に含まれる述語を返します。何も指定しない `facts()` を入力した場合、現行の文脈が保持する述語を全て

表示します.

では `assume` 函数を用いて文脈上に述語を設定し, その述語を用いて式を簡易化する例と文脈の切り替えによる効果を見ましょう:

```
(%i6) assume(y>0)$
(%i7) assume(x>0,z<0)$
(%i8) facts();
(%o8) [y > 0, x > 0, 0 > z]
(%i9) sqrt(x^2);
(%o9) x
(%i10) context:initial;
(%o10) initial
(%i11) sqrt(x^2);
(%o11) abs(x)
(%i12) facts();
(%o12) []
(%i13) activate(neko);
(%o13) done
(%i14) context;
(%o14) initial
(%i15) sqrt(x^2);
(%o15) x
(%i16) facts();
(%o16) []
(%i17) deactivate(neko);
(%o17) done
(%i18) sqrt(x^2);
(%o18) abs(x)
(%i19) killcontext(neko);
(%o19) done
(%i20) contexts;
(%o20) [mike, initial, global]
```

先程の文脈の定義に続いて, 文脈 `neko` で `assum` 函数を使って変数 `x,y,z` に対して仮定, 即ち, 述語を設定します. 利用している文脈で設定した述語全体と対象の属性は `facts();` で表示が出来ます. 文脈 `neko` で `abs(x^2)` を実行すると, 述語 `x>0` より `x`

が返されます. 次に, 文脈の切替は変数 `context` に文脈名を割当てる事で行います. この例では, 文脈 `neko` から文脈 `initial` へと現行の文脈を `context:initial;` で変更しています. それから, `sqrt(x^2)` を実行すると `abs(x)` が返されます. これは何故でしょうか? ここで用いた文脈 `initial` には $x > 0$ という情報が無いからです. 実際, `facts();` を実行すると, 文脈 `initial` には何も述語が無い事が分ります. ここで, 文脈 `neko` の述語を文脈 `initial` で利用したければ, `activate(neko);` を実行します. その結果, `sqrt(x^2)` の結果が x になります. 但し, 文脈 `initial` の内容には変化がありません. 不要になった文脈は `deactivate` 関数で無効にしたり, `killcontext` 関数で削除する事が可能です.

次に, 大域変数 `features` には Maxima に組込まれた属性のリストが割当てられています. Maxima の式に `features` のリストに含まれる属性を持たせたければ, `declare` 関数を用います.

例えば `birthday` を整数として宣言する事で, 整数属性を持たせたければ, `declare(birthday,integer);` とします. 逆に式 p が属性 q を持つかどうかを調べる場合は `featurep` 関数を使って, `featurep(p,q);` で調べられます.

猶, `declare` 関数による宣言は, `assume` 関数と違い大域的なものになります. その一方で, `featurep` 関数は `declare` 関数を用いた文脈上でしか使えません. 詳細は, 5.4.2 節を参照して下さい.

```
(%i1) newcontext("mike")$
(%i2) supcontext("neko","mike")$
(%i3) context:mike$
(%i4) declare(bb,lassociative)$
(%i5) assume(x>0)$
(%i6) facts();
(%o6) [kind(bb, lassociative), x > 0]
(%i7) bb(bb(a,b),bb(c,d));
(%o7) bb(bb(bb(a, b), c), d)
(%i8) sqrt(x^2);
(%o8) x
(%i9) context:initial;
(%o9) initial
(%i10) bb(bb(a,b),bb(c,d));
(%o10) bb(bb(bb(a, b), c), d)
(%i11) aa(aa(a,b),aa(c,d));
(%o11) aa(aa(a, b), aa(c, d))
```



```
(%i12) facts();
(%o12) [kind(kron_delta, symmetric)]
(%i13) sqrt(x^2);
(%o13) abs(x)
```

この例では、文脈 `mike` と `neko` を生成し、`context:mike;` で文脈をデフォルトの文脈 `initial` から文脈 `mike` に切替えています。そこで、`bb` に左結合律が成立する事を宣言し、`x>0` も `assume` 関数で文脈 `mike` に入れています。`facts` 関数で文脈に蓄積した述語を確認し、`bb(bb(a,b),bb(c,d))` が左結合律によって、`bb(bb(bb(a,b),c),d)` に自動的に変形される事と、`sqrt(x^2)` が `x` に簡易化される事を確認しています。それから、`context:initial;` で文脈 `initial` に移動します。

ここで、`declare` 関数による宣言は `assume` 関数による述語とは違い、大域的な影響を及ぼす為、`bb` に関しては左結合律が成立します。その一方で、述語 `x>0` は文脈 `initial` の親文脈 `mike` 上の述語である為、文脈 `initial` では影響を受けません。その為、`sqrt(x^2)` は `abs(x)` になります。更に、`bb` に与えた属性は、文脈 `initial` 上で `facts()` を実行しても一切表示されません。

最後に、文脈で多く設定されるものの一つが変数の正値性です。この正値性の設定では通常、`assume` 関数を用いて設定を行いますが、大域変数を用いて `assume` 関数による設定を省く事も可能です。

5.6.4 文脈内部での属性と述語の表現

今度は、Maxima の関係を表現する述語がどの様に内部で表現されるかを観察してみましょう：

```
(%i1) assume(x1>y1);
(%o1) [x1 > y1]
(%i2) assume(x1<z1);
(%o2) [z1 > x1]
(%i3) assume(x1<=w1);
(%o3) [w1 >= x1]
(%i4) assume(equal(a1,a2));
(%o4) [equal(a1, a2)]
(%i5) assume(notequal(a1,a4));
(%o5) [notequal(a1, a4)]
```

```
(%i6) :lisp (get '$initial 'data)
(((MNQP $A1 $A4) CON $INITIAL) ((MEQP $A1 $A2) CON $INITIAL)
 ((MGQP $W1 $X1) CON $INITIAL) ((MGRP $Z1 $X1) CON $INITIAL)
 ((MGRP $X1 $Y1) CON $INITIAL))
```

不等号を持つ述語は演算子”>=”と演算子”>”に置換えられている事が判ります. この述語は assume 関数によって文脈に登録されますが, この登録は LISP の属性リストが使われます. その為, 述語は文脈の data 属性の属性値リストに追加されて, putprop 関数を用いて登録されます.

従って, LISP 側から文脈に登録した述語を調べたければ, get 関数を用いて取出す事が可能です. さて, 文脈の data 属性値は登録した述語に対応する S 式のリストとなっています.

例えば, $x1 \leq w1$ は被演算子の並び換えの後に $w1 \geq x1$ として登録され, 内部的には ((MGQP \$W1 \$X1) CON \$INITIAL) で表現されています. この S 式の第一成分 (MGQP \$W1 \$X1) が, 入力した述語に直接対応します. この第一成分は Maxima の式 $w1 \geq x1$ の内部表現と比較し, 先頭の関数名の先頭に \$ が無く, 関数名を括る () も無い平坦なリストとして表現されています. そして, 第二成分が CON で文脈に関連する S 式である事を示し, 第三成分が文脈名 (Maxima 側から見ると文脈 initial) になっています.

では, 今度は declare 関数を用いて対象に幾つかの属性を指定してみましょう:

```
(%i6) declare(x1,odd)$
(%i7) declare(a1,complex)$
(%i8) :lisp (get '$initial 'data)
(((KIND $A1 $COMPLEX) CON $INITIAL) ((KIND $X1 $ODD) CON $INITIAL)
 ((MNQP $A1 $A4) CON $INITIAL) ((MEQP $A1 $A2) CON $INITIAL)
 ((MGQP $W1 $X1) CON $INITIAL) ((MGRP $Z1 $X1) CON $INITIAL)
 ((MGRP $X1 $Y1) CON $INITIAL))
(%i8) facts();
(%o8) [x1 > y1, z1 > x1, w1 >= x1, equal(a1, a2), notequal(a1, a4),
      kind(x1, odd), kind(a1, complex)]
```

declare 関数で対象に属性を指定すると, 述語側の第一成分が KIND になっています. 次に facts 関数を使って文脈に登録されている情報を全て表示させていますが, 属性値のリストが Maxima の式に変換されて表示され, 逆順で表示されている事が判ります. ここで, 述語や属性の内部表現を次に纏めておきましょう:

文脈内部での述語と属性の表現

表現名	内部表現	対応する述語表現	概要
kind	(kind x y)	kind(x,y)	含意 ($\rightarrow$) を表現
par	(par x y)	par(x,y)	含意 ($\rightarrow$) を表現 (Maxima 側から定義出来ない)
mgrp	(mgrp x y)	$x > y$	大小関係 ($>$) を表現
mgrpq	(mgrpq x y)	$x \geq y$	大小関係 ($\geq$) を表現
meqp	(meqp x y)	equal(x,y)	同値関係を表現
mnqp	(mnqp x y)	notequal(x,y)	非同値関係を表現

文脈に登録される述語は、Maxima の概念間や対象と概念間の含意 $a \rightarrow b$ に対応する kind(a,b), 互いに独立した概念 a, b ($\Leftrightarrow a \cap b = \emptyset$) の連言 $a \vee b \equiv c$ から得られる二つの含意 $a \rightarrow c$ と $b \rightarrow c$ に対応する par(a,c), par(b,c), そして, 同値性と非同値性, それと大小関係が平坦なリストとして表現されます. ここで, 内部関数 par を利用者が設定する事は出来ません. Maxima の初期状態では予め登録された事柄 (§5.4.6 参照) に限定されています. 自由に利用可能なのは内部関数 kind で設定可能な属性のみです.

文脈を用いた判断では, 文脈に登録された述語表現を Modus Ponens(MP) (§4.12.3 参照) を用いて推論を行う事, より具体的には述語表現リストの結合処理で行えます.

例えば, 仮定として, $x_1 > x_2, x_2 > x_3, \dots, x_{n-1} > x_n$ が文脈に与えられていた時, $x_i > x_j$ の判断は, x_i から出発して, 文脈に登録された $>$ に対応する mgrp を持つ述語表現を用いて x_j に到達出来るかどうかで行います. 最初に, x_i を第二項に持つ mgrp の述語を文脈の data 属性値の述語リストの先頭から順番に捜します. ここで, 述語 (mgrp x_i x_{i_1}) が該当する述語であれば, その第三項の x_{i_1} が x_j と一致するか確認します. ここで一致すれば真であると判断しますが, 一致しなければ, 今度は $x_{i_1} > x_j$ として x_{i_1} に対して該当する述語を捜します. 以下, 同様に処理を行って, 項の列 $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ が得られますが, この列が最終的に x_j に到達出来れば true と判断します. この操作は, $x_i > x_{i_1}, x_{i_1} > x_{i_2}, \dots, x_{i_{k-1}} > x_{i_k}$ に MP を適用して $x_i > x_{i_k} = x_j$ を得る事に対応しています. もし, x_j に到達出来なければ, 再び, x_i を第二項に持つ mgrp の述語を検出し, 同様の処理を行います. そして, x_i に到達出来なければ, 今度は大小関係を $<$ に反転させて, x_i から x_k に到達出来るかを試みます. ここで関係 $<$ に対応するのが関数 dlsf です. これに成功すると false と判断しますが, こちらでも最終的に到達出来なければ, 判断不能, 即ち, unknown と判断します. 具体的な例を次に示しておきましょう:

```
(%i1) assume(x1>x2,x2>x3,x3>x4,x4>x5,x5>x6,x6>x7);
```

```

(%o1)      [x1 > x2, x2 > x3, x3 > x4, x4 > x5, x5 > x6, x6 > x7]
(%o1):lisp (trace dlqf dgqf dlsf dgrf deq dcompare)
WARNING: TRACE: redefining function DLQF in top-level, was defi
      /usr/local/maxima-5.15.0/src/binary-clisp/compar.fas
;; Tracing function DLQF.
WARNING: TRACE: redefining function DGQF in top-level, was defi
      /usr/local/maxima-5.15.0/src/binary-clisp/compar.fas
;; Tracing function DGQF.
WARNING: TRACE: redefining function DLSF in top-level, was defi
      /usr/local/maxima-5.15.0/src/binary-clisp/compar.fas
;; Tracing function DLSF.
WARNING: TRACE: redefining function DGRF in top-level, was defi
      /usr/local/maxima-5.15.0/src/binary-clisp/compar.fas
;; Tracing function DGRF.
WARNING: TRACE: redefining function DEQ in top-level, was defin
      /usr/local/maxima-5.15.0/src/binary-clisp/compar.fas
;; Tracing function DEQ.
WARNING: TRACE: redefining function DCOMPARE in top-level, was
      /usr/local/maxima-5.15.0/src/binary-clisp/compar.fas
;; Tracing function DCOMPARE.
(DLQF DGQF DLSF DGRF DEQ DCOMPARE)
(%i2) is(x2>x5);
1. Trace: (DCOMPARE '$X2 '$X5)
2. Trace: (DEQ '$X2 '$X5)
3. Trace: (DGRF '$X3 '$X5 '((MGRP $X3 $X4) CON $INITIAL))
4. Trace: (DGRF '$X4 '$X5 '((MGRP $X4 $X5) CON $INITIAL))
4. Trace: DGRF ==> T
3. Trace: DGRF ==> T
2. Trace: DEQ ==> T
1. Trace: DCOMPARE ==> $POS
(%o2)                                     true

```

ここでの例では、 $x_1 > x_2 > \dots > x_6 > x_7$ を仮定し、 $x_2 > x_5$ を判断している例です。内部関数 `deq` と内部関数 `dgrf` によって述語を用いた推論が実行されています。この例では最終的に x_5 に到達した為、`T` を返しています。ここで、`dgrf` が大小関係の `>` に対応

する推論を行う内部函数です. ここで, 内部函数 dcompare 函数が pos を返していますが, Maxima で $x > y$ の判断は $x - y$ の正值性で行う為です. ここでの例では, 内部函数 dcompare が pos を返した為, is 函数は true を返却しています.

さて, 同じ文脈上で今度は $x_5 > x_2$ の判断をさせてみましょう:

```
(%i3) is(x5>x2);
1. Trace: (DCOMPARE '$X5 '$X2)
2. Trace: (DEQ '$X5 '$X2)
3. Trace: (DGRF '$X6 '$X2 '((MGRP $X6 $X7) CON $INITIAL))
4. Trace: (DGRF '$X7 '$X2 '((MGRP $X6 $X7) CON $INITIAL))
4. Trace: DGRF ==> NIL
3. Trace: DGRF ==> NIL
3. Trace: (DGRF '$X6 '$X2 '((MGRP $X5 $X6) CON $INITIAL))
3. Trace: DGRF ==> NIL
3. Trace: (DLSF '$X4 '$X2 '((MGRP $X4 $X5) CON $INITIAL))
3. Trace: DLSF ==> NIL
3. Trace: (DLSF '$X4 '$X2 '((MGRP $X3 $X4) CON $INITIAL))
4. Trace: (DLSF '$X3 '$X2 '((MGRP $X3 $X4) CON $INITIAL))
4. Trace: DLSF ==> NIL
4. Trace: (DLSF '$X3 '$X2 '((MGRP $X2 $X3) CON $INITIAL))
4. Trace: DLSF ==> T
3. Trace: DLSF ==> T
2. Trace: DEQ ==> T
1. Trace: DCOMPARE ==> $NEG
(%o3)                                     false
```

この例では $x_5 > x_2$ の判断を行っています. この例では, 最初に x_5 から出発して x_2 に到達する事に失敗しています. そこで, 内部函数 dgrf は NIL を返しています. そこで, 今度は関係 $>$ ではなく, 関係 $<$ に対応する内部函数 dlsf に切り換え, 文脈に登録された述語の第二項を取出した列で x_2 に到達出来るかを検証しています. その結果, 今度は x_2 に到着した為, 内部函数 dlsf は T を返しています. その事から, 内部函数 dcompare は $x_5 - x_2$ が負であると判断し, その結果から is 函数は false を返します.

次に, 同値性はどの様に推論しているのでしょうか? これは大小関係と同様に, 差を取って符号の判定を行う手法ですが, 文脈から項の同値性に関する述語を与式に ratsubst 函数を用いて代入し, 与式が 0 になるかどうかで判定しています.

```
(%i2) assume(equal(x1,x2),equal(x2,x3),equal(x3,x4),equal(x4,x5));
(%o2) [equal(x1, x2), equal(x2, x3), equal(x3, x4), equal(x4, x5)]
(%i3) :lisp (trace alike $ratsubst)
WARNING: TRACE: redefining function ALIKE in top-level, was defined in
        /usr/local/maxima-5.15.0/src/binary-clisp/simp.fas
;; Tracing function ALIKE.
WARNING: TRACE: redefining function $RATSUBST in top-level, was defined in
        /usr/local/maxima-5.15.0/src/binary-clisp/nrat4.fas
;; Tracing function $RATSUBST.
(ALIKE $RATSUBST)
(%i3) is(equal(x2,x5));
1. Trace: (ALIKE '($X2) '($X5))
1. Trace: ALIKE ==> NIL
1. Trace: (ALIKE '($X3 $X4) '($X4 $X5))
1. Trace: ALIKE ==> NIL
1. Trace: (ALIKE '($X2 $X3) '($X3 $X4))
1. Trace: ALIKE ==> NIL
1. Trace: (ALIKE '($X2 $X3) '($X4 $X5))
1. Trace: ALIKE ==> NIL
1. Trace: (ALIKE '($X1 $X2) '($X2 $X3))
1. Trace: ALIKE ==> NIL
1. Trace: (ALIKE '($X1 $X2) '($X3 $X4))
1. Trace: ALIKE ==> NIL
1. Trace: (ALIKE '($X1 $X2) '($X4 $X5))
1. Trace: ALIKE ==> NIL
1. Trace: ($RATSUBST '$X2 '$X1 '((MPLUS SIMP) $X2 ((MTIMES SIMP RATSIM) -1 $X5)))
2. Trace: (ALIKE '($X1 $X2 $X5) '($X1 $X2 $X5))
2. Trace: ALIKE ==> T
1. Trace: $RATSUBST ==> ((MPLUS RATSIM) ((MTIMES RATSIM) -1 $X5) $X2)
1. Trace: ($RATSUBST '$X3 '$X2 '((MPLUS RATSIM) ((MTIMES RATSIM) -1 $X5) $X2))
2. Trace: (ALIKE '($X3 $X2 $X5) '($X3 $X2 $X5))
2. Trace: ALIKE ==> T
1. Trace: $RATSUBST ==> ((MPLUS RATSIM) ((MTIMES RATSIM) -1 $X5) $X3)
1. Trace: ($RATSUBST '$X4 '$X3 '((MPLUS RATSIM) ((MTIMES RATSIM) -1 $X5) $X3))
2. Trace: (ALIKE '($X4 $X3 $X5) '($X4 $X3 $X5))
```

```

2. Trace: ALIKE ==> T
1. Trace: $RATSUBST ==> ((MPLUS RATSIMP) ((MTIMES RATSIMP) -1 $X5) $X4)
1. Trace: ($RATSUBST '$X5 '$X4 '((MPLUS RATSIMP) ((MTIMES RATSIMP) -1 $X5) $X4))
2. Trace: (ALIKE '($X4 $X5) '($X4 $X5))
2. Trace: ALIKE ==> T
1. Trace: $RATSUBST ==> 0
(%o3)                                     true

```

猶,Maxima は assume 関数によって述語が文脈に登録される時点でも, 登録した述語で矛盾や再登録が無いかを, これに似た方法で検証します. 即ち, 登録する述語を検証して unknown であれば登録し, そうでなければ登録を行いません.

この様に, 文脈を用いた推論では, 尻取りの様な方法, 即ち, Modus Ponens による推論が行なわれているのです.

5.6.5 文脈の階層

Maxima の文脈には階層構造が入ります. この階層構造は親子関係に似たもので, 親側を上層とする構造です. 内部的に, この階層構造は属性 subc で与えられます.

ここで文脈 global が最下層の文脈です. Maxima を立ち上げた時には既に文脈 global の親文脈 initial があります. 猶, 初期値の文脈はこの initial です.

Maxima の起動時に存在する文脈

文脈名	概要
initial	Maxima を立ち上げた時点で利用される文脈名.
global	Maxima の文脈全体の親文脈.

Maxima では文脈を複数持たせる事が可能です. この事は, 文脈 A では $x > 0$ を仮定して述語を蓄積し, 文脈 B では $x < 0$ を仮定して述語を文脈 A とは独立して蓄積させる事が可能です. Maxima では, 文脈を新たに生成したり, 既存の文脈の間に親子関係を入れる等の操作が行えます.

文脈操作函数の一覧を次に示します:

文脈操作函数

```

activate(〈文脈1〉,...)
deactivate(〈文脈1〉,...)
killcontext(〈文脈1〉,...)
newcontext(〈文脈〉)
supcontext(〈親文脈〉,〈子文脈〉)

```

先ず, 既存の文脈を有効にする場合, activate 函数を用い, 逆に無効にする場合は deactivate 函数を用います.

新しい文脈の生成は newcontext 函数を用います. この時, 新規に生成した文脈は文脈 global の親文脈になります. 猶, 〈文脈名〉が既存の文脈でない時は, `context : 〈文脈名〉` で newcontext(〈文脈名〉) と同じ効果が得られます.

文脈 global 以外の既存の文脈の親文脈を生成する場合は supcontext 函数を用います. newcontext 函数と supcontext 函数で新規に生成された文脈は, 後述の大域変数 contexts に割当てられたリストに登録されます.

```

(%i1) supcontext(ペット, initial)$
(%i2) supcontext(猫, ペット)$
(%i3) supcontext(性格, 猫)$
(%i4) assume(equal(みけ, 穏やか))$
(%i5) assume(equal(たま, 不思議と目立たない))$
(%i6) assume(equal(とら, 名前どおりヤンチャ))$
(%i7) facts();
(%o7) [equal(みけ, 穏やか), equal(たま, 不思議と目立たない), equal(とら, 名前どおりヤンチャ)]
(%i8) :lisp (get '$猫 'subc)
($ペット)
(%i8) :lisp (get '$性格 'subc)
($猫)
(%i8) :lisp (get '$性格 'data)
((MEQP $とら $名前どおりヤンチャ) CON $性格)
((MEQP $たま $不思議と目立たない) CON $性格) ((MEQP $みけ $穏やか) CON $性格)

```

この例では, supcontext 函数を使って, 文脈 initial の直上の文脈「ペット」, そして, 文脈「ペット」の直上の文脈「猫」, 最後に文脈「猫」の上の文脈「性格」を生成して

います. それから, `assume` 函数を用いて述語を文脈「性格」に登録しています. ここで文脈に登録した述語は `facts` 函数を用いて表示可能です.

更に, `:lisp` 演算子を用いて LISP の `get` 函数で, 文脈「猫」や文脈「性格」の `subc` 属性値が文脈「ペット」である事を示し, 最後に文脈「性格」の `data` 属性に今迄登録した属性がリストとして登録されている事を示しています.

次に, 文脈の削除は `killcontext` 函数で行います. 但し, 利用中の文脈や文脈が有効 (`activate`) の場合は削除が出来ません.

5.6.6 文脈の指定に関連する大域変数

デフォルトの文脈は `initial` ですが, 文脈を生成して行くと, 現在利用中の文脈が何で, どのような文脈があるのか判らなくなるかもしれません. Maxima の大域変数 `context` に現在, 利用中の文脈名が割当てられ, 大域変数 `contexts` に Maxima の中にある全ての文脈名のリストが割当てられています.

文脈の指定に関連する大域変数

変数名	初期値	概要
<code>context</code>	<code>initial</code>	現行の文脈名が割当てられた変数
<code>contexts</code>	<code>[initial, global]</code>	Maxima 内部の文脈名リストが割当てられた変数
<code>activecontexts</code>	<code>[]</code>	<code>active</code> 函数で有効にされた文脈を成分とするリストが割当てられた変数

まず, どのような文脈が Maxima に存在するかは `contexts;` と入力すれば文脈の一覧がリストの形式で表示されます. 例えば, Maxima を立ち上げた時点では, 文脈 `initial` と文脈 `global` の二種類の文脈がある事が判り, `context;` と入力すれば, 文脈 `initial` が現行の文脈, 即ち, `assume` 函数等で設定した述語が文脈 `initial` に蓄積される事が判ります.

```
(%i1) contexts;
(%o1) [initial, global]
(%i2) context;
(%o2) initial
```

複数の文脈が存在する場合に文脈を切替えたければ, 大域変数 `context` に文脈名を直接指定します. 猶, `context` に指定した文脈が既存の文脈でなければ, Maxima は `newcontext` 函数を用いて `context` に指定した記号の文脈を新規に生成します.

大域変数 `activecontexts` には `activate` 関数を用いて有効にされた文脈が登録されたリストが割当てられています. このリストに登録された文脈に対し, `deactivate` 関数を用いると, 大域変数 `activecontexts` のリストから削除されます.

では, 以下に文脈の使い方の実例を示しましょう.

```
(%i1) contexts;
(%o1) [initial, global]
(%i2) context;
(%o2) initial
(%i3) newcontext(mike);
(%o3) mike
(%i4) supcontext(neko,mike);
(%o4) neko
(%i5) context;
(%o5) neko
```

この例では, 最初に立ち上げた時点で Maxima が持っている文脈を `contexts` 関数で表示しています. 次に `context` 関数を使って最初に用いている文脈が文脈 `initial` であることを示しています. それから, `newcontext` 関数を使って新しい文脈 `mike` を生成しています. そして文脈 `mike` の親文脈となる文脈 `neko` を今度は `supcontext` 関数を使って生成しています. この `supcontext` 関数には子文脈に既存の文脈を指定しなければなりません.

5.6.7 変数の正値性に関連する大域変数

変数の正値性に関連する大域変数

変数名	初期値	概要
<code>assume_pos</code>	<code>false</code>	<code>assume_pos_pred</code> で指定した真理関数が <code>true</code> となる対象を正値とする為のフラグ
<code>assume_pos_pred</code>	<code>false</code>	真理関数を指定

大域変数 `assume_pos` と大域変数 `assume_pos_pred` と対で用いる大域変数で, 引数の正値性の評価で用いられます. 仕組は内部処理で式の正値性の判定で用いられる内部の LISP 関数 `sign-any` で大域変数 `assume_pos` が `true` の時にのみ, 大域変数 `assume_pos_pred` に指定した真理関数を用いて引数の評価を行います. この `sign-any` 関

数は大域変数 `assume_pos_pred` に指定した真理関数が `true` を返す場合に内部変数 `sign` に値として `pos` を割当てます. 但し, `assume` 関数による設定がある場合, その `assume` 関数の設定が大域変数 `assume_pos` よりも優先されます.

```
(%i13) declare(aa,even)$
(%i14) featurep(aa,even);
(%o14)                                     true
(%i15) assume_pos_pred:lambda([x],featurep(x,even));
(%o15)                                     lambda([x], featurep(x, even))
(%i16) assume_pos:true$
(%i17) sqrt(aa^2);
(%o17)                                     aa
(%i18) sqrt(bb^2);
(%o18)                                     abs(bb)
```

この例では変数 `aa` に偶数として属性を与えています. 実際, `featurep` 関数による検査では `true` になります. ここで, 大域変数 `assume_pos_pred` に `featurep` 関数による検査を行う関数を割当てます. それから, 大域変数 `assume_pos` を `true` に設定すると, 偶数としての属性を持つ変数 `aa` には正值性が付与されます. その為, `sqrt(aa^2)` は `aa` になりますが, 偶数としての属性を持たない変数 `bb` は `abs(bb)` となります.

5.7 規則と式の並びについて

5.7.1 規則の概要

Maxima は単純に式を纏めたり展開する事で式を変形するだけではなく、函数や変数に規則というものを設定する事で、利用者が指定した式の並びに対して変換操作を行う事が可能です。

この規則を用いた処理は、与式の項を別の式で置換えるもので、代入の一種とも言えます。但し、通常の代入は項や部分式を直接指定しますが、規則を用いた処理では、雛形となる項を指定し、その雛形となる項に合致した与式の項を別の式に変換する処理です。即ち、Maxima の規則の実体は述語 P と述語 P に対応する Maxima の項の変換函数 f_P で構成されています。ここで函数 f_P は $P(t)$ が真となる Maxima の項 t に対し、指定された式の変換を行い、 $P(t)$ が偽になる項 t に対しては変換を行わない様な Maxima の項を Maxima の式に対応させる函数になります。そして、規則の適用は項 $t_1, \dots, t_n$ で構成された Maxima の式 $E(t_1, \dots, t_n)$ を $E(f_P(t_1), \dots, f_P(t_n))$ で置換える処理と言い換えられます。

では、述語と変換函数をどの様に設定すべきなのでしょう？ここでは微分演算子 D の定義を例に一寸考えてみましょう。先ず、前置式表現の演算子 D が微分演算子と呼ばれる為には以下の条件を満たさなければなりません。

- 線形性:

$$D(a + b) = Da + Db$$

- Leibniz 則:

$$D(ab) = (Da)b + a(Db)$$

Maxima で前置式表現の演算子 D を定義した場合、これを微分演算子として使う為には、線形性と Leibniz 則の二つを Maxima に教えてやらなければなりません。この性質を全て規則として与える事も考えられなくもありませんが、一般的な演算子の性質の多くは Maxima に予め定義された属性として与えられています。実際、線形性は `linear` 属性を演算子 D に `declare` 函数を用いて付加する事で対処可能です。この様に規則を定義する際には、予め Maxima に使える属性があるかどうかを確認し、無い場合に規則として定義する事になります。

ここでは演算子 D の線形性は `declare` 函数を用いて `linear` 属性を与えれば良いので、残りの Leibniz 則だけを規則として与えれば良い事になります。そこで、二つの Maxima の式 a と b の積 ab に演算子 D を作用させると、 $(Da)b + a(Db)$ という公式 (置換) を規則にすれば良いのです。この規則を *Leibniz*: $D(a\ b) \rightarrow (D\ a)\ b + a\ (D\ b)$ と表

記しましょう. この時, ab の様に規則を適用する式の事を式の並び (パターン) と呼びます. そして, 式が与えられた時, この式の並びに合致する項を検出する操作の事を, 並びの照合 (パターンマッチング) と呼びましょう.

ところで, Maxima の規則は述語とそれに対応する変換函数で構成されていると述べました. 上の例では, 変換函数は $D(a b) \rightarrow (D a) b + a (D b)$ を実現する函数となります. 人間が公式を適用する場合は, 適用可能な式の並びを検出して, 適合する式の並びに対して公式を当て嵌めますね. ですから, 雛形さえ与えてしまえば良さそうに見えますが, 述語が何故必要なのでしょう?

ここで演算子 D を変数 x に対する微分 $\frac{d}{dx}$ と考えてみましょう. この場合, 演算子 D を作用させる式が x の函数でなければ 0 になります. ところが, 変換式 $D(a b) \rightarrow (D a) b + a (D b)$ だけであれば困った事が生じます. 実際, $1 = 1 \cdot 1$ という関係を間違えて式 $\frac{d1}{dx}$ の計算で利用してしまうと, $\frac{d1}{dx}$ は $\frac{d(1 \cdot 1)}{dx}$ になります. ここで定義した Leibniz 則を適用すると $\frac{d1}{dx} \cdot 1 + 1 \cdot \frac{d1}{dx}$ となり, 以下同様に処理を実行してしまえば, この様に単純な計算でも終らなくなり, 単純な式でも非常に危険な事態に陥る可能性があるのです.

その上, 実用上の問題もあります. 規則を定義しても実際の式には規則を適用すべき項が埋れているので, 与えられた式を検査する必要もあります. 例えば, 函数 f を周期 ω_0 を持つ周期函数とし, $\forall n \in \mathbb{Z} [f(x + n\omega_0) \rightarrow f(x)]$ といった規則を入れたとします. そこで, $f(a_0)$ と $f(a)$ の関係を調べる必要が出た場合, $\exists n_0 \in \mathbb{Z} [a = a_0 + n_0\omega_0]$ であるかどうかを検出する必要がありますね. この様な理由から, 規則を述語と変換函数の対として考える必要があるのです.

ここで, Maxima の規則の設定には三種類の方法があります. 第一の方法が, `tellsimp` 函数と `tellsimpafter` 函数を用いる方法, 第二の方法が `defrule` 函数を用いる方法で, 第三の方法が `let` 函数を用いる方法です. 先ず, `tellsimp` 函数と `tellsimpafter` 函数を用いた場合, Maxima は入力された Maxima の式が条件に合致する場合, 自動的に規則の適用を行います. それに対し, `defrule` 函数や `let` 函数を用いた場合, 規則の適用は式の入力と同時に自動的にされず, 別途, 規則の適用を行う函数を用いなければなりません. 更に, `defrule` 函数では個別の規則を定義し, 規則を適用する函数もその規則を帰納的に式に作用させる事で, 式の変換を行います. そして, `let` 函数を用いると, 複数の規則を含むパッケージが構築され, `let` 函数で定義した規則を適用する函数は, このパッケージに含まれる規則を順番に処理して式の変換を実行します.

猶, 定義した規則を実際の式に適用する函数, 規則の表示, 及び削除を行う函数は, これらの系統毎に異なったものになり, これらの規則に互換性が無い事に注意が必要です. この節では, 規則を適用する式の並びを指定する変数に与える述語の説明を最初に行

い, その後に `defrule` 関数, `let` 関数, `tellsimp` 関数と `tellafter` 関数による規則の定義と適用について解説をします. 猶, ここで例に出した Leibniz 則に関しては, Maxima の規則の定義で重要な事実が明瞭になる為, 後の小節で再度述べる事にします.

5.7.2 述語と変数の指定

Maxima では指定した式の並びに対して規則の変換を行います, 一般の式では, 指定した式の並びが埋没している為に, 規則が適用されるべき式であるかどうかを判別する必要があります. 更に, 適用すべき式かどうかを判断させる事で, 処理の効率化や間違った処理を防ぐ必要もあります.

そこで, 規則を適用すべき式の並びであるかどうかは, 式の並びと類似した式が与式に項として包含され, 更に, その部分項を構成する関数, 或いは変数等が, 式の並びで用いた関数や変数と, 何らかの意味で一致するという判断を行わなければなりません.

この判断は, 式の並びで用いた関数や変数と, 与式に含まれる部分項が対応するという関係が成立するかどうかで行えます.

そこで, Maxima では `matchdeclare` 関数を用いて, 変数とその変数に対応する述語との組み合わせとして並びを定義します.

— `matchdeclare` 関数 —

```
matchdeclare(<並びの変数1>, <述語1>, ..., <並びの変数n>, <述語n>)
matchdeclare([<並びの変数1>, ..., <並びの変数n>], <述語>)
```

`matchdeclare` 関数の引数は並びの指定で用いる並びの変数と, その並びの変数に対応する真理関数の対を引数とします. 従って, 引数の個数は常に偶数個になり, 引数が偶数個でなければエラーを返す仕様となっています.

次に, 並びの変数に対応する述語は `matchdeclare` 属性という形で対応する変数に結び付けられます. 猶, `matchdeclare` 関数で述語として指定する真理関数は独特の表記になります.

例えば, 式の並びで用いる変数 `_a` に対し, 定数であるかどうかを判断する `constantp` 関数を結び付けた場合を以下に示しましょう.

```
(%i30) matchdeclare(_a, constantp);
(%o30)                                     done
(%i31) properties(_a);
(%o31)                                     [matchdeclare]
(%i32) printprops(_a, matchdeclare);
```

```
(%o32)                                [constantp(_a)]
```

この例では並びの変数を `_a` としており、真理関数を定数であれば `true` を返す `constantp` 関数としています。変数に指定された属性を返す関数 `properties` で見ると、変数 `_a` には `matchdeclare` 属性が追加されている事が判ります。更に、`printprops` 関数で `_a` の `matchdeclare` 属性として何が付加されているかを表示させると、`constantp(_a)` が返却されていますね。この様に、`matchdeclare` 属性の実際の値として並びの変数が満すべき述語が付加されているのです。

ここで、`matchdeclare` 関数による `constantp` の処理に注目してみましょう。処理によって `matchdeclare` 属性として `constantp(_a)` が付加されていますが、これは `matchdeclare` 関数が引数として与えられた真理関数の変数部分に並び変数が追加されているのです。一般的に、並びの変数を `_a` とする時、真理関数が $P(x_1, \dots, x_n, _a)$ で与えられる場合、`matchdeclare` 関数には `matchdeclare(_a, P(x_1, \dots, x_n))` の様に入力します。例えば、並びの変数 `_b` を変数 `x, y, z` を含まない式とすると、`freeof(x, y, z, _b)` は `true` になります。そこで、`matchdeclare` 関数では、`freeof(x, y, z)` を真理関数と指定すれば良い事になります。この処理は Maxima の式の内部表現の構造を考えるとより明瞭になります。項 `freeof(x, y, z)` の内部表現は `((FREEOF SIMP) (X Y Z))` になります。`matchdeclare` 関数は、この関数の内部表現の変数リスト末尾に並びの変数 `_b` を追加し、その S 式を `matchdeclare` 属性値として並びの変数に与えているのです。

```
(%i33) matchdeclare(_b, freeof(x, y, z));
(%o33)                                done
(%i34) printprops(_b, matchdeclare);
(%o34)                                [freeof(x, y, z, _b)]
```

猶、この `matchdeclare` 関数の引数として与えられる変数は、Maxima の変数に限定されません。関数自体もここでの変数として扱う事も可能です。これも関数の内部表現を考えると明瞭になります。例えば、関数 $f(x, \dots, z)$ の内部表現は `((f SIMP) (X ... Z))` となります。規則は条件付きの置換なので、変数部分を置換えるか、関数部分を置換えるかは基本的に S 式の先頭か末尾の何れかを置換するかの違いでしかない為です。この特性の為、述語として与えられるのは、真理値関数、`lambda` 式、`block` 関数を含まない Maxima の文となります。

又、並びの変数に制限を追加する必要が無ければ、述語として、`true` や `all` を指定します。では、`matchdeclare` 関数で並び変数を予め定義し、定義した並び変数に対して `defrule` 関数で規則を定義し、`apllly1` 関数で規則を適用してみましょう。

```
(%i28) prefix("dfx");
```

```

(%o28)                                     dfx
(%i29) declare("dfx",linear);
(%o29)                                     done
(%i30) matchdeclare([_a,_b],lambda([y],not(freeof(x,y))));
(%o30)                                     done
(%i31) defrule(leibniz,dfx (_a._b),(dfx _a)._b + _a. (dfx _b));
(%o31)      leibniz : dfx (_a . _b) -> dfx _a . _b + _a . dfx _b
(%i32) apply1(dfx(sin(x).cos(x)),leibniz);
(%o32)      dfx sin(x) . cos(x) + sin(x) . dfx cos(x)
(%i33) apply1(dfx(sin(x).cos(y)),leibniz);
(%o33)      dfx (sin(x) . cos(y))
(%i34) printprops([_a,_b],matchdeclare);
(%o34) [lambda([y], not freeof(x, y), b), lambda([y], not freeof(x, y), a)]

```

この例では、変数 a と b を変数 x を含む関数とし、変数 x の関数に対して Leibniz 則を適用させるものです。その為、 $\sin(x) \cdot \cos(x)$ に対しては規則が適用されても、 $\sin(x) \cdot \cos(y)$ に対しては規則が適用されていない事に注目して下さい。又、`printprops` で変数 a, b の `matchdeclare` 属性を見ると、`matchdeclare` で指定した関数が属性値として割り当てられている事が判ります。

さて、ここでの Leibniz 則の定義では通常の積演算子 $*$ ではなく非可換積演算子を用いています。これは Maxima の規則では和 $+$ や可換積 $*$ を含む式に対しては、規則の適用が上手く動作しない為です。

そこで、和を使って演算子の線形性、可換積を使って Leibniz 則を定義してみましょう。

```

(%i1) matchdeclare([_a,_b],all);
(%o1)                                     done
(%i2) prefix("dfx");
(%o2)                                     dfx
(%i3) defrule(Leibniz2,dfx(_a*_b),dfx(_a)*_b+_a*(dfx _b));
_b _a partitions 'product'
(%o3)      Leibniz2 : dfx (_a _b) -> _a dfx _b + dfx _a _b
(%i4) defrule(Linear2,dfx(_a+_b),dfx _a + dfx _b);
_b + _a partitions 'sum'
(%o4)      Linear2 : dfx (_b + _a) -> dfx _b + dfx _a
(%i5) applyb1(dfx(a*b),Leibniz2);
(%o5)      dfx (a b) + dfx 1 a b

```



```
(%i6) applyb1(dfxx(a*b),Linear2);
(%o6)          dfx (a b) + dfx 0
```

この例に示す様に,defrule 関数で線形性や Leibniz 則を定義すると, `_d _c partitions 'product'` といった警告が出ていますね. ここで `applyb1` という関数で規則を適用させるとどうなるでしょうか? 線形性の場合には 0 が, Leibniz 則の場合には 1 が出ている事に注意して下さい. これは非常に危険な事態を招きます. 何故なら, `apply1` 関数では $1 = 1 * 1$ や $0 = 0 + 0$ となる事から無限ループに陥ってしまうからです.

これは, 並びの式を S 式で表現した場合に, ((演算子) (+ 並びの式の変数₁, ..., 並びの式の変数_n)) や ((演算子) * 並びの式の変数₁, ..., 並びの式の変数_n)) となる時に規則の適用を行うと, 並びの式の変数₁ に演算子の被演算子全てが割り当てられ, 残りの並びの変数₂, ..., 並びの変数_n に和や積の単位元が割り当てられる仕様となっている為です. この処理は和の演算子と可換積の演算子の場合に限定されます. それ以外の演算子ではこのような現象は生じません. 猶, ここでの例で `applyb1` 関数を用いたのは, `applyb1` 関数が式の木構造の葉の部分, 即ち, 底 (bottom) から式を適用する為に, 無限ループに陥らないからです. 木構造の上, 即ち, 根の部分から規則を適用する `apply1` を用いると無限ループに陥ります. 但し, 演算子が単位元に対して 0 を返す性質を与えていれば, 無限ループには陥りません. しかし, 期待した規則の適用による式の変形は出来ません.

5.7.3 並びの指定に関連する関数

この `matchdeclare` 関数に関連した関数として, 与式から指定した式の並びを検出する関数 `defmatch` 関数があります.

defmatch 関数

```
defmatch(< 関数名 >, < 式の並び >, < 変数1 >, ..., < 変数n > )
```

この `defmatch` 関数は $n + 1$ 個の引数を持ち, 特定の並びに適合するかどうか, 式を検査する関数を < 関数名 > で指定した名前で生成する関数です.

`defmatch` 関数に与える < 式の並び > は < 変数₁ >, ..., < 変数_n > を含む式になります. ここで最初の引数は式の並びを照合する関数となり, その関数で与えられた式が, n 個の引数は式の中の実際の値, 並びの中で変数として置かれるものです.

`defmatch` で構成された関数は照合に成功すると < 助変数_i > = 適合する変数のリストを返します. ここで, 照合に失敗すれば `false` を返します.

猶、 $\langle$ 式の並び $\rangle$ 中の $\langle$ 変数₁ $\rangle, \dots, \langle$ 変数_n $\rangle$ 以外の項に対して matchdeclare 関数による宣言を加えていた場合、matchdeclare 関数による述語を用いて式の分解が defmatch 関数で定義した関数で行える様になります。

次の例では、与えられた関数の線形性を調べる関数 linear の定義を行い、関数 linear を実行しています。

```
(%i2) defmatch(linear,a*x+b,x)
(%i3) linear(3*z+(y+1)*z+y^2,z);
(%o3)                                     false
(%i4) linear(a*z+b,z);
(%o4)                                     [x = z]
(%i5) nonzeroandfreeof(x,e):=if e#0 and freeof(x,e)
      then true else false
(%i6) matchdeclare(a,nonzeroandfreeof(x),b,freeof(x))
(%i7) linear(3*z+(y+1)*z+y^2,z);
(%o7)                                     false
(%i8) defmatch(linear,a*x+b,x)
(%i9) linear(3*z+(y+1)*z+y^2,z);
      2
(%o9)                                     [b = y , a = y + 4, x = z]
```

最初に defmatch で linear を定義し、次に式 $3*z+(y+1)*z+y^2$ が変数 z の一次式であるかどうか検証しています。この例では false が返却されています。これは最初の変数 a と b に関して何等の情報も無い為、単純に原子 a と原子 b を持たない一次式と判断されて false となっています。次の $a*z+b$ の場合、並び変数 x に対応するものが z であると判断した為に $[x = z]$ が返却されています。

そこで、並び変数 a と b の情報を追加します。この例では、最初に $is(e\#0 \text{ and } freeof(x,e))$ と同値な関数 nonzeroandfreeof を定義します。それから並び変数 a と b に対して、共に 0 ではなく、変数 x も含まない式であると matchdeclare 関数を用いて宣言しています。この宣言の後に defmatch 関数で関数 linear を再定義します。この再定義を行わないと、並び変数 a と b に関する情報は更新されません。これによって並び変数 a と b が変数 x から独立したもので、変数 x に関して線形であれば、対応する成分リストを返す関数 linear が定義されます。

そこで、`linear(3*z+(y+1)*z+y^2,z)` を実行すると、linear で与式と並びの式 $a*x+b$ を比較し、今度は $[b=y^2, a=y+4, x=z]$ を返します。

5.7.4 defrule 関数による規則

defrule 関数による規則の定義方法を以下に示しておきます.

defrule 関数

```
defrule(<規則名>,<並び>,<置換>)
```

defrule 関数は与えられた $\langle$ 並び $\rangle$ を指定した $\langle$ 置換 $\rangle$ で置換える規則の定義と規則の名付けを行う関数です.

この場合 $\langle$ 規則名 $\rangle$ で指定された規則が apply 関数族の関数の一つで式に適用されると、 $\langle$ 並び $\rangle$ に適合する全ての部分式が $\langle$ 置換 $\rangle$ で指定した値で置換されます. 猶、照合に失敗すると元の式を返却します.

猶,defrule 関数で設定した規則は大域変数 rules に割当てられたリストに名称が登録されます. 単純に `rules;` と入力すると、その時点で Maxima に追加されている規則名一覧のリストが表示されます. 猶,Maxima-5.9.3 では三角関数の簡易化に関連する規則が登録されていましたが、5.10.0 以降では、これらの規則は削除されている為、利用者が規則を登録していなければ、空リスト `[]` が返されます.

では、実例で解説しましょう. 先ず、最初に演算子を定義し、その演算子に対して規則を入れます. ここで定義する演算子 dfx は微分演算子の様な癖を持った前置表現の演算子とし、この演算子に対する微分としての性質を規則として defrule 関数で入れてみましょう.

```
(%i2) prefix("dfx");
(%o2)                                     dfx
(%i3) declare("dfx",linear);
(%o3)                                     done
(%i4) defrule(leibniz,dfx (a.b), (dfx a).b + a.(dfx b));
(%o4)      Leibniz : dfx (a . b) -> dfx a . b + a . dfx b
(%i5) rules;
(%o5)      [leibniz]
(%i6) dfx( a+b );
(%o6)      dfx b + dfx a
```

defrule の後で、`rules;` と入力する事で、その時点での defrule 関数による規則のリストが表示されます. この例で示す様に、rules に規則 Leibniz が登録されていますね.

さて,defrule 関数で設定した規則の内容を見たくなったら場合はどうすれば良いのでしょうか? この場合には disprule 関数 を用いると,defrule 関数で設定した規則の内容が表

示されます.

```
(%i7) disprule(chain1);
(%t7)          chain1 : dfx (a . b) -> dfx a . b + a .

dfx b

(%o7)                                     [%t4]
(%i8) disprule(all);
(%t8)          chain1 : dfx (a . b) -> dfx a . b + a . dfx b

(%o8)                                     [%t5]
(%i9) apply1(dfx (a*b),chain1);
(%o9)                                     a dfx b + dfx a b
(%i10) apply1(dfx (x*y),chain1);
(%o10)                                     dfx (x y)

(%i11) clear_rules();
(%o11)                                     false
(%i12) disprule(all);
(%o12)                                     []
```

ここでの例では、最初に disprule 関数を用いて大域変数 rules に登録された規則の表示を行っています。それから apply1 関数を用いて規則を適用させていますが、この規則を適用させる範囲を定める述語が無い為に、規則通りの書式にしか対応していません。

それから最後に、clear_rules 関数を用いて規則を全て消去しています。

defrule 関数による規則の表示

defrule 関数による規則の表示を行う disprule 関数の構文を以下に纏めておきます。

規則の表示を行う関数

```
disprule(< 規則1 > , < 規則2 > , ... )
disprule(all)
```

disprule 関数は defrule 関数, tellsimp 関数, tellsimpafter 関数で利用者が設定した規則の詳細を defmatch 関数によって定義された並びの名前込みで表示します. 特に, 引数を all とすると Maxima が持つ全ての規則を表示します.

disprule 関数による規則の表示では, $\rightarrow$ の左側に規則を適用される式が表示され, 右側に適用した結果が表示されます. この書式は defrule で定義した規則の内部書式に関連します.

先程の, defrule 関数の動作を確認してみましょう. 猶, defrule 関数の実体は porc-\$defrule 関数です. そこで, :lisp 演算子を用いてこの関数の動きを調べてみましょう.

```
(%i8) :lisp (trace proc-$defrule)
WARNING: TRACE: redefining function PROC-$DEFRULE in top-level, was defined in
      /usr/local/maxima-5.13.0/src/binary-clisp/matcom.fas
;; Tracing function PROC-$DEFRULE.
(PROC-$DEFRULE)
(%i8) defrule(chain1,dfx (a*b),(dfx a)*b+a*dfx b);
1. Trace:
(PROC-$DEFRULE
  '($CHAIN1 (($DFX) ((MTIMES) $A $B))
    ((MPLUS) ((MTIMES) (($DFX) $A) $B) ((MTIMES) $A (($DFX) $B))))))
1. Trace: PROC-$DEFRULE ==>
((MSETQ) $CHAIN1
  ((MARROW) (($DFX SIMP) ((MTIMES SIMP) $A $B))
    ((MPLUS SIMP) ((MTIMES SIMP) (($DFX SIMP) $A) $B)
      ((MTIMES SIMP) $A (($DFX SIMP) $B))))))
(%o8)          chain1 : dfx (a b) -> a dfx b + dfx a b
```

ここで示す様に, defrule 関数で定義される規則の内部表現は ((MARROW) 適用前の式 適用後の式) の書式になっています. disprule 関数は, この S 式のヘッダの (MARROW) を単純に $\rightarrow$ で置き換えて表示しています.

defrule 関数による規則の適用は apply1, apply2, applyb1 関数を用います.

$$\begin{aligned} & \text{apply1}(\langle \text{式} \rangle, \langle \text{規則}_1 \rangle, \dots, \langle \text{規則}_n \rangle) \\ & \text{apply2}(\langle \text{式} \rangle, \langle \text{規則}_1 \rangle, \dots, \langle \text{規則}_n \rangle) \\ & \text{applyb1}(\langle \text{式} \rangle, \langle \text{規則}_1 \rangle, \dots, \langle \text{規則}_n \rangle) \end{aligned}$$

最初の `apply1` 関数は与えられた〈式〉に対して〈規則₁〉を最初に作用させます。この時、〈式〉の木構造の根側から大域変数 `maxapplydepth` で指定される深さ迄の全ての部分式に〈規則₁〉を適用させます。これによって得られた〈式₂〉に対し、次の〈規則₂〉を同様に適用します。以降、帰納的に各部分式に作用させ、〈規則_n〉を全ての部分式に作用させて終了します。

applyb1 関数は apply1 関数と似ていますが、apply1 関数が〈式〉の木構造に対し、上から下へと作用して行くのに対して applyb1 関数は、〈式〉の最下層の部分式から作用させ、規則の適合に失敗すると、もう一つ上の階層の部分式に帰納的に作用させます。

これらの適用関数は与式の木構造, 即ち, 内部構造に密接に関連します. その為, 二項演算子の場合は特に, nary 型か infix 型であるかで結果が異なります.

```
(%i2) matchdeclare([_a,_b],all);  
(%o2) done  
(%i3) prefix("dfx");
```

```

(%o3)                                dfx
(%i4) nary("><");
(%o4)                                ><
(%i5) defrule(Leibniz,dfx (_a><_b),(dfx _a)><_b+_a><(dfx _b));
(%o5)      Leibniz : dfx (_a >< _b) -> dfx _a >< _b + _a >< dfx _b
(%i6) applyb1(dfx (a><b><c),Leibniz);
(%o6)                                dfx (a >< b >< c)
(%i7) apply1(dfx (a><b><c),Leibniz);
(%o7)                                dfx (a >< b >< c)
(%i8) apply2(dfx (a><b><c),Leibniz);
(%o8)                                dfx (a >< b >< c)

(%i9) infix("<>");
(%o9)                                <>
(%i10) defrule(Leibniz2,dfx (_a<>_b),(dfx _a)<>_b+_a<>(dfx _b));
(%o10)      Leibniz2 : dfx (_a <> _b) -> dfx _a <> _b + _a <> dfx _b
(%i11) applyb1(dfx (a<>b<>c),Leibniz2);
(%o11)                                dfx (a <> b) <> c + a <> b <> dfx c
(%i12) apply1(applyb1(dfx (a<>b<>c),Leibniz2),Leibniz2);
(%o12)                                (dfx a <> b + a <> dfx b) <> c + a <> b <> dfx c
(%i13) apply1(dfx (a<>b<>c),Leibniz2);
(%o13)                                (dfx a <> b + a <> dfx b) <> c + a <> b <> dfx c
(%i14) apply2(dfx (a<>b<>c),Leibniz2);
(%o14)                                (dfx a <> b + a <> dfx b) <> c + a <> b <> dfx c

```

ここでは二つの演算子 $><$ と $<>$ を定義し、Leibniz 則を与えています。ところが、nary 型の演算子 $><$ を用いた $a><c$ に対しては、規則の適用が上手く出来ないのに対し、infix 型の演算子 $<>$ では上手く処理が来ています。これは、内部表現の違いによるものです。nary 型の場合、項の内部表現は ((nary 型演算子) 被演算子₁, ..., 被演算子_n) の様に nary 演算子が先頭に一つ置かれ、その後に被演算子が並ぶ書式となります。これに対し、infix 型の演算子では ((infix 型演算子)(... ((infix 型演算子) 被演算子₁, 被演算子₂) ...), 被演算子_n) となります。その為、 $a><c$ が内部で (($><$ SIMP) A B C) となるのに対し、 $a<>b<>c$ は (($<>$ SIMP)(($<>$ SIMP) A B) C) となります。ここで規則は $_a><_b$ や $_a<>_b$ に対して適用させる為、 $_a$ と $_b$ に対応する被演算子を捜します。その為、nary 型の場合は被演算子が平リストとして配置される為に、適応するものが

無いと判断し, infix 型の場合は, 被演算子が二つ一組になる為に, 照合に成功するので
す. nary 型の演算子に規則を適用させる場合には, 括弧 () を上手く利用して, 規則の
適用を行う函数にとって分かり易い書式にしておく必要があります.

```
(%i15) apply1(df x((a><b)><c),Leibniz);
(%o15)      (df x a >< b + a >< df x b) >< c + (a >< b) >< df x c
(%i16) apply1(df x((a><b><c)><d),Leibniz);
(%o16)      df x (a >< b >< c) >< d + (a >< b >< c) >< df x d
```

この例では, $(a><b)><c$ とした為に, 最初の適合で, $_a \rightarrow a><b$, $_b \rightarrow c$ の対応が付き, 更に, $a><b$ に対しても, $_a$ と $_b$ との対応が付く為, apply1 函数によって規則の適用に成功しているのです. ところが, 最後の例では $(a><c)><d$ の為, 最初の照合には成功するものの, $a><c$ の照合に失敗するのです.

5.7.5 defrule 函数で定義した規則の適用に関連する大域変数

規則に関連する大域変数

変数名	初期値	概要
maxapplydepth	10000	apply1 と apply2 が停止する階層
maxapplyheight	10000	applyb1 が停止する階層

大域変数 maxapplyheight は apply1 函数, apply2 函数や applyb1 函数が停止する与式の階層構造の最高位となります. ここで, maxima の式には LISP の S 式の様な階層構造があります. apply1 等の函数は maxapplyheight よりも低い個所, 即ち, 木構造の根本側に作用し, それよりも高ければ作用しません (木構造の根元側を最低辺としてみま
す). 但し, デフォルト値の 10000 は式の殆ど全てと言える程の高さになるでしょう.

5.7.6 tellsimp 函数と tellsimpafter 函数による規則の定義

Maxima では入力された式を Maxima の項順序 $>_m$ に従って並び換えたり, 和や差を自動的に簡易化します. これは内部函数 simplifya を用いる為ですが, tellsimp 函数と tellsimpafter 函数は指定した規則を simplifya を用いて簡易化させる函数です.

— tellsimp と tellsimpafter —

```
tellsimp(<並び>, <置換>)
tellsimpafter(<並び>, <置換>)
```

tellsimp 関数と tellsimpafter 関数は、規則を定義する関数ですが、後述の defrule 関数や let 関数の様に、定義した規則を適用する関数を用いて式の変形を行う必要が無い点で異なります。

又、tellsimp 関数と tellsimpafter 関数で定義した規則は自動的に大域変数 rules に追加されます。この時、規則名は自動的に設定されますが、この時の規則名は並び変数の内部表現の最初の成分、即ち、演算子名や関数名に、文字列 rule と、この演算子/関数に対応する規則番号を並べた文字列で表現されます。ここで規則番号は、その演算子/関数に tellsimp 関数や tellsimpafter 関数で設定された規則の総数に 1 を加えた正整数値で与えられるものです。

```
(%i1) matchdeclare([_a,_b],all);
(%o1)                                     done
(%i2) prefix("dfx");
(%o2)                                     dfx
(%i3) nary("><");
(%o3)                                     ><
(%i4) rules;
(%o4)                                     []
(%i5) tellsimp(dfx(_a><_b),(dfx _a)><_b+_a><(dfx _b));
(%o5)                                     [dfxrule1, false]
(%i6) rules;
(%o6)                                     [dfxrule1]
(%i7) dfx(a><b><c);
(%o7)                                     dfx (a >< b >< c)
(%i8) dfx(a><b);
(%o8)                                     dfx a >< b + a >< dfx b
(%i9) dfx((a><b)><c);
(%o9)                                     (dfx a >< b + a >< dfx b) >< c + (a >< b) >< dfx c
(%i10) nary("<>");
(%o10)                                     <>
(%i11) tellsimpafter(dfx(_a<>_b),(dfx _a)<>_b+_a<>(dfx _b));
```

```
(%o11) [dfxrule2, dfxrule1, false]
(%i12) dfx((a<>b)<>c);
(%o12) dfx (a <> b) <> c + (a <> b) <> dfx c
```

tellsimp と tellsimpafter 関数は共に第一引数に式の並び, 第二引数に置換式を指定します. この例では, 演算子 dfx に対して規則を二つ定めていますが, 最初の tellsimp による規則には dfxrule1, 次の tellsimpafter による規則では, dfxrule2 と演算子名, 文字列 rule, 既存の規則の総数に 1 を加えた数値で規則名が構成されている事が判ります. 猶, 式の並びでは演算子に注意が必要になります. 基本的に和+や可換積*のみで構成された式の並びに対しては, 式の並びの照合に失敗する可能性が高く, 最悪の場合, 無限ループに陥る可能性もあります. 特に, tellsimp 関数の場合, 再帰的に規則の適用を行う為, 式の入力と同時に無限ループに陥る事になり, 非常に危険です.

5.7.7 let 関数による規則

次に, let 関数による規則の定義を解説しましょう. 先ず, let 関数の構文を以下に示しておきましょう.

— let 関数 —

```
let([< 式の並び >, < 式 >, < 述語 >, < 変数1 >, ..., < 変数n >], < パッケージ名 > )
let(< 式の並び >, < 式 >, < 述語 >, < 変数 >, ..., < 変数n > )
let(< 式の並び >, < 式 > )
let([< 式の並び >, < 式 >], < パッケージ名 > )
```

この let 関数は < 述語 > が true の場合, < 式の並び > を < 式 > で置換する規則を定義する関数です. 猶, defrule 関数, tellsimp や tellsimpafter 関数による規則と違う点は, 規則の評価関数 letsimp 関数が内部で CRE 表現を用いる為に, 特に変数変換を行う事で, 有理式として表現可能な, CRE 表現に向けた式が適しています.

let 関数は式の並びと変換式で構成されたリストをパッケージと呼ばれる大域変数に属性として割当てられたリストに追加する作業を行います. 以下に具体的な例を示しておきましょう.

```
(%i5) let(pochi2(_a*_b),pochi2(_a)+3*_a*pochi2(_b));
(%o5) pochi2(_a _b) --> 3 _a pochi2(_b) + pochi2(_a)
(%i6) :lisp (mget $current_let_rule_package 'letrules)
((MTEXT) (($POCHI2 SIMP) (MTIMES SIMP) $_A $_B)) -->
```

```

((MPLUS) (($POCHI2) $ _A) ((MTIMES) 3 $ _A (($POCHI2) $ _B))))
((MTEXT) (($POCHI SIMP) ((MTIMES SIMP) $ _A $ _B)) -->
((MPLUS) (($POCHI) $ _A) ((MTIMES) $ _A (($POCHI) $ _B))))
(%i6) mike(_a,_b):=true;
(%o6)
mike(_a, _b) := true
(%i7) let(pochi3(_a*_b),pochi3(_a)+3*_a*pochi3(_b),mike,_a,_b);
(%o7) pochi3(_a _b) --> 3 _a pochi3(_b) + pochi3(_a) where mike(_a, _b)
(%i8) :lisp (mget $current_let_rule_package 'letrules)
((MTEXT) (($POCHI3 SIMP) ((MTIMES SIMP) $ _A $ _B)) -->
((MPLUS) (($POCHI3) $ _A) ((MTIMES) 3 $ _A (($POCHI3) $ _B))) WHERE
(($MIKE) $ _A $ _B))
(MTEXT) (($POCHI2 SIMP) ((MTIMES SIMP) $ _A $ _B)) -->
((MPLUS) (($POCHI2) $ _A) ((MTIMES) 3 $ _A (($POCHI2) $ _B))))
(MTEXT) (($POCHI SIMP) ((MTIMES SIMP) $ _A $ _B)) -->
((MPLUS) (($POCHI) $ _A) ((MTIMES) $ _A (($POCHI) $ _B))))
(%i8)

```

この例では、最初に述語を指定せずに式の並びと置換式だけを let 関数で指定し、次には述語と変数を加えた指定を行っています。この例で判る様に、規則は大域変数 `current_let_rule_package` に束縛されたリストに追加されている事が判ります。

〈式の並び〉には、Maxima の原子、 $\sin(x)$ や $f(x,y)$ の様な関数の可換積 $*$ 、商 $/$ や冪 $^$ を含む項となります。但し、負の冪を用いる場合には大域変数 `letrat` を `true` にする必要があります。

〈式₁〉に含まれる〈変数_i〉と対応する〈述語〉を省く場合には、それらの変数が `matchdeclare` 関数によって予め `true` であると宣言されていなければなりません。

let 関数の引数の末尾に〈パッケージ名〉を追加すれば、定義した置換規則を指定したパッケージに追加します。未設定の場合、自動的に `current_let_rule_package` に割当てられたパッケージに追加されます。

これらの置換関数は一度に幾つかの規則の組合せを用いて作用させられます。各々の規則の組合せは任意数の let 関数で操作された任意の数の規則を含む事が可能で、利用者が与えた名前を参照されます。

さて、let 関数で定義した規則を式に適用する場合、`defrule` 関数による定義とは異なり、`letsimp` 関数を用います。この `letsimp` 関数の構文を以下に示しておきましょう。

5.7.8 letsimp 関数による規則の適用

· letsimp 函数

$$\begin{aligned} & \text{letsimp}(\langle \text{式} \rangle, \langle \text{規則パッケージ名}_1 \rangle, \dots, \langle \text{規則パッケージ名}_n \rangle) \\ & \text{letsimp}(\langle \text{式} \rangle, \langle \text{規則パッケージ名} \rangle) \\ & \text{letsimp}(\langle \text{式} \rangle) \end{aligned}$$

ここで `letsimp` 関数は〈式〉が指定した規則パッケージに含まれる規則の適用を続けて、式の変化が無くなるまで規則の適用を続けます。

猶, 規則パッケージの指定が無い場合, `current_net_rule_package` に割当てられた規則パッケージが利用されます.

パッケージを複数指定した場合、(式)には左端のパッケージから順番に適用されます。例えば、`letsimp(expr, package1, package2)`を実行すると、最初に`letsimp(expr, package1)`を実行し、次に`letsimp(%, package2)`を実行したものと同じ結果が得られます。

ここで規則パッケージを指定して `current_let_rule_package` が切替えられる事はありません。

又, `let` 関数で構築した規則の表示は `letrules` 関数を用いて表示します.

- let 関数による規則の表示を行う関数

```
letrules(〈規則パッケージ〉)
letrules()
```

letrules 関数は、引数で指定した〈規則パッケージ〉に含まれる規則の詳細を表示します。ここで、引数を指定しない letrules() の場合、大域変数 current_let_rule_package に割当てられた規則パッケージに含まれる規則を表示します。猶、この大域変数の初期値は default_let_rule_package です。

それでは、簡単に `let` 関数と `letsimp` 関数の動作を確認しておきましょう。

```
(%i1) matchdeclare([_a,_b],true);  
(%o1) done  
(%i2) let([tama(_a)^2,tama(2*_a)+1],nekoneko);  
2  
(%o2) tama (_a) --> tama(2 _a) + 1  
(%i3) letrules();  
(%o3) done  
(%i4) letrules(nekoneko);
```

```

                2
      tama (_a) --> tama(2 _a) + 1

(%o4)
done
(%i5) letrules(all);
(%o5)
done
(%i6) letsimp(tama(x)^2);

                2
(%o6)      tama (x)
(%i7) letsimp(tama(x)^2,nekoneko);
(%o7)      tama(2 x) + 1
(%i8) current_let_rule_package:nekoneko;
(%o8)      nekoneko
(%i9) letsimp(tama(x)^2);
(%o9)      tama(2 x) + 1

```

この例では, $\sin$ 関数の倍角公式を模擬したものです. 先ず, `let` 関数で `nekoneko` という名前の規則パッケージに規則を追加し, `letrules` 関数で規則の表示を行っています. ここで最初の `letrules()` で `current_let_rule_package` に指定された `default_let_rule_package` の内容を表示します. 次の `letrules` では引数に `nekoneko` が指定されている為, `nekoneko` に含まれる規則が表示される事になります. 次に, `letsimp` 関数ではパッケージを指定しなければ, `current_let_rule_package` にある規則が適用されますが, ここには規則が無い為に, 入力と同じ式を返す事になります. ここで, `letsimp` にパッケージを指定した場合と, `current_let_rule_package` にパッケージの `nekoneko` を割り当てた場合には, `letsimp` によって引数の式に規則が適用されています.

この `letsimp` 関数は内部的に CRE 表現に式を変換して処理します. 従って, 本質的に多項式の処理となる式の計算では効力を発揮するでしょう.

5.7.9 letsimp 関数 に関連する大域変数

letsimp に関連する大域変数		
変数名	初期値	概要
default_let_rule_package	'default_let_rule_package	デフォルトで用いられる規則パッケージ
current_let_rule_package	'default_let_rule_package	規則パッケージを指定しない場合に用いられる規則パッケージ
let_rule_packages	[default_let_rule_packages	規則パッケージ名の一覧
letrat	false	有理式の簡易化に関連
letvarsimp	false	はて？

default_let_rule_package はパッケージ名を指定しなかった場合に let 関数で定義された規則が格納されるパッケージになります。

current_let_rule_package に let 関数で構築したパッケージ名を割り当てると, let 関数や letsimp 関数で用いられるパッケージは自動的に current_let_rule_package に割り当てられたパッケージに切り替えられます。

let_rule_packages は let 関数で構築した規則のパッケージのリストを返します。初期状態では default_let_rule_package のみが存在する為, let_rule_packages の初期値は [default_let_rule_package] になります。

大域変数 letrat の値が false の場合 letsimp 関数が式の分子と分母を各々別に簡易化して結果を返します。但し, $n!/n$ を $(n-1)!$ にする様な置換は出来ません。この様な置換を行う為には, 大域変数 letrat を true に設定しておかなければなりません。すると, 分子, 分母の商は要求の通りに簡易化されます。

5.7.10 規則の削除

規則の削除も規則を定義した関数の系統によって異なりますが, let 関数とそれ以外の関数に二分されます。先ず, defrule 関数, tellsimp 関数と tellsimpafter 関数で定義した規則は, clear_rules で削除する事が可能です。

規則の削除を行う関数

```
clear_rules()
```

`clear_rules` 関数は引数を取らない関数です. 大域変数 `rules` に含まれている規則を削除する関数です. `clear_rules` 関数を実行すると, 可換積 $*$, 可換冪 $^$ と和 $+$ の規則番号を初期状態の 1 に戻します.

```
(%i4) defrule(Leibniz,dfx(_a<>_b),(dfx _a)<>_b+_a<>dfx _b);
(%o4)      Leibniz : dfx (_a <> _b) -> dfx _a <> _b + _a <> dfx _b
(%i5) rules;
(%o5)      [Leibniz]
(%i6) tellsimp(dfx(_a<>_b),(dfx _a)<>_b+_a<>dfx _b);
(%o6)      [dfxrule1, false]
(%i7) rules;
(%o7)      [Leibniz, dfxrule1]
(%i8) dfx (a<>b);
(%o8)      dfx a <> b + a <> dfx b
(%i9) clear_rules();
(%o9)      false
(%i10) rules;
(%o10)     []
```

この例では `clear_rules` によって, `defrule` 関数による規則と, `tellsimp` による規則が消去されている事が判ります.

猶, `tellsimp` 関数と `tellsimpafter` 関数に対しては, `remrule` 関数を用いて直接, 規則を指定して削除する事が可能です.

tellsimp 関数と tellsimpafter 関数による規則の削除を行う関数

```
remrule(<対象>,<規則名>)
remrule (<対象>,all)
```

`remrule` 関数は `<規則名>` で指定した規則を `<対象>` から削除します. 第一引数の `<対象>` は, 演算子, 又は関数になります. この理由は, `tellsimp` 関数や `tellsimpafter` 関数による規則は, 演算子や関数に対して付加される規則だからです. その為, 規則名を指定しない場合, 指定した対象に附属する規則をその命名規則に従って全て削除し, 引数が `all` の場合は大域変数 `rules` に登録されている全ての `tellsimp` 関数と `tellsimpafter` 関

数で設定した規則を削除します.

最後に let 関数で設定した規則の削除は,remlet 関数を用います.

— let 関数による規則の削除を行う関数 —

```
remlet(< 項 > )
remlet(< 項 > , < パッケージ名 > )
remlet(all)
remlet()
```

これらの関数は全て let 関数で定義された置換規則 $\langle \text{項} \rangle \rightarrow \langle \text{式} \rangle$ を削除します.

$\langle \text{パッケージ名} \rangle$ が与えられると指定された規則パッケージから規則を削除します.

それに対し,引数を指定しない `remlet()` と `clear_rules()`, 引数が all の `remlet(all)` は規則パッケージから代入規則の全てを削除します.

ここで, 規則パッケージ名が, 例えば, `remlet(all, < パッケージ名 >)` で与えられていれば, 指定された規則パッケージも削除されます. もし, 構築した規則が古い規則を上書きしたものであれば,remlet で新しい規則を削除すれば, 古い規則が復活します.

大域変数 `let_rule_package` の値は全ての利用者定義の規則パッケージに特殊なパッケージを加えたもののリストとなります. ここで,`default_let_rule_package` は利用者が特に規則パッケージを指定しない場合に用いられる規則パッケージの名前です.

5.8 式の評価

5.8.1 式の自動簡易化

入力した式は大域変数 `simp` が `true` の場合に自動的に解釈されます。例えば, `texttt1+2` や `cos(%pi/2)` は, 夫々, 3 や 0 に変換されます。これは関数や演算子に付加された属性を基に, その属性に対応する内部関数が解釈を実行する為です。

Maxima に組み込みの関数の場合, 内部関数 `defprop` を用いて, Maxima の関数や演算子の `operator` 属性に対して簡易化を行う内部関数が付加されています。

例えば, `cos` 関数の場合, 内部関数の `simp-%cos` 関数が対応します。基本的に, 関数の自動簡易化を実行する内部関数の名前は, `simp-%` が頭に付くか, `-simp` が後に付いた名前になります。

この属性として設定された簡易化を行う内部関数では, Maxima の大域変数によってその式の簡易化制御が行われる事もあります。

例えば, `cos` 関数の自動簡易化を行う内部関数 `simp-%cos` の内容を以下に示しておきましょう。

```
(defmfun simp-%cos (form y z)
  (oneargcheck form)
  (setq y (simpcheck (cadr form) z))
  (cond ((double-float-eval (mop form) y))
        ((and (not (member 'simp (car form))) (big-float-eval (mop form) y)))
        ((taylorize (mop form) (second form)))
        ((and $piargs (cond ((zerop1 y) 1) ((linearp y '%pi) (%piargs-sin/cos (add %pi/2 y))))))
        ((and $iargs (multiplep y '%i)) (cons-exp '%cosh (coeff y '%i 1)))
        ((and $triginverses (not (atom y))
          (cond ((eq '%acos (setq z (caar y))) (cadr y))
                ((eq '%asin z) (sqrt1-x^2 (cadr y)))
                ((eq '%atan z) (div 1 (sqrt1+x^2 (cadr y))))
                ((eq '%acot z) (div (cadr y) (sqrt1+x^2 (cadr y))))
                ((eq '%asec z) (div 1 (cadr y)))
                ((eq '%acsc z) (div (sqrtx^2-1 (cadr y)) (cadr y)))
                ((eq '%atan2 z) (div (caddr y) (sq-sumsq (cadr y) (caddr y))))))
        ((and $trigexpand (trigexpand '%cos y)))
        ($exponentialize (exponentialize '%cos y))
        ((and $halfangles (halfangle '%cos y)))
        ((apply-reflection-simp (mop form) y $trigsign))
        ;((and $trigsign (mminusp* y)) (cons-exp '%cos (neg y)))
        (t (eqtest (list '(%cos) y) form))))
```

ここで Maxima の関数や大域変数には先頭に `$` が付くので, どのような関数や定数が利用されているか判るかと思いますが, ここで `%piargs`, `trigexpand` と `halfangles` が

Maxima の大域変数です.

従って, これらの大域変数が `cos` 関数の自動簡易化に大きな影響を持つ事が判るでしょう.

猶, 関数の自動簡易化に影響を与える大域変数はソースファイルを調べなくても, `options` 関数で調べる事が可能です.

この `cos` 関数を `options` 関数で調べた結果を以下に示しておきましょう.

```
(%i7) options(cos);
(%o7) [float, numer, bfloat, %piargs, %iargs, triginverses, trigexpand,
      exponentialize, halfangles, trigsign, logarc]
```

この様に, `options` 関数で返されたリストに含まれる大域変数が, この `cos` 関数の評価で影響を及ぼす大域変数となるのです.

これに対し, 演算子や利用者が Maxima の関数を用いて定義した関数に対しては, `declare` 関数によって宣言された属性に対応する内部関数を用いて式の評価が自動的に実行されます (5.4.5 参照).

5.8.2 ev 関数

次に Maxima で最も重要な関数の一つである `ev` 関数を解説します. この `ev` 関数は非常に強力であり, 柔軟性のある Maxima の式を評価する関数です.

この `ev` 関数の構文を以下に示しておきます.

ev 関数

```
ev(<式>, <引数1>, ..., <引数n>)
```

`ev` 関数は与式 `<式>` の評価を実行しますが, その際に, 評価を行う為の局所的な環境を設定し, その環境で式の評価を実行します. この環境は `<引数1>, ..., <引数n>` で設定されるもので, Maxima の様々な大域変数を `true` にしたり, 式の変数に値を割当てる事, 更には, 式を評価する関数 (`evfun`) を指定する事で設定されるものです.

猶, Maxima の最上層のみで `ev()` を外した表記も可能です.

```
(%i1) ev((x+1)^4, expand);
              4      3      2
(%o1)      x  + 4 x  + 6 x  + 4 x + 1
(%i2) (x+1)^4, expand;
```

```

(%o2)

$$x^4 + 4x^3 + 6x^2 + 4x + 1$$

(%i3) x.y.z;
(%o3)

$$x \cdot y \cdot z$$

(%i4) (x.y).z,dotassoc:false;
(%o4)

$$(x \cdot y) \cdot z$$

(%i5) (x.y).z;
(%o5)

$$x \cdot y \cdot z$$

(%i6) x^2+2*x+1,factor;
(%o6)

$$(x + 1)^2$$

(%i7) x^2/(y+1)+2*x/(y^2-1)+1,ratsimp;
(%o7)

$$\frac{y^2 + x^2 y - x^2 + 2x - 1}{y^2 - 1}$$


```

この例では、最初に $(x+1)^4$ の展開を行います。最初の書き方が `ev` 関数の正規の記述方法ですが、Maxima の最上層に限定して $(x+1)^4$, `expand` の様に `ev()` を省略出来ます。この表記が使えないのは `block` 文内部や `lambda` 関数内部で、Maxima の最上層、即ち、通常の入力プロンプトが出ている場合や Maxima のバッチファイル内部にはこの表記が利用出来ます。

次に、大域変数 `dotassoc` の値を変更して式の評価を実行しています。ここで、非可換積の結合律を制御する大域変数 `dotassoc` がデフォルトで `true` となっている為、 $(x \cdot y) \cdot z$ は $x \cdot y \cdot z$ に自動的に評価されます。ここで、`ev` 関数で評価を行う際に、一時的に大域変数 `dotassoc` を `false` にしています。`ev` 関数の評価と大域変数の値が無関係な事に注目して下さい。

最後に `evfun` 属性を持つ関数である `factor` 関数と `ratsimp` 関数を使って評価しています。ここで `evfun` 属性を持たない関数を用いた場合、式が評価されずにそのままの式が返されます。

更に、`ev` 関数では式に含まれる変数に値を一時的に割当てて式を評価する事も可能です。この評価は方程式の解を解いた後に、その結果を使って他の式の評価する場合や解の検証で便利です。

```

(%i29) solve([x^2-y^2+x*y-1,x+y-3],[x,y]);

```

```

(%o29) [[x = -  $\frac{\sqrt{41} - 9}{2}$ , y =  $\frac{\sqrt{41} - 3}{2}$ ],
          [x =  $\frac{\sqrt{41} + 9}{2}$ , y = -  $\frac{\sqrt{41} + 3}{2}$ ]]

(%i30) x*y,%[1];

(%o30)  $-\frac{(\sqrt{41} - 9)(\sqrt{41} - 3)}{4}$ 

```

この例では, 最初に連立方程式 $x^2 - y^2 + xy - 1 = 0$, $x + y - 3 = 0$ を解き, その結果を用いて式 xy を評価しています. これは `ev` 関数の性質で, 引数に論理演算子 `=` を含む式で, 左辺が通常の変数であれば, 左辺の変数に右辺の式が割当てられます.

ここで, 具体的に `ev` 関数で利用可能な引数を示しましょう.

ev 関数で利用可能な引数

evflag 属性を持つ原子	evflag 属性を持つ原子に true を一時的に設定.
evfun 属性を持つ関数	evfun 属性を持つ関数を, 引数の左側から順番に式に適用.
expand	大域変数 expop に大域変数 maxposex の値, 大域変数 expon に大域変数 maxnegex の値を各々割当てて式を展開.
expand($\langle$ 整数 ₁ $\rangle$, $\langle$ 整数 ₂ $\rangle$)	大域変数 maxposex に $\langle$ 整数 ₁ $\rangle$, maxnegex に $\langle$ 整数 ₂ $\rangle$ を各々設定し式を展開.
eval	式を評価.
noeval	式を無評価.
nouns	式を名詞型で評価.
numer	大域変数 numer と大域変数 float を true にして式を評価
risch	risch 積分を実行
diff	式中の名詞型の微分を評価.
derivlist($\langle x_1 \rangle, \dots, \langle x_n \rangle$)	名詞型の微分で, $\langle x_1 \rangle, \dots, \langle x_n \rangle$ による微分のみを評価.
変数への式の割当	大域変数や式中の変数に割当と式の評価を実行.
local($\langle x_1 \rangle, \dots, \langle x_n \rangle$)	ev 内部で用いる局所変数 $\langle x_1 \rangle, \dots, \langle x_n \rangle$ を宣言.
detout	逆行列の計算で行列式を行列の外に出す.

ev 関数では, evflag 属性を持つ引数を true に設定したり, 大域変数に値を設定する事で式の評価の為に局所的な環境設定を行い, それから, 与式に代入を実行し, evfun 属性を持つ関数や, 大域変数の値による Maxima の自動変換によって式を評価します. 先ず, evflag 属性を持つ大域変数はデフォルトで以下のものがあります.

— evflag 属性をデフォルトで持つ大域変数 —

exponentialize	%emode	demoivre	logexpand
logarc	lognumber	radexpand	keepfloat
listarith	float	ratsimpexpons	ratmx
simp	simpsum	simpproduct	algebraic
ratalgdenom	factorflag	ratfac	infeval
%enumer	programmode	lognegint	logabs
letrat	halfangles	exptisolate	isolate_wrt_times
sumexpand	cauchysum	numer_pbranch	mipbranch
dotscrules	trigexpand		

大域変数に evflag 属性を持たせる為には, declare 関数を用います. 更に, ある大域変数が evflag 属性を持つかどうかは, properties 関数を使って調べられます.

evflag 属性を持つ大域変数を定義し, ev 関数によって評価する例を以下に示します.

```
(%i1) declare(tama, evflag)$
(%i2) tama:false$
(%i3) ev('(if tama=true then print("nekoneko") else print("1234")));
1234
(%o3)                                     1234
(%i4) ev('(if tama=true then print("nekoneko") else print("1234")),tama);
nekoneko
(%o4)                                     nekoneko
(%i5) properties(tama);
(%o5)                                     [value, evflag]
(%i6) :lisp (get '$tama 'evflag)
T
```

この例では, 変数 tama に evflag 属性を declare 関数を用いて持たせ, それから, if 文で構成された式の評価を行っています. 最初の例では tama には false を設定している為, 評価式では tama が false の場合の処理が実行されています. 次に, ev 関数の引数として tama を与えると, tama に evflag 属性を持たせている為, 自動的に値に true が設定され, その値を用いて式が評価され, 結果として tama が true の場合の処理が実行されている事が判ります. 次に, properties 関数を用いて属性を調べています. LISP で調べる場合には, 内部表現に対して get 関数で evflag 属性を取出します. evflag 属性があれば T が返却され, そうでない場合には NIL が返却される事で判別出来ます.

evflag 属性に似た属性に evfun 属性があります。これは函数に対する属性で、ev 函数の引数として利用可能な函数である事を意味します。デフォルトで evfun 属性を持つ函数を以下に示しておきます。

evfun 属性を持つ函数			
radcan	factor	ratsimp	trigexpand
trigreduce	logcontract	rootscontract	bfloat
ratexpand	fullratsimp	rectform	polarform

evflag 属性と同様に、declare 函数を用いて函数に evfun 属性を持たせる事が出来ます。ここでは簡単な函数を定義して、実際にその作用を観察してみましょう。

```
(%i1) mike(z):=diff(z,x,2)$
(%i2) properties(mike);
(%o2)                                     [function]
(%i3) x^2,mike;
                                     2
(%o3)                                     x
(%i4) declare(mike,evfun)$
(%i5) properties(mike);
(%o5)                                     [evfun, function, noun]
(%i6) x^2,mike;
                                     2
(%o6)                                     2
(%i7) :lisp (get '$mike 'evfun)
T
```

この例では、変数 x で二階微分を行う函数 `mike` を定義しています。この函数は evfun 属性を最初は持っていない為に ev 函数による評価が出来ません。ところが、declare 函数で evfun 属性を付加すれば、ev 函数による評価では `mike` が実行され、結果として二階微分が得られます。函数が evfun 属性を持つかどうかの判定は、properties 函数や、LISP の get 函数を用いて行えます。この点も evflag 属性と同様です。

evfun 属性を持つ函数が複数 ev 函数に与えられた場合、この evfun 函数の作用の順番は ev 函数の左端の evfun 函数から順番に式に作用させます。

```
(%o29) tst(z) := expand(z )
(%i30) declare(tst,evfun)$
(%i31) (x+1)^2,tst,factor;
```

```

(%o31)

$$(x + 1)^4$$

(%i32) (x+1)^2,factor,tst;
(%o32)

$$x^4 + 4x^3 + 6x^2 + 4x + 1$$

(%i33) tst(factor(x+1)^2);
(%o33)

$$x^4 + 4x^3 + 6x^2 + 4x + 1$$

(%i34) factor(tst((x+1)^2));
(%o34)

$$(x + 1)^4$$


```

この例で示す様に, ev 関数の引数として, evfun 属性を持つ factor と利用者定義関数の tst を与えています. 先ずはじめに ev 関数の引数として左から tst, factor の順に与えた為, factor(tst(与式)) を処理しています. 次に, factor, tst の順で ev 関数の引数として引き渡した場合, 式は展開されています. 即ち, tst(factor(与式)) で処理したからです.

expand を指定すると, 大域変数 maxposex と maxnegex に設定された値を大域変数 expop と expon に各々割当てて式の展開を行います. 猶, 大域変数 expop と expon は $(x+1)^3$ の様な冪乗で自動展開する次数を指定する大域変数です.

大域変数 maxposex と maxnegex は expand 関数で展開する冪乗の最大と最小次数を設定する大域変数です. デフォルトで大域変数 maxposex と maxnegex の値が 1000 の為, 冪の最大次数が 1000 以下であれば式の展開を自動実行します. expand(⟨整数₁⟩, ⟨整数₂⟩) の場合は大域変数 maxposex に ⟨整数₁⟩, maxnegex に ⟨整数₂⟩ を各々設定して与式を展開します.

```

(%i1) (x+2)^1001,expand;
(%o1)

$$(x + 2)^{1001}$$

(%i2) (x+2)^2/(x+1)^3,expand(2,3);
(%o2)

$$\frac{x^2}{x^3 + 3x^2 + 3x + 1} + \frac{4x}{x^3 + 3x^2 + 3x + 1} + \frac{4}{x^3 + 3x^2 + 3x + 1}$$

(%i3) (x+2)^2/(x+1)^3,expand(2,2);

```



```
(%o3)
```

$$\frac{x^2}{(x+1)^3} + \frac{4x^4}{(x+1)^3} + \frac{4}{(x+1)^3}$$

最初の例では、冪の次数が 1001 と maxposex の値 1000 を越えている為に展開を行いません。次の例では、expop に 2, expon に 3 を指定していますが、展開する式の冪が正の冪の次数の絶対値が 2, 負の冪の絶対値が 3 の為、展開を実行しています。しかし、expop に 2, expon に 2 を指定すると、指定した値が負の冪の次数の絶対値よりも小さい為に、負の冪乗の部分式の展開のみが実行されません。

noun の場合は与式を名詞型として扱う為に、式の評価を行いません。

numer の場合、大域変数 numer と大域変数 float を true にして式の評価を実行します。

```
(%i45) sin(%pi/10);
(%o45)
          %pi
        sin(---)
          10

(%i46) sin(%pi/10),numer;
(%o46)
.3090169943749474

(%i47) 2*%e*x+%pi/4,numer;
(%o47)
5.43656365691809 x + .7853981633974483

(%i48) 2*%e^x+%pi/4,numer;
(%o48)
          x
        2 %e  + .7853981633974483
```

猶、式の中に冪乗でない定数 %e が存在する場合は、大域変数 %enumer を true にして式の評価を行っています。但し、%e の冪の場合は %e を浮動小数点に変換しません。

risch は integrate 関数に Risch 積分を実行させます。これは、integrate 関数内部で、risch が指定される場合、rischint 関数を用い、それ以外は sinit 関数を用いる様になっている為です。

derivlist は関数に似た構文を持っています。

```
derivlist
derivlist(⟨変数1⟩, ..., ⟨変数k⟩)
```

ev 関数は derivlist で指定した変数 $\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle$ を含む名詞型の微分を評価します.

```
(%i9) a1:'diff('diff(x^2+2*x*y^2+y^4,x),y)$
(%i10) a1,diff;
(%o10)

$$4 y^3$$

(%i11) a1,derivlist(x);
(%o11)

$$\frac{d}{dy} (2 y^2 + 2 x)$$

(%i12) a1,derivlist(y);
(%o12)

$$\frac{d}{dx} (4 y^3 + 4 x y)$$

```

この derivlist 引数に似たものに local 引数があります. この local も derivlist に似た構文を持ちます,

$$\text{local}(\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle)$$

こちらは ev 関数内部で利用する局所変数の宣言に用いられます. 但し, ev 関数では後述の式への割当がある為, local でなければ局所変数が扱えない訳ではありません.

```
(%i16) solve(x^3+2*x-b,x),local(b),b=3;
(%o16)

$$[x = -\frac{\sqrt{11} i + 1}{2}, x = \frac{\sqrt{11} i - 1}{2}, x = 1]$$

(%i17) solve(x^3+2*x-b,x),b=3;
(%o17)

$$[x = -\frac{\sqrt{11} i + 1}{2}, x = \frac{\sqrt{11} i - 1}{2}, x = 1]$$

```

式の割当は, x=1 や x:1 等によって評価する式の変数に値を割当てて事で与式の評価を実行するものです.

```
(%i31) ev(sin(x),x=1);
```

```
(%o31)                                sin(1)
(%i32) ev(sin(x),x=1,float);
(%o32)                                sin(1)
(%i33) ev(sin(x),x=1,bfloat);
(%o33)                                8.414709848078965B-1
(%i34) ev(sin(x),x=1);
(%o34)                                sin(1)
(%i35) ev(sin(x),x=1,bfloat);
(%o35)                                8.414709848078965B-1
(%i36) ev(sin(x),x:%pi/4,bfloat);
(%o36)                                7.071067811865475B-1
(%i37) ev(sin(sqrt(x^2+y^2)),[x:%pi/4,y=1]);
                                         2
                                         %pi
(%o37)                                sin(sqrt(---- + 1))
                                         16
```

この例では、 $\sin(x)$ の変数 x に 1 や $\frac{\pi}{4}$ 等を割当てています。割当は演算子=でも演算子:でも構いません。複数の変数に一度に変数を割当てる場合はリストで与えます。この際に、演算子=:が混在していても問題はありません。

この評価方法は、algsys 関数や solve 関数等の方程式を解く関数と評価したい式を組合せると強力です。

```
(%i42) algsys([x^5-x^3+5],[x]);
(%o42) [[x = - 1.53955007256894], [x =
- 1.183445980013718 %i - .4590933961159689],
[x = 1.183445980013718 %i - .4590933961159689],
[x = 1.228868436016586 - .7109481105485196 %i],
[x = .7109481105485196 %i + 1.228868436016586]]
(%i43) map(lambda([z],ev(realpart(x^2),z)),%);
(%o43) [2.37021442594703, - 1.189777641253336,
- 1.189777641253336, 1.004670417145341, 1.004670417145341]
```

この例では、方程式 $x^5 - x^3 + 5 = 0$ の数値近似解を求め、その近似解を二乗したものの実部を求めています。

detout は大域変数 detout と似た結果が得られます。detout を指定した場合、大域変数


```
(%i67) test:%pi/4;
                                     %pi
(%o67)                               ---
                                     4

(%i68) ''%i66;
                                     1/2
                                     2
(%o68)                               ----
                                     2

(%i69) ''sin(test);
(%o69)                               .7071067811865475
```

eval 関数は〈式〉の評価を行います.LISP の eval 関数と同じ働きをします.

5.8.4 関数や演算子に影響を与える大域変数の表示

Maxima の関数や演算子には様々な属性や大域変数による制御が加えられています. 属性に関しては properties 関数で表示可能ですが, 影響を与える大域変数の一覧を表示する事が可能な関数として, options 関数があります.

options 関数の構文

```
options()
options(< 記号 >)
```

options 関数は引数を与えなければ, フロントエンドを立ち上げて, 数字入力で下の階層に進む事が出来ます. 直接, 関数名や大域変数名を入力しても構いません. 猶, 入力行の末尾には; や\$を入れます. そして, このフロントエンドから抜ける為には, `exit;` と入力します.

```
(%i14) options();
'options' interpreter (Type 'exit' to exit.)
1 - INTERACTION
2 - DEBUGGING
3 - EVALUATION
4 - LISTS
```

```

5 - MATRICES
6 - SIMPLIFICATION
7 - REPRESENTATIONS
8 - PLOTTING
9 - TRANSLATION
10 - PATTERN-MATCHING
11 - TENSORS
:
sin;
1 - FLOAT (C)
2 - NUMER
3 - BFLOAT (C)
4 - %PIARGS (S)
5 - %IARGS (S)
6 - TRIGINVERSES (S)
7 - TRIGEXPAND (C S)
8 - EXPONENTIALIZE (S)
9 - HALFANGLES
10 - TRIGSIGN (S)
11 - LOGARC (S)
:
exit$
(%o14)                                     done

```

option 関数の引数として, 直接, 関数名や演算子名を指定する事も可能です. この場合, options 関数は関数や演算子の属性や, 影響を与える大域変数の名前で構成されたリストを返却します.

```

(%i1) options(".");
(%o1) [dotassoc, dotscrules, dotconstrules, dotexptsimp, dotdistrib,
                                             assumescalar]

(%i2) options(sin);
(%o2) [float, numer, bfloat, %piargs, %iargs, triginverses, trigexpand,
                                             exponentialize, halfangles, trigsign, logarc]

```

猶, 演算子名を指定する場合は二重引用符で括る必要がありますが, 関数や大域変数の場合は, 二重引用符で括る必要はありません.

5.9 LISP に関係する函数

5.9.1 Maxima と LISP

Maxima は Common Lisp と呼ばれる LISP の一方言で記述されています。LISP は函数型と呼ばれるプログラム言語で、プログラムは様々な函数を定義して、それらを組合せて行く作業とも言えます。因みに、C や FORTRAN は手続型と呼ばれる言語です。

Maxima はこの LISP の上で動作する環境ですが、Maxima 自体は PASCAL 風の処理言語を持っており、構文的にも LISP を意識する事は単純な利用では殆どありません。但し、Maxima で酷いエラーを出すと LISP のデバッガに落ちる事があります。LISP のデバッガからの抜け方は、Maxima を実装した LISP によって微妙に異なりますが、CLISP の場合は `:q` と入力してみてください。すると Maxima に戻ります。

LISP の特徴は言語仕様が非常に柔軟な点です。LISP には原子 (アトム) と呼ばれる項があり、それらを空行で区切って小括弧 () で括ったリストと呼ばれるデータが、最も基本的なデータとなります。このリストはリストのリストといったものも許容します。この原子とリスト等で構成されたデータを S 式と呼びます。因に、LISP のプログラム自体も S 式です。その為、LISP の函数でプログラムを操作する事も容易に行えます。

その為、Maxima のプログラムでは足りない所を LISP で代用する事も多くあります。又、Maxima から LISP を直接利用する事も可能です。この場合は Maxima の `to_lisp` 函数を利用します。Maxima で `to_lisp();` と入力すると、裏方の LISP が表に出ます。

これで LISP を使って遊べます。この状態から Maxima に戻りたければ `(to-maxima)` と入力します。すると通常の Maxima に戻ります。

```
(%i1) to_lisp();
type (to-maxima) to restart, ($quit) to quit Maxima.
```

```
Maxima> (setq $a '1)
1
Maxima> (to-Maxima)
returning to Maxima
(%o1)                                     true
(%i2) a;
(%o2)                                     1
```

この例では `to_lisp();` で LISP に入って項 `$a` に 1 を割当てています。それから `(to-maxima)` で Maxima に戻っています。to_lisp 函数の返却値は true です。最後に `a;` を入力する

と,LISP で \$a に割当てた値 1 が返却されます. これは Maxima での原子の内部表現では \$a の様に先頭に \$ が付くからです.

この例で注目して頂きたい事は,Maxima で表示されているものと LISP 側で見たものと様子が違う事です. 即ち,Maxima で扱うデータ, 函数それ自体も含めて LISP の別の表記方法があります. この LISP 側での表現を単純に内部表現と呼んでいます. この内部表現を通常の処理で気にする事は殆どありませんが, 細かな処理を行う必要が出た時点で初めて意識する事になります.

猶,Maxima の函数名で先頭に ? が付いているものが幾つか存在します. この様な函数は ? を外した部分は LISP の函数で,Maxima から裏の LISP で処理させて, その結果を Maxima 側に持って来る函数です. ? は通常の LISP の函数にも適応可能で ? を頭に付けた函数は,Maxima 内部では ? を外して,LISP の函数として処理されます. この様に Maxima が介在する為, ? を用いて LISP の函数を利用する場合には, 引数は Maxima で見えているものを設定します.

又, ? を変数に付ける事で,LISP 側の変数に束縛された値を参照したり, LISP 側の変数に値を束縛する事も可能です.

```
(%i3) ?neko:'123;
(%o3)                                     123
(%i4) :lisp neko
123
(%i4) ?neko;
(%o4)                                     123
```

この例では, ?neko:123 で LISP 側の変数 neko に値 '123 を束縛させています. この操作は LISP 側の (setq neko '123) に相当します.

猶, 関連する大域変数に, 大域変数 lisplisp disp があります. この lisplisp disp は LISP 側の変数を表示する時に ? を表示するかどうかを制御する大域変数で, true の場合のみ, ? を付けて表示させます.

ここで LISP の函数や変数を利用する為に, ? を使う場合, ? の直後に空行を入れると Maxima のオンラインマニュアルを呼出そうとするので, 注意が必要です.

? と似た函数に :lisp 演算子があります. こちらは, 直接 LISP の S 式を Maxima 側から入力し,LISP に評価させた値が得られる演算子です. ? 演算子との違いは, ? の被演算子が LISP の函数であっても, その LISP の函数の引数は Maxima に準じたものとなりますが, :lisp 演算子の場合, 被演算子は通常の LISP の S 式になります. 従って,Maxima の式を処理させる場合,Maxima の内部表現そのものを与えなければなりません.


```
(%i26) a:x+y+z;
(%o26)                z + y + x
(%i27) :lisp $a;
((MPLUS SIMP) $X $Y $Z)
(%i27) :lisp (car $a)
(MPLUS SIMP)
(%i27) ?car(a);
(%o27)                ("+", simp)
```

上記の例では変数 a に $x+y+z$ を割当てていますが、 $\$a$ が変数 a の内部変数名となります。`:lisp $a;` でこの変数に割当てた値を参照していますが、返却値は内部表現そのものです。この様な変数の参照は演算子 `?` では出来ません。

更に、`:lisp (car $a);` の値は内部表現そのものですが、`?car(a);` の値はそれを Maxima で解釈した `("+", simp)` となっており、引数に $\$$ が付いていない事と、`:lisp` 演算子の結果に `%o` ラベルが無い事に注目して下さい。

この様に、演算子 `?` は入出力で Maxima が介在する為、入出力の度に Maxima の評価を受ける事になりますが、`:lisp` 演算子の場合は直接操作となります。`:lisp` 演算子は Maxima で内部表現を確認する必要がある場合に特に便利です。

さて、ここまでは Maxima から LISP の関数を用いる話でしたが、逆に、Maxima の関数を裏で走っている LISP から利用する事も出来ます。この為に `mfuncall` 関数を用います。`mfuncall` 関数の引数は Maxima の関数名の頭に `'$` を付け、その後に引数を並べます。

```
MAXIMA> (mfuncall '$diff '$x '$x 1)
1
```

この例では、関数 x を `diff` 関数を使って変数 x で微分するものです。この様に、引数は全て内部表現に対応したものでなければなりません。従って、Maxima 上で引数を割当ててある状態でなければ、使い難いものです。現実的には Maxima 関数の虫取りや動作の確認で用いる程度でしょう。

第6章 Maximaの対象とその操作

Der erste

Dieses Buch leg' ich in Eure Hand.

Der zweite

Von mir erhaltet Ihr den Schlüssel.

Der Dritte

Diese Briefschaft macht es zu Eurem Eigentum.

Faust

Wie kommt ein solches Geschenk mir zu?

Die drei

Du bist der Meister!

第一の学生

この本を貴方のお手に.

第二の学生

これが本を開ける鍵

第三の学生

この書簡で貴方は法的に所有者だ.

ファウスト

どうして, この様な物を私に?

三人で

貴方が道に通じた者であるが故に!

Buzoni, Doktor Faust[58] より

この章では, Maxima の数値, 多項式, リスト, 配列, 集合, 行列, 及び文字列とそれらで構成される式について述べ, それらの直接処理を, 代入, 簡易化に分けて記述しています. Maxima の式の内部表現に関しては式の節を参照して下さい. この式の内部表現は代入処理や式の簡易化を行う上で非常に重要で, Maxima の要の一つです.

猶, 比較的規模の大きな数値行列の処理に関しては余程の理由でも無い限り, Maxima よりも Octave 等の数値行列処理に長じたソフトを利用される事を勧めます. MATLAB クローンの Octave に関しては, 15 章で数値行列の処理方法やファイル処理, Maxima とのインターフェイスに関して記述しているので, 必要があれば参照して下さい.

6.1 数値

6.1.1 Maxima で扱える数値について

Maxima で扱える数値には, 整数, 有理数, 浮動小数点数と多倍長浮動小数点数, そして, 複素数があります. この中で, 整数, 浮動小数点数と多倍長浮動小数点数は Maxima の原子ですが, 有理数と複素数は Maxima の原子となりません (§4.12.3 参照).

浮動小数点と整数や有理数の演算は, 型の変更を実行しない限り浮動小数点となります. 又, 多倍長浮動小数点数と浮動小数点数が混在した式の型は, 浮動小数点数になりますが, その際に警告が表示されます. 猶, 浮動小数点数から多倍長浮動小数点数への変換は `bigfloat` 関数を用います.

多倍長浮動小数点数に関連する大域変数に大域変数 `fpprec` があります. この大域変数 `fpprec` は多倍長浮動小数点数の桁数を定めます. 従って, 大域変数 `fpprec` の値を大きくすると, 精度も当然向上します. 又, 計算精度がそれなりに必要でも, 結果の表示では多くを必要としない場合, 大域変数 `fpprintprec` に表示に必要な桁数を設定すれば, この大域変数 `fpprintprec` に設定した桁数で多倍長浮動小数点数の数値表示が行われます.

複素数は実部と `%i` をかけた虚部との和で指定されます. 例えば, 方程式 $x^2 - 4x + 13 = 0$ の根を Maxima では `2+3*%i` と `2-3*%i` で表現します. 更に, 複素数の実部と虚部は夫々 `realpart` 関数と `imagpart` 関数で取り出せます. 複素数は整数, 有理数, 浮動小数点数や多倍長浮動小数点数の自然な拡張となっている為, 実部と虚部は, これらの数で表現されます.

この他の数に代数的整数もありますが, 代数的整数は最小多項式や大域変数を用いる為, §6.2 にて説明します.

6.1.2 数値に関連する大域変数

数値に関連する大域変数

変数名	初期値	取り得る値	概要
domain	real	[real,complex]	多項式の係数環を指定
float2bf	false	[true,false]	float → 多倍長浮動小数点数の際の警告の有無を指定
fpprec	16	正整数	多倍長浮動小数点数の桁数を指定
fpprintprec	0	正整数	多倍長浮動小数点数の表示桁数を指定
bfloat	true	[true,false]	bfloat 関数の表示を制御
bfloatat	false	[true,false]	多倍長浮動小数点数の有理数への変換を制御
m1pbranch	false	[ture,false]	-1 の原始 n 乗根自動変換の有無を指定
radexpand	true	[true,false]	根号の外に出すかどうかを指定

大域変数 domain は関数等の動作に影響を与えます. 大域変数 domain は初期値として real が設定されています. これは Maxima が主に実数上で処理を行う事を意味しています. ここで値を complex にすると, Maxima で処理する世界は複素数の世界となる事を意味します.

更に, 大域変数 domain を complex, 大域変数 m1pbranch を true にした場合, -1 の n 乗根は Gauss 平面上の点として自動的に変換されます. 例えば, $(-1)^{\frac{5}{7}}$ は $e^{\frac{5i\pi}{7}}$ に自動的に変換されます.

大域変数 float2bf は false の場合に bfloat 関数で浮動小数点数を多倍長浮動小数点数に変換する時点で計算精度が落ちるとの警告メッセージを表示させます.

大域変数 fpprec は多倍長浮動小数点数の桁数を定めます. 即ち, fpprec を正整数 n に設定すると多倍長浮動小数点数は n 桁になります.

大域変数 bfloat は多倍長浮動小数点数の表示を短縮して表示するかどうかを制御する大域変数です. bfloat が true の場合は短縮して表示しますが, false の場合は長く表示します.

大域変数 bfloatat は大域変数 ratepsilon と組合せて用いられる変数です. この bfloatat が false の場合, 大域変数 ratepsilon が多倍長浮動小数点数から有理数型への変換で用いられ, 大域変数 bfloatat が true の場合, 生成される有理数は多倍長浮動小数点数にな

ります。

大域変数 `m1pbranch` は -1 の原始 n 乗根の自動変換を制御する大域変数です。大域変数 `m1pbranch` が `true` で、大域変数 `domain` が `complex` の場合、原始 n 乗根が Gauss 平面上の点として自動的に変換されます。

大域変数 `radeexpand` は式 $\sqrt{a^2b}$ の様に根号の外に出せる因子が式に含まれる場合、`true` であれば、その様な因子を根号の外に自動的に出させる大域変数です。

6.1.3 Maxima の定数

Maxima の定数

<code>%e</code>	自然底 e
<code>%gamma</code>	Euler の定数
<code>%phi</code>	$\frac{1+\sqrt{5}}{2}$
<code>%pi</code>	円周率 π
<code>false</code>	Bool 代数の定数. 偽 (LISP の <code>nil</code>)
<code>true</code>	Bool 代数の定数. 真 (LISP の <code>t</code>)
<code>inf</code>	正の実無限大.
<code>minf</code>	負の実無限大.
<code>infinity</code>	複素無限大, 任意の偏角で無限大
<code>zeroa</code>	0_+ .limit 関数で利用
<code>zerob</code>	0_- .limit 関数で利用

Maxima には幾つかの定数があり、大雑把に 3 種類に分類出来ます。

最初の一つは `%pi` の様な通常の定数、もうひとつは、`t` や `nil` といった真偽値、最後に Maxima が用いる `inf` の様な定数です。更に、`zeroa` や `zerob` は `limit` 関数だけで用いる定数です。

```
(%i55) limit(1/x,x,zeroa);
(%o55)                                     inf
(%i56) limit(1/x,x,zerob);
(%o56)                                     minf
```

但し、`limit(1/(x-1),x,1,'plus)` の様に、`limit(1/(x-1),x,1+'zeroa)` とする使い方は出来ません。

猶, `inf` や `minf` といった極限に関連する数を含む式の評価は `limit` 関数で行えます. この場合は, `limit(〈式〉)` で式の評価が行えます.

6.1.4 数値に関連する関数

数値に関連する真理関数

関数名	true となる条件
<code>numberp</code>	整数, 有理数, 浮動小数点数や多倍長浮動小数点数
<code>bfloatp</code>	多倍長浮動小数点数
<code>floatnump</code>	浮動小数点数
<code>integerp</code>	整数
<code>evenp</code>	偶数
<code>oddp</code>	奇数
<code>primep</code>	素数
<code>constantp</code>	定数の場合

これらの真理関数は, 真理値が `{true,false}` で, 一つの引数を取ります.

`float_approx_equal` 関数と `bfloat_approx_equal` 関数

```
float_approx_equal(〈浮動小数点数1〉, 〈浮動小数点数2〉)
bfloat_approx_equal(〈多倍長浮動小数点数1〉, 〈多倍長浮動小数点数2〉)
```

これらの関数は, その真理値集合を `{true,false}` とする真理関数で, 共に, 二つの引数を取り, それらの数値の差が大域変数 `float_approx_equal.tolerance` を用いて計算した値よりも小さくなった場合に, 等しいと判断して `true` を返し, それ以外は `false` を返す関数です.

`max` と `min`

```
max(〈実数値1〉, 〈実数値2〉, ...)  〈実数値1〉, 〈実数値2〉, ... の最大値
min(〈実数値1〉, 〈実数値2〉, ...)  〈実数値1〉, 〈実数値2〉, ... の最小値
```

Maxima は与えられた実数列の最大値を `max` 関数, 最小値を `min` 関数で求めます. 猶, リストや集合に含まれる数値の最大値や最小値は `lmax` 関数と `lmin` 関数 で求める事が可能です. 猶, これらの関数は実質的に `max` 関数と `min` 関数です.

inrt

$$\text{inrt}(\langle \text{整数値}_1 \rangle, \langle \text{整数値}_2 \rangle)$$

inrt 関数は $\langle \text{整数値}_2 \rangle$ 乗が $\langle \text{整数値}_1 \rangle$ の絶対値を越えない最大の整数を返す関数です。猶, $\text{inrt}(x, 2)$ は内部表現自体が $\text{isqrt}(x, 2)$ と等しくなります。

```
(%i19) inrt(x^2, 2);
```

2

```
(%o19) isqrt(x )
```

```
(%i20) makelist(inrt(128,x),x,1,8);
```

```
(%o20) [128, 11, 5, 3, 2, 2, 2, 1]
```

整数変換に関連する関数

関数名	変換前	変換後
bfloat	全ての数	→ 多倍長浮動小数点数
isqrt	整数	→ 与えられた整数の平方根を越えない整数
fix	実数	→ 与えられた実数を越えない最大の整数
entier	実数	→ 与えられた実数を越えない最大の整数
floor	実数	→ 与えられた実数を越えない最大の整数
ceiling	実数	→ 与えられた実数を越える最小の整数
round	数値	→ 与えられた実数を四捨五入で丸めた整数

最初の bfloat 関数は全ての数と数値関数を多倍長浮動小数点数型に変換します。

isqrt 関数は引数 $\langle \text{整数} \rangle$ の絶対値の平方根を越えない整数値を返します。猶, isqrt 関数は内部で inrt 関数を用いており, $\text{isqrt}(x)$ は $\text{inrt}(x, 2)$ で処理しています。

```
(%i50) isqrt(-3);
```

```
(%o50) 1
```

```
(%i51) isqrt(-4);
```

```
(%o51) 2
```

```
(%i52) isqrt(10);
```

```
(%o52) 3
```

```
(%i53) isqrt(-10);
```

```
(%o53) 3
```

猶, isqrt 関数は引数が整数で無ければ, 入力式をそのまま返却します。

fix 関数, entier 関数と floor 関数は $\langle \text{数値} \rangle$ が実数の場合, $\langle \text{数値} \rangle$ を越えない最大の整数 n を返す関数で, 両者の違いは全くありません.

(%i42) fix(10);	
(%o42)	10
(%i43) fix(-10);	
(%o43)	- 10
(%i44) fix(10.5);	
(%o44)	10
(%i45) fix(-10.5);	
(%o45)	- 11
(%i46) entier(10);	
(%o46)	10
(%i47) entier(-10);	
(%o47)	- 10
(%i48) entier(10.5);	
(%o48)	10
(%i49) entier(-10.5);	
(%o49)	- 11
(%i50) floor(10);	
(%o50)	10
(%i51) floor(-10);	
(%o52)	- 10
(%i53) floor(10.5);	
(%o53)	10
(%i54) floor(-10.5);	
(%o54)	- 11

猶, この例の様に, 絶対値で越えない数を返すのではないので注意が必要です.
ceiling 関数は引数以上の整数で最小の整数を返す関数です.

```
(%i19) ceiling(10);
(%o19) 10
(%i20) ceiling(-10);
(%o20) - 10
(%i21) ceiling(10.5);
```

```
(%o21)                                     11
(%i22) ceiling(-10.5);
(%o22)                                     - 10
```

—— 複素数に関連する関数 ——

cabs	式	→	与式の複素数としての絶対値
realpart	式	→	与式の実部
imagpart	式	→	与式の虚部
carg	式	→	与式の偏角

cabs 関数, realpart 関数と imagpart 関数は各々複素数の絶対値, 実部と虚部を返す関数です. 一般の式に対しては, 展開した式の%i を含まない部分式を実部, %i を含む部分式を虚部とします.

carg 関数は与えられた複素数の偏角 θ を $\pi \geq \theta > -\pi$ の範囲で返す関数です.

```
(%i46) cabs(x+%i*y);
                                     2    2
(%o46)                               sqrt(y  + x )
(%i47) realpart(x+%i*y);
                                     x
(%o47)
(%i48) imagpart(x+%i*(z+%i*y));
                                     z
(%o48)
(%i49) carg(1+%i/2);
                                     1
(%o49)                               atan(-)
                                     2
(%i50) %,float;
(%o50)                               0.463647609000806
```

—— 数値の変換に関連する関数 ——

sqrt	式	→	与式の平方根
rationalize	式	→	式中の float 型や bfloat 型を有理数化

sqrt 関数は〈式〉が実数の場合はその平方根を返します. 猶,Maxima 内部で, 通常, $\langle \text{式} \rangle^{(1/2)}$ で表現されています. その為,substpart 関数等を用いて式の操作を行う場

合には注意が必要になります. 猶, 大域変数 `sqrtdispflag` を初期値の `false` から `true` に変更すると, `sqrt` ではなく $1/2$ の冪乗として表示されます.

```
(%i34) sqrt(x);
(%o34)                                sqrt(x)
(%i35) sqrtdispflag:false;
(%o35)                                false
(%i36) sqrt(x);
                                     1/2
(%o36)                                x
```

`rationalize` 関数は与えられた式に含まれる `float` 型や `bfloat` 型の数値を有理数に変換する関数です. 式は通常の式やリストが扱えます. 猶, 与式がリストや集合であれば, `fullmap` 関数で `rationalize` 関数が作用させられる仕組みになっています.

```
(%i54) rationalize({1,2.03,3.0,4.0});
                                     203
(%o54)                                {1, ---, 3, 4}
                                     100
(%i55) rationalize({1,2.03,3.0,4.0,{x^4+1.03}});
                                     203          4    103
(%o55)                                {1, ---, 3, 4, {x  + ---}}
                                     100          100
(%i56) rationalize([1,2.03,3.0,4.0,{x^4+1.03}]);
                                     203          4    103
(%o56)                                [1, ---, 3, 4, {x  + ---}]
                                     100          100
```

—— 疑似乱数に関連する関数 ——

```
make_random_state(< 正整数 >)
make_random_state(< 乱数状態 >)
make_random_state(true)
make_random_state(false)
set_random_state(< 乱数状態 >)
random(< 数値 >)
```

Maxima の疑似乱数生成では, 松本眞と西村拓士によって開発された Mersenne Twister 法 (略して MT 法¹) が用いられています.

`make_random_state` 関数は乱数状態を生成する関数です.

`set_random_state` 関数は乱数の生成で利用する乱数状態を指定する関数で, その引数は乱数状態です.

`random` 関数は, 引数として〈数値〉を取り, 0 から〈数値〉-1 の間の整数乱数を返す関数です.

—— LISP 由来の数値関数 ——

?round 〈数値〉の最も近い整数

?truncate 〈数値〉の小数点以下を切り捨て

これらの数値関数は LISP の関数をそのまま利用します. その為, 先頭に?が付きます. ?round 関数は〈数値〉を最も近い整数に丸めます. ここで, 引数は浮動小数点数であって, 多倍長浮動小数点数ではありません.

?truncate 関数は浮動小数点数の〈数値〉を引数とし, 小数点以下を切り捨てます.

¹もう一つの由来は Makoto Takushi です.
<http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/name.html> を参照

6.2 多項式

この節では,Maxima の多項式の表現について述べます. 猶, 数学的な事項や, 順序, 式の表現に関しては,§4 を参照して下さい.

6.2.1 多項式の一般表現

Maxima の多項式は通常の C や FORTRAN で記述する $x^2+3*x*z+4$ や $x**2+3*x*z+4$ の様な書式で入力されます. ところが, 内部表現と表示は入力式そのままではありません. 大域変数の設定に従って簡易化を行ない,Maxima の項順序 $>_m$ に従って項内部の変数や項自体の並び換えを実行したものが返されます.

以下に, 入力式と内部式の例を示します.

```
(%i28) a:x+y+z;
(%o28)                                     z + y + x
(%i29) :lisp $a;
((MPLUS SIMP) $X $Y $Z)
(%i29) b:z+x+y;
(%o29)                                     z + y + x
(%i30) :lisp $b;
((MPLUS SIMP) $X $Y $Z)
(%i31) c:(1+2)*x+3*y+(2+1-2)*z-z;
(%o31)                                     3 y + 3 x
(%i32) :lisp $c;
((MPLUS SIMP) ((MTIMES SIMP) 3 $X) ((MTIMES SIMP) 3 $Y))
(%i33) a1*x+a2*x;
(%o33)                                     a2 x + a1 x
(%i34) d:x1^2*x8^2*x3;
(%o34)                                     2      2
                                     x1  x3 x8
(%i35) :lisp $d;
((MTIMES SIMP) ((MEXPT SIMP) $X1 2) $X3 ((MEXPT SIMP) $X8 2))
```

この例では多項式 $x+y+z$ と $(1+2)*x+3*y+(2+1-2)*z-z$ の処理を示しています. 最初に変数 a に $x+y+z$ を割当てています. `:lisp $a;` で変数 a の内部表現を参照すると, `((MPLUS SIMP) $X $Y $Z)` が返却されます. この S 式の頭にある `(MPLUS SIMP)` は

式の主演算子が和+である事を示し、その後に被演算子となる項が並んでいます。この様に Maxima の多項式の内部表現でも先頭に演算子が置かれる前置式表現になっています。猶、項の並びは同じ式であれば項や変数の順番を変更しても同じ結果になります。この例で示す様に $x+y+z$ でも $z+x+y$ と入力しても同じです。これは Maxima の項順序 $>_m$ に従って与えられた多項式の項を構成する変数や項を並べ替えているからです。

この項順序 $>_m$ は基本的に逆アルファベット順で変数を並べる順序です。順序一般に関しては §4.8, 順序 $>_m$ に関しては §5.2 を参照して下さい。多項式の内部表現では、項に対して、その項を構成する変数を順序 $>_m$ に対して小さいもの順に並べ、式を構成する項に対して、順序 $>_m$ で小さな項で並べます。最初の例の式 $x+y+z$ の内部表現リストは $(+ \ x \ y \ z)$ となります。これは、項順序が $z >_m y >_m x$ となる為で、項を順序 $>_m$ に従って小さいものから順に並べたからです。次に、単項式 $x^2 x^8 x^3$ の場合、単項式は $x^2 x^3 x^8$ と各変数が並べ替えられています。これは、 $x^8 >_m x^3 >_m x^1$ となる為、Maxima 内部で変数を x^1, x^3, x^8 の順に並べ替えた為です。更に、 x^4 の様な変数と数値で構成された項の場合、数値が $>_m$ の下位に来る為に、 $4*x$ と数値と変数が入れ替えられます。

以下に、多項式と単項式の一般表現について纏めておきましょう。

多項式と単項式の一般表現

多項式	$((\text{mplus simp}) \text{項}_1 \cdots \text{項}_m)$	$\text{項}_m >_m \cdots >_m \text{項}_1$
単項式	$((\text{mtimes simp}) \text{数値} \text{変数}_1 \text{の冪} \cdots \text{変数}_n \text{の冪})$	$\text{変数}_n >_m \cdots >_m \text{変数}_1$

式の表示では、この内部表現を基に式の表示を行います。多項式の場合、大きな項から順番に表示されます。その為、 $x+y+z$ と入力しても $z+y+x$ と表示されます。ところが項に関しては、大きな変数からではなく、小さい順に並べたままに表示されます。その為、項 $x^2 x^8 x^3$ は変数を順序 $>_m$ に従って並べ替えられた $x^2 x^3 x^8$ で表示されます。

では $x+y, x-y, x*y, x/y, x^y$ についても :lisp 演算子を使って、その内部表現を調べてみましょう。

```
(%i33) t0:x+y;
(%o34)
y + x
(%i34) :lisp $t0;
((MPLUS SIMP) $X $Y)
(%i35) t1:x-y;
(%o35)
x - y
```

```

(%i35) :lisp $t1;
((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y))
(%i36) t2:x*y;
(%o36)
x y
(%i37) :lisp $t2;
((MTIMES SIMP) $X $Y)
(%i38) t2:x/y;
(%o38)
x
-
y
(%i39) :lisp $t3;
((MTIMES SIMP) $X ((MEXPT SIMP) $Y -1))
(%i40) t4:x^y;
(%o40)
y
x
(%i41) :lisp $t4;
((MEXPT SIMP) $X $Y)

```

Maxima では $x-y$ が $x+(-y)$, x/y が $x^{(-y)}$ に置換されている事が判ります. この様に Maxima の内部では可換積や可換積の冪と和を用いて多項式が表現されます. この様に変換しておく事で, 差の処理や商の処理の手間を減らす事が出来るのです. この一般表現に対し, Maxima には内部表現をより簡潔にし, 係数を有理数に変換した正準有理式表現 (Canonical Rational Expressions, 略して CRE 表現) もあります.

6.2.2 多項式の CRE 表現

CRE 表現は factor 関数や ratsimp 関数等々で内部的に利用されるもので, 利用者はこの表現をあまり意識する必要はありません. この CRE 表現は本質的に展開された多項式や有理式関数に適したリストによる表現の一つです.

最初に 1 変数多項式 $\langle \text{変数} \rangle^{\langle \text{次数}_1 \rangle} + \langle \text{変数} \rangle^{\langle \text{次数}_2 \rangle} + \dots$ の CRE 表現は以下の書式で与えられます.

—— 単変数多項式の CRE 表現 ——

$(\langle \text{変数} \rangle \langle \text{次数}_1 \rangle \text{係数}_1 \langle \text{次数}_2 \rangle \text{係数}_2 \dots)$

猶, この CRE 表現では, 次数に関し, $\langle \text{次数}_1 \rangle > \dots > \langle \text{次数}_2 \rangle > \dots$ を満しています.

有理数係数の1変数多項式とそのCRE表現は項順序 $>_m$ により一対一に対応します。これを多項式 $3x^2 - 1$ を使って説明しましょう。この多項式はMaxima内部では、 $3x^2 + (-1)x^0$ で表現されます。これをλ式風に考えれば、 $\lambda x \cdot (3x^2 + (-1)x^0)$ となりますね。以上から、係数と次数の対から構成されるリストは $((2\ 3)\ (0\ -1))$ になります。ここでは予め変数が x であるとして処理していたので、変数の情報を落しても問題はありません。しかし、一般的には変数を明確にしておくべきです。そこで、リストの先頭に x を入れてみましょう。すると $(x\ (2\ 3)\ (0\ -1))$ となります。更に、リスト中の小括弧を外して $(x\ 2\ 3\ 0\ -1)$ としても多項式 $3x^2 + (-1)x^0$ の復元には問題ありません。

多変数多項式の場合も同様にCRE表現で書換える事が出来ます。但し、1変数の場合に様に平坦なリストでCRE表現は表現されず、CRE表現のリストによる複合リストになります。多変数の場合で重要な事は変数の間に順序を入れる事です。Maximaの場合、辞書式順序を基礎にした順序 $>_m$ が入っています。この順序の詳細は§5.2を参照して下さい。

Maximaのデフォルトの順序 $>_m$ の変数順序の要点だけを述べると、アルファベットの大きい文字が小さい文字よりも大きく、アルファベットの中では z が一番大きく a が一番小さいという逆アルファベット順で順番が付けられています。変数が二文字以上の場合、先頭の文字から順番に比較します。もし、先頭の文字が等しければ次の文字で比較します。途中でMaximaの順序 $>_m$ で大小関係がつくと、その順序で変数に順番が入ります。例えば、 xxz と xyy の場合は頭の二つが x なので順序はまだ決まりませんが、最後の z と y に関して、 $z >_m y$ となるので、最終的に、 $xxz >_m xyy$ となります。

さて、多変数多項式のCRE表現に戻りますが、この場合は、再帰的な考えで処理します。ここでは例として $2xy + x - 3$ で考えてみましょう。まず、この多項式を x の多項式と看做すと $(2y + 1)x - 3$ となるので、CRE表現の第一段目は $(x\ 1\ 2y+1\ 0\ -3)$ となりますね。但し、第二成分の $2y+1$ はCRE表現ではないので、これをCRE表現に変換した $(y\ 1\ 2\ 0\ 1)$ で置換える必要があります。以上から $(x\ 1\ (y\ 1\ 2\ 0\ 1)\ 0\ -3)$ が求めるCRE表現となります。次に、 y の多項式として考えると、 $2xy + x - 3$ なので、中間的には $(y\ 1\ 2x\ 0\ x-3)$ 中の x の式をCRE表現に置換えると、最終的に、 $(y\ 1\ (x\ 1\ 2)\ 0\ (x\ 1\ 1\ 0\ -3))$ が得られます。

この様に多変数の場合、CRE表現は順序をあやふやにしていると、表現が一意に定まるとは限りません。変数に順序を入れて大きな変数順に式を括れば、一意に定まります。Maximaでは主変数として宣言された変数が式に含まれている場合は、その主変数の多項式と看做し、それ以外の変数に対してはMaximaの変数順序 $>_m$ を用いて順序を入れます。式に主変数として宣言された変数が存在しなければ、順序 $>_m$ に対して最高位の変数の式の中の変数を主変数としてCRE表現を構築します。

ここで、二つの CRE 表現が与えられた時に、それらの変数順序が異なれば、処理は非常に厄介な事になるのが判るかと思います。その為、CRE 表現を用いる函数に関しては、後の事も考えて主変数の設定を行う必要があります。

又、CRE 表現の変数には、通常の算術演算子 (+, -, *, /) や整数冪 (^) を持たないものを与えます。その為、 y^2 の様な式は変数に使えませんが、 $\log(x)$ や $\cos(x+1)$ の様な函数は使えます。

Maxima の項順序は基本的に辞書式順序です。猶、局所的な変数の順序の変更が `ratvars` 函数等で指定出来ます。

6.2.3 一般表現と CRE 表現への変換を行う函数

Maxima には与えられた式を一般表現から CRE 表現、CRE 表現から一般表現に変換する函数が用意されています。

一般表現と CRE 表現への変換に関連する函数

<code>rat(<式>, <変数₁>, ..., <変数_n>)</code>	一般表現	⇒	CRE 表現
<code>ratdisrep(<式>)</code>	CRE 表現	⇒	一般表現
<code>totaldisrep(<式>)</code>	CRE 表現	⇒	一般表現

先ず、最初の `rat` 函数は与えられた <式> を展開し、浮動小数点数を `ratepsilon` で指定された許容範囲以内の有理数に変換し、全ての項を共通の分母で纏め、分子と分母の最大公約因子を除去します。ここで変数の順序は無指定であれば、Maxima の順序 $>_m$ に従いますが、`ratvars` 函数を用いて導入された順序があれば、その順序を用います。`rat` 函数は +, -, *, / と冪乗の他の函数を一般的には簡易化しません。CRE 表現での原子は一般形式のものと異なります。従って、`rat(x)-x` は 0 と異なる内部表現の `rat(0)` で計算されます。

```
(%i1) exp1:rat(x)-x;
(%o1) R/
0
(%i2) :lisp $exp1;
((MRAT SIMP ($X) (X13157)) 0 . 1)
(%i2) exp0:0;
(%o2)
0
(%i3) :lisp $exp0;
0
```

この `rat` 函数には `ratfac`, `ratprint`, `keepfloat` といった直接動作に関連する大域変数があります.

`ratdisrep` 函数は CRE 表現から一般表現に引数を変換します.

`totaldisrep` 函数は CRE 表現から一般表現に $\langle \text{式} \rangle$ の全ての部分式を変換します.

6.2.4 有理式の CRE 表現

有理式は多項式の分数ですが, 有理式の CRE 表現の表現は, 多項式の分母と分子に共通因子が無く, 分母の筆頭項 (leading term) の係数を正にしたものとなります.

以下に実例を示しましょう.

```
(%i1) r1: rat((y-1)/((y-x)*z^2+1));
```

```
(%o1)/R/
```

$$\frac{y - 1}{(y - x) z^2 + 1}$$

```
(%i2) r2: rat((y-1)/((x-y)*z^2+1));
```

```
(%o2)/R/
```

$$-\frac{y - 1}{(y - x) z^2 - 1}$$

```
(%i3) r3:rat((y-1)/(-(y-z)*x^2+1));
```

```
(%o3)/R/
```

$$\frac{y - 1}{x^2 z - x^2 y + 1}$$

```
(%i4) :lisp $r3;
```

```
((MRAT SIMP $(X $Y $Z) (X13180 Y13181 Z13182)) (Y13181 1 1 0 -1)
Z13182 1 (X13180 2 1) 0 (Y13181 1 (X13180 2 -1) 0 1))
```

```
(%i4) t3:(y-1)/(-(y-z)*x^2+1);
```

```
(%o4)
```

$$\frac{y - 1}{x^2 (z - y) + 1}$$

```
(%i5) :lisp $t3;
((MTIMES SIMP)((MPLUS SIMP) -1 $Y)
((MEXPT SIMP)
((MPLUS SIMP) 1
((MTIMES SIMP)((MEXPT SIMP) $X 2)((MPLUS SIMP)
((MTIMES SIMP) -1 $Y) $Z)))
-1))
(%i5)
```

この例では変数順序 $>_m$ が逆アルファベット順の為、変数の順序は $z >_m y >_m x$ になります。その為、 z が主変数となり、式は z の多項式として纏められます。最初の二つの例では、 $(x-y)*z^2$ は、 $y >_m x$ となる為に、自動的に $-(y-x)*z^2$ に並び替えられてしまいます。この際に、最も順序の高い項となる $y* z^2$ の係数を正にする為に、必要に応じて -1 がかけられています。この例で示す様に、式が CRE 表現、或いは CRE 表現の部分式を含む場合、記号 $/R/$ が行ラベルに続きます。

`:lisp $r3;` で CRE 表現の内部表現を表示しています。先頭の MRAT で CRE 表現である事を示し、その後のリストで有理式の変数が X, Y, Z 、これらの変数に対応する内部変数が $X13180, Y13181$ と $Z1382$ である事を示しています。ここで内部変数は LISP の gensym 関数で生成されるもので、後ろの番号は生成順に付けられるものです。猶、有理式の変数は Maxima の showvar 関数を使って取り出す事が可能ですが、それらの変数に対応する内部変数は Maxima の内部変数の genvar に登録され、参照する場合には `:lisp genvar` の様にします。

又、変数リストの順版は順序 $>_m$ に対して小さい順に左から並ぶ為、左端が最小で右端が主変数となります。その為、この式では Z が主変数となります。この様な変数リストを含むリスト (MAT ...) の後に分子となる式が来ます。この式は先頭が $Y13181$ の為、変数 Y の多項式で、後の $1\ 1\ 0\ -1$ から次数が 1 でその係数が 1 の項と、次数が 0 で係数が -1 である事が判ります。最後の副リストが分母の式となり、先頭が $Z13182$ となっているので、主変数 Z の多項式である事が判ります。以降、分子の時と同じ様に読めば良いのです。但し、式は Maxima の変数順序 $>_m$ に従って読む必要があります。

この例では $Z >_m Y >_m X$ の順なので、変数 Z が式中にあれば、 Z の多項式として表現し、 Z がなくて変数 Y があれば Y の多項式、そして、変数 Z と Y の両方が無ければ変数 X の多項式と、帰納的に解釈します。

これに対し、同じ多項式の一般表現を最後に示しますが、非常に長いものになっている事が判ります。

猶、CRE 表現で分母が整数の場合 (CRE 表現では、浮動小数点数は有理数で近似され

ます), CRE 表現の内部表現は少し変化します.

```
(%i4) r1:rat((x-1)/5);
```

```
(%o4)/R/
```

$$\frac{x - 1}{5}$$

```
(%i5) :lisp $r1;
```

```
((MRAT SIMP $(X) (X13157)) (X13157 1 1 0 -1) . 5)
```

この例で示す様に分子は $x-1$ ですが, 分子と分母の間に記号. が入っています. CRE 表現では分子と分母の間に, 分母が整数の時に限って記号. を入れています. これは CRE 表現の生成で LISP の cons 関数が用いられている事を示しています. また変数 X の後に数値が入っていますが, これは LISP 内部の処理で変数 X に対応するシンボルを生成した際に割当てられた通し番号です.

拡張 CRE 表現は taylor 級数を表現する為に用いられています. 有理式の表記は正の整数では無く, 正か負の有理数となる様に拡張されており, 係数はそれ自身, 多項式と云うよりは, 上で記述した様に有理式となっています. これらは内部的に再帰的な多項式形式によって表現され, 多項式形式は CRE 表現に類似していますが, より一般化したものです. 猶, 切り捨てられる次数の様な情報も追加されています.

式を表示する際に, 拡張 CRE 表現の場合は記号/T/が式の行ラベルに続きます.

```
(%i1) t1:taylor(exp(x),x,0,5);
```

```
(%o1)/T/
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \dots$$

```
(%i2) :lisp $t1;
```

```
((MRAT SIMP ((MEXPT SIMP) $E $X) $X) (%e^x13162 X13163)
 ($ (X ((5 . 1)) 0 NIL X13163 . 2)) TRUNC)
 PS (X13163 . 2) ((5 . 1)) ((0 . 1) 1 . 1)
 ((1 . 1) 1 . 1) ((2 . 1) 1 . 2)
 ((3 . 1) 1 . 6) ((4 . 1) 1 . 24) ((5 . 1) 1 . 120))
```

雰囲気は CRE 表現に近いものですが, 係数と冪のリストの書式が cons で結合されたリストであり, 他に, リストの第一成分に Taylor 展開の様々な情報が追加されている事が判ります.

CRE 表現変換に関連する大域変数

変数名	初期値	概要
keepfloat	false	実係数の有理数への近似を制御
ratepsilon	2.0E-8	実係数の有理数近似する際の誤差を指定
ratalgdenom	true	代数的整数を分母とする項の有理化を制御
ratprint	true	CRE 表現変換時のメッセージを制御

大域変数 keepfloat が true であれば、浮動小数点数を含む式が CRE 表現に変換される際に、浮動小数点数が有理数に近似変換される事を防ぎます。猶、浮動小数点数が有理数に近似される際に生じる誤差は大域変数 ratepsilon で制御されます。

大域変数 ratepsilon は式を CRE 表現に変換する際に、係数を有理数に変換する時に用いられる許容範囲となります。大域変数 ratepsilon よりも小さな浮動小数点数は無視されます。浮動小数点数を有理数に変換したくなければ、大域変数 keepfloat を true に設定します。

```
(%i30) ratepsilon;
(%o30)                                     2.0e-8
(%i31) ratsimp((1+2.0e-8)*x);

rat replaced 1.00000002 by 1//1 = 1.0
(%o31)                                     x
(%i32) ratsimp((1+2.0e-7)*x);

rat replaced 1.0000002 by 5000001//5000000 = 1.0000002
(%o32)                                     5000001 x
                                     -----
                                     5000000
```

この例で示す様に、ratsimp 関数に作用させた場合に大域変数 ratepsilon よりも小さな数が無視され、浮動小数点数が有理数に変換されている事が判ります。

大域変数 ratalgdenom は式中に代数的整数を項の分母として持つ式に対し、大域変数 ratalgdenom が true の場合に、その分母を有理化します。これを実行する為には、大域変数 algebraic を true に設定して、式を CRE 表現に変換しておく必要があります。

```
(%i16) algebraic:true;
(%o16)                                     true
```

```
(%i17) ratalgdenom:true;
(%o17) true
(%i18) rat(1/sqrt(2)*x^2+1);
          2
      sqrt(2) x  + 2
(%o18) /R/ -----
          2

(%i19) ratalgdenom:false;
(%o19) false

(%i20) rat(1/sqrt(2)*x^2+1);
          2
      x  + sqrt(2)
(%o20) /R/ -----
      sqrt(2)
```

この例で示す様に, 大域変数 `algebraic` と大域変数 `ratalgdenom` を同時に `true` にすると, 分母に $\sqrt{2}$ を持つ式の分母が有理化されている事に注意して下さい.

大域変数 `ratprint` が `false` であれば, 浮動小数点数の有理数への変換を報せるメッセージ出力を抑制します.

CRE 表現の式の処理に関連する大域変数

変数名	初期値	概要
<code>ratdenomdivide</code>	<code>true</code>	分子の項の分離を制御
<code>ratexpand</code>	<code>false</code>	CRE 表現の展開を制御
<code>ratfac</code>	<code>false</code>	CRE 表現式の因子分解を制御
<code>ratsimpexpons</code>	<code>false</code>	<code>ratsimp</code> の自動実行を制御
<code>ratwtlvl</code>	<code>false</code>	近似の際の切捨てを制御
<code>ratweights</code>	<code>[]</code>	重みのリスト
<code>rootsconmode</code>	<code>true</code>	<code>rootscontract</code> 関数を制御
<code>psexpandnd</code>	<code>false</code>	CRE 表現の展開を制御

大域変数 `ratdenomdivide` が `false` であれば, `ratexpand` 関数を作動させた式に対して, 分子の項を分離する事を抑制します.

大域変数 `ratexpand` が `true` であれば, それらが一般形式に変換されるか表示された時に CRE 式が展開されます.

大域変数 `ratfac` が `true` であれば, CRE 有理式に対して部分的に因子分解された形式で出力します. 有理的操作の間, `factor` 関数を実際に呼ばずに, 式を可能な限り因子分解します. これでメモリ空間を節約し, 計算時間を幾らかを節約します. 有理式の分子と分母は互いに素とします.

例えば, `rat((x^2 - 1)^4 / (x + 1)^2)` は $(x-1)^4(x+1)^2$ になりますが, 各部分の因子は互いに素とは限りません.

大域変数 `ratfac` と大域変数 `ratweights` の手法は互換性が無いので, 両者を同時に使ってはいけません.

大域変数 `ratsimpexpons` が `true` であれば, 簡易化中に式の冪に対し, 自動的に `ratsimp` 関数が実行されます.

大域変数 `ratwtlvl` は `ratweight` 関数を用いた式を纏める際に, CRE 表現の切捨ての制御で用いられます. 猶, デフォルト値が `false` であれば, 切捨ては生じません.

大域変数 `ratweights` は `ratweight` 関数で設定される指定された重みのリストです. 大域変数 `ratweights` や `ratweight()` 関数でそのリストが見られます.

大域変数 `rootsconmode` は `rootscontract` 関数の挙動を定めます. 大域変数 `rootsconmode` が `false` ならば, `rootscontract` 関数は有理数次数の分母が同じ次数の冪だけを纏めます. `true` の場合, 次数の分母が割切れる冪だけを纏めます `all` の場合, 全ての有理次数の分母の LCM を取って纏めます.

式	<code>rootsconmode</code>	<code>rootscontract</code> の結果
$x^{1/2} * y^{3/2}$	<code>false</code>	$(x * y^3)^{1/2}$
$x^{1/2} * y^{1/4}$	<code>false</code>	$x^{1/2} * y^{1/4}$
$x^{1/2} * y^{1/4}$	<code>true</code>	$(x * y^{1/2})^{1/2}$
$x^{1/2} * y^{1/3}$	<code>true</code>	$x^{1/2} * y^{1/3}$
$x^{1/2} * y^{1/4}$	<code>all</code>	$(x^2 * y)^{1/4}$
$x^{1/2} * y^{1/3}$	<code>all</code>	$(x^3 * y^2)^{1/6}$

大域変数 `psexpand` は `ratexpand` と似た作用となります. `psexpand` が `true` の場合, 式全体の展開が実行されます. `psexpand` が `multi` の場合, 総次数が同じ項毎で纏めて表示されます.

6.2.5 係数体について

Maxima では多項式環の係数体として有理数 $\mathbb{Q}$ に純虚数 i を付加した体 $\mathbb{Q}[i]/\langle i^2 + 1 \rangle$ がデフォルトです.

その他の係数体を Maxima では扱う事が可能です. 先ず, p を正素数とする場合に, 多項式の係数体を $\mathbb{Z}_p[i]/\langle i^2 + 1 \rangle$ とする事も可能です.

更に, Maxima の係数体 $\mathbb{Q}[i]/\langle i^2 + 1 \rangle$ に代数的整数を追加した体も扱えます. ここで代数的整数とは整数係数の 1 変数多項式で, 最高次数の項の係数が 1 となる monic な多項式の解となる数の事です. より一般的に整数係数の 1 変数多項式の解となる数は代数的数と呼ばれます. 但し, Maxima では代数的数は扱えません.

代数的整数はその定義から多項式と密接に関連する数です. a が代数的整数である為には, a を解とする多項式が定義から必ず存在します. その a を解とする多項式の中で, 最小次数の多項式を a の最小多項式と呼びます. 代数的整数の例として, 純虚数 i や $\sqrt{2}$ 等が挙げられます. これらの最小多項式は各々 $x^2 + 1$ と $x^2 - 2$ になります.

この係数体に関連する Maxima の大域変数を以下に示しておきます.

環に関連する大域変数

変数名	初期値	概要
modulus	false	剰余 p を設定
algebraic	false	代数的整数の自動簡易化を制御

大域変数 modulus に正素数 p を設定すると, 多項式の CRE 表現への変換の際に, p の剰余で計算されます. 即ち, CRE 表現の係数体は $\mathbb{Q}[i]/\langle i^2 + 1 \rangle$ から $\mathbb{Z}_p[i]/\langle i^2 + 1 \rangle$ になります. この大域変数は mod 関数にも影響を与えます. 但し, mod 関数の第二引数を与えない場合, 大域変数 modulus の値が用いられる為です. 又, 大域変数 modulus に正素数以外の値を設定する事も可能です.

大域変数 modulus に正素数 p を設定した場合 $p/2$ よりも大きな整数全てを考えなくても良くなります. 例えば, p として 5 を採った場合, 整数の剰余は $\{0, 1, 2, 3, 4\}$ となりますが, $3 \equiv -2 \pmod{5}$, $4 \equiv -1 \pmod{5}$ となります. 実際, $3 - (-2) = 4 - (-1) = 5 \pmod{5}$ となるので, 絶対値では $\{0, 1, 2\}$ だけを考えれば良い事になります. この事を利用すれば, 計算をより簡単に行う事が可能になります.

次の大域変数 algebraic は Maxima 上で代数的整数の簡易化を行う場合, 必ず true にしなければなりません. この大域変数だけでは, 自動的に代数的整数が処理される訳ではありませんが, 代数的整数を処理する関数や他の大域変数では, この algebraic が true となっている事が前提となっているものがある為, 代数的整数を扱う場合には, true に設定するのが良いでしょう. ここで, 代数的整数に関連する大域変数としては, ratalgdenom があります. これは代数的整数を分母とする項の有理化を制御するものです. これらの変数は ratexpand 関数, ratsimp 関数等の CRE 表現の式を扱う関数に大きく影響します. その他の関数では, gcd 関数も影響を受けます.

factor 函数の様に最小多項式を与える事で, 多項式の処理を代数的整数を付加した係数体上で処理が行える函数もあります. 例えば, factor 函数の場合, 最小多項式が $x^2 - 2$ となる代数的整数 a を用いて式を分解する場合, 最小多項式に, その代数的整数を代入した形で factor 函数に引き渡します.

```
(%i1) factor(x^4-4,a^2-2);
```

```
(%o1) (x - a) (x + a) (x + 2)
```

この例では, 多項式 $x^2 - 4$ を環 $\mathbb{Q}[\sqrt{2}][x]$ 上で因子分解を行っています.

6.2.6 tellrat 函数

より徹底して代数的整数を利用する場合は tellrat 函数を用います. この tellrat 函数は最小多項式や等式に tellrat 属性を設定するもので, ratsimp 函数や ratexpand 函数で処理を行う際に大域変数 algebraic が true となっていれば, その属性が処理に反映されます.

この tellrat 函数は次の写像を Maxima に組込むものです.

`tellrat(式) : Maxima の係数環  $\rightarrow$  Maxima の係数環/ $\langle$  式  $\rangle$`

但し, tellrat 属性は CRE 表現を扱う ratexpand 函数や ratsimp 函数で処理を行う時のみに反映されます.

— tellrat 函数 —

```
tellrat( $\langle$ monic な多項式 $\rangle$ )
tellrat( $\langle$  等式  $\rangle$ )
tellrat()
```

引数として与えられる式は主変数に対して monic な多項式に限定されます. これは tellrat 函数が代数的整数を Maxima に与える事を目的としているからです. 又, 等式として与える場合に等式の左辺は代数的整数の冪乗で係数が 1 のものに限定されます. この時, 左辺の変数が主変数と看做され, 右辺を左辺に移した式を最小多項式とする代数的整数が与えられます. 従って, $a^2=c^3-2$ を tellrat 函数に与えた場合, 主変数が a で, a が Maxima に追加される代数的整数となり, 更に, その最小多項式が a^2-c^3+2 で与えられます. ここで a^2-c^3+2 を引数として与えると, 主変数は Maxima の項順序 $>_m$ から c となる為に代数的整数が c となる事に注意して下さい.

この tellrat 関数は任意個の因子を取る事が可能です.

tellrat() で,Maxima に設定された代数的整数を表現する最小多項式のリストを返します.

```
(%i22) tellrat(x^2+x+1);
```

```
(%o22) [x2 + x + 1]
```

```
(%i23) tellrat(y^3+y^2+y+1);
```

```
(%o23) [y3 + y2 + y + 1, x2 + x + 1]
```

```
(%i24) tellrat();
```

```
(%o24) [y3 + y2 + y + 1, x2 + x + 1]
```

```
(%i25)
```

tellrat で最小多項式を導入しても即座には反映されません. 先ず, 代数的整数の簡易化を行う為に大域変数 algebraic を true に設定していなければなりません. それから ratsimp 等の CRE 表現が扱える関数で処理を行う必要があります.

例として,tellrat(a^2-2,b^2=c^4) と入力した場合を示します.

```
(%i11) tellrat(a^2-2,b^2=c^4);
```

```
(%o11) [b2 - c4, a2 - 2]
```

```
(%i12) (a+2)^4,algebraic,expand;
```

```
(%o12) a4 + 8 a3 + 24 a2 + 32 a + 16
```

```
(%i13) (a+2)^4,algebraic,expand,ratsimp;
```

```
(%o13) 48 a + 68
```

```
(%i14) (b+c)^3,algebraic,expand,ratsimp;
```

```
(%o14) 3 c5 + b c4 + c3 + 3 b c2
```

```
(%i15) (b+c)^3,expand,ratsimp;
```

```
(%o15) c3 + 3 b c2 + 3 b2 c + b3
```

```
(%i16) algebraic:true;
```

```
(%o16) true
```

```
(%i17) ratexpand((b+c)^3);
              5      4      3      2
(%o17)      3 c  + b c  + c  + 3 b c
(%i18) expand((b+c)^3);
              3      2      2      3
(%o18)      c  + 3 b c  + 3 b  c + b
```

この例で示す様に, `tellrat` で設定した属性は大域変数 `algebraic` を `true` に設定した環境下で, `ratsimp` 関数や `ratexpand` 関数等の CRE 表現を内部で用いる関数で処理する時に反映されます.

猶, この例での `(a+2)^4, algebraic, expand, ratsimp` といった入力は, `ev` 関数の表記方法の一つで, `ev((a+2)^4, algebraic, expand, ratsimp)` と同値です.

次に, `tellrat(a^2-2, b^2=c^4);` で代数的整数 `b` を追加した為, `ratexpand((b+c)^3)` で, `b^2` が `c^4` で置換されている事に注意して下さい. この様に, 多変数の多項式を用いて代数的整数を入れる場合, 通常は Maxima の項順序 $>_m$ の影響を受ける為, 等式の右辺に代数的整数の項を置きます.

例えば, `a=a^2+c^3` や `a^2=c^3-a` の様にします.

ここで, `tellrat` 関数を用いて多項式を被約する際に零因子で分母の有理化を試みない様に注意しなければなりません.

例えば, `tellrat(w^3-1); algebraic:true; rat(1/(w^2-w))` は零による割算になります. このエラーは `ratalgdenom:false` に設定する事で回避出来ます.

`tellrat` 関数で設定した属性は `tellrat()` で見る事が出来ます. この場合, 引数は特に設定する必要がありません.

untellrat 関数

```
untellrat(<x> )
```

`tellrat` 関数で設定した属性は `untellrat` 関数を使えば削除出来ます. この場合, `untellrat` で代数的整数を直接指定します.

```
(%i36) tellrat(a^2-2, b^3-c^2);
              2      3      2
(%o36)      [c  - b  , a  - 2]
(%i37) tellrat();
              2      3      2
(%o37)      [c  - b  , a  - 2]
```

```
(%i38) untellrat(a);
(%o38) 
$$[c^2 - b^3]$$

(%i39) untellrat(b);
(%o39) 
$$[c^2 - b^3]$$

(%i40) untellrat(c);
(%o40) 
$$[]$$

(%i41) tellrat(a^2-2,b^3=c^2);
(%o41) 
$$[b^3 - c^2, a^2 - 2]$$

(%i42) untellrat(c);
(%o42) 
$$[b^3 - c^2, a^2 - 2]$$

(%i43) untellrat(b);
(%o43) 
$$[a^2 - 2]$$

```

次に、複数の変数に対して tellrat と untelrat を実行した例を示します。

```
(%i21) tellrat(x^2+1,y^2+1);
(%o21) 
$$[y^2 + 1, x^2 + 1]$$

(%i22) ev(rat(x^3+1+y^3+y),algebraic);
(%o22)/R/ 
$$-x + 1$$

(%i23) untellrat(y);
(%o23) 
$$[x^2 + 1]$$

(%i24) ev(rat(x^3+1+y^3+y),algebraic);
(%o24)/R/ 
$$y^3 + y - x + 1$$

```

この例では変数 x, y が $x^2 + 1 = 0$ と $y^2 + 1 = 0$ を満す代数的整数と設定しています。この様に複数の変数に対して整係数多項式を tellrat に入力する事も可能で, ev でも的確に評価されています。

次に `untellrat(y)` で y に関してのみ, `tellrat` で設定した性質 (y は $y^2 + 1 = 0$ を満す代数的整数) である事を除去しています. その後には, x に関する性質だけで評価が行われています.

猶, `tellrat` 関数で入力可能な多項式は主変数に関して `monic`(最高次項の係数が 1) なものでなければなりません. 又, 多変数の場合, `untellrat` 関数は主変数 (`maimvar`) に対して行います.

```
(%i15) tellrat(x+y+z*y1);
Minimal polynomial must be monic
-- an error. Quitting. To debug this try debugmode(true);
(%i16) tellrat(x+y+z+1);
(%o16) [z + y + x + 1]
(%i17) untellrat(y);
(%o17) [z + y + x + 1]
(%i18) untellrat(z);
(%o18) []
(%i19) tellrat(2*x+y+z+1);
(%o19) [z + y + 2 x + 1]
(%i20) untellrat(z);
(%o20) []
```

この例で示す様に, $x+y+z*y+1$ に関しては主変数が z で係数が y となる為にエラーになります. 但し, $2*x+y+z+1$ の様に主変数 z の筆頭項が `monic` でありさえすれば問題はありません. `untellrat` 関数が主変数のみに使える事も上の例から分ります.

6.2.7 Horner 則

多項式の表記で Horner 則に基く式の表記方法があります. これは X の多項式が与えられた場合, 変数 X の次数の高い順番に項を並べます. 例えば, 与えられた多項式が $a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$ とすると, $X ((\cdots (a_n X + a_{n-1}) + \cdots) + a_1) + a_0$ の様に帰納的に変数 X の積で式を纏める方法です. この Horner 則を用いると, 式の積の回数を減らす事が出来るので, 複雑な多項式の数値計算を高速化する場合には非常に有効な手段の一つです.

———— Horner 表記に変換する関数 ————

```
horner(<式>,<主変数>)
horner(<式>)
```

主変数を指定した場合には, その主変数を用いて Horner 則を適用します. 式の主変数を指定しない場合, Maxima の変数順序 $>_m$ に従って, $>_m$ で最大の変数を主変数とし, その変数に対して与式に Horner 則を適用します.

```
(%i3) expr: (x+2*y)^5, expand;
              5      4      2 3      3 2      4      5
(%o3)      32 y  + 80 x y  + 80 x y  + 40 x y  + 10 x y  + x
(%i4) horner(expr,x);
              5      4      3      2
(%o4)      32 y  + x (80 y  + x (80 y  + x (40 y  + x (10 y  + x))))
(%i5) horner(expr);
              2      3      4      5
(%o5)      y (y (y (y (32 y  + 80 x) + 80 x ) + 40 x ) + 10 x ) + x
```

猶, 主変数として関数を指定する事も可能です.

```
(%i12) neko: (sin(x)+2*y)^5, expand$
(%i13) horner(neko,sin(x));
              5      4      3
(%o13) 32 y  + sin(x) (80 y  + sin(x) (80 y
              2
              + sin(x) (40 y  + sin(x) (10 y  + sin(x)))))
```

6.2.8 多項式に関する関数

———— 多項式に関連する真理関数 ————

関数	true を返す条件
ratnumb(<式>)	<式> が有理式の場合
ratp(<式>)	<式> が拡張 CRE 表現の場合

ratnumb 関数は <式> が <有理数式> であれば true, それ以外は false を返します.

ratp 関数は $\langle \text{式} \rangle$ が CRE 表現, 或いは拡張 CRE 表現であれば true で, それ以外は false を返します.

—— 多項式の係数を取り出す関数 ——

```
coeff( $\langle \text{式} \rangle$ ,  $\langle \text{変数} \rangle$ ,  $\langle \text{次数} \rangle$ )
ratcoef( $\langle \text{式}_1 \rangle$ ,  $\langle \text{式}_2 \rangle$ ,  $\langle n \rangle$ )
ratcoef( $\langle \text{式}_1 \rangle$ ,  $\langle \text{式}_2 \rangle$ )
bothcoef( $\langle \text{式} \rangle$ ,  $\langle \text{変数} \rangle$ )
```

coeff 関数は, $\langle \text{式} \rangle$ に含まれる項 $\langle \text{変数} \rangle^{\langle \text{次数} \rangle}$ の係数を求めます. $\langle \text{次数} \rangle$ を省略すると次数は 1 が設定されます. $\langle \text{変数} \rangle$ は原子か真の部分式です. 具体的には, $x, \sin(x), a[i+1], x+y$ 等です.

猶, 真の部分式の場合, $(x+y)$ が式の中に現れていなければなりません. ここで $\langle \text{変数} \rangle^{\langle \text{次数} \rangle}$ の項を正確に求める為には, 式の展開や因子分解が必要な場合もあります. 何故なら, coeff 関数では自動的に式の展開や因子分解が実行されないからです.

```
(%i1) coeff(2*a*tan(x)+tan(x)+b=5*tan(x)+3,tan(x));
```

```
(%o1) 2 a + 1 = 5
```

```
(%i2) coeff(y+x*e**x+1,x,0);
```

```
(%o2) y + 1
```

ratcoef 関数は $\langle \text{式}_1$ に含まれる項 $\langle \text{式}_2 \rangle^{\langle n \rangle}$ の係数を返します. $\langle n \rangle$ が 1 の場合は $\langle n \rangle$ が省略出来ます. 猶, 返却値には, $\langle \text{式}_2 \rangle$ に含まれる変数を関数の変数としても含まないものです.

この様な係数が存在しない場合は零を返します. ratcoef は展開等を行って式を簡易化するので, 単純に $\langle \text{式}_2 \rangle^{\langle n \rangle}$ の係数を返す coef と異った答を返します.

例えば, ratcoeff((x+1)/y+x,x) は $(y+1)/y$ を返却しますが, coeff は 1 を返します. ratcoef($\langle \text{式}_1 \rangle$, $\langle \text{式}_2 \rangle$, 0) は, $\langle \text{式}_1 \rangle$ から $\langle \text{式}_2 \rangle$ を含まない項の和を返します. その為, $\langle \text{式}_2 \rangle$ が負の冪の項に含まれれば, ratcoef は使ってはいけません. $\langle \text{式}_1 \rangle$ が有理的に簡易化されていれば, 係数は予期した様に現れないかもしれません.

bothcoef 関数は二成分のリストを返し, このリストの第一成分が $\langle \text{式} \rangle$ 中の $\langle \text{変数} \rangle$ の係数 (式が CRE 表現であれば ratcoef, それ以外は coeff 関数で見つけたもの) となります. 第二成分が $\langle \text{式} \rangle$ の残りとなります. 即ち, $[a,b]$ が返却値であれば, $\langle \text{式} \rangle = a * \langle \text{変数} \rangle + b$ となります.

項数や次数を返す関数

```
nterms(< 式 >)
powers(< 式 >, < 変数 >)
```

nterms 関数は < 式 > を展開した時の項数を返却します. この関数は多項式以外の関数を含む通常の式でも利用可能ですが, 関数は引数の形式を問わず一つで数えられます.

```
(%i26) nterms((x+1)^2);
(%o26)                                     3
(%i27) nterms(sin(x+1)^2);
(%o27)                                     1
(%i28) nterms((sin(x+1)+1)^3);
(%o28)                                     4
(%i29) nterms((sin((x+1)^10)+1)^3);
(%o29)                                     4
```

この例で示す様に, sin 関数の引数がどのような式であっても, 一つで数えられている事に注意して下さい.

次の powers 関数は < 式 > に現われる < 変数 > の次数リストを返します. この関数を利用する為には, 予め, `load(powers);` で関数の読込を行う必要があります.

多項式の項の次数を返す関数

```
hipow(< 多項式 >, < 変数 >)
lopow(< 多項式 >, < 変数 >)
```

hipow 関数は < 多項式 > に含まれる < 変数 > の項の最高次数を返します. 猶, hipow では式の展開を自分で実行しない為, 予め式を展開しておく必要があります. 以下の例では $(x+1)^4$ を展開せずに hipow を用いた結果と expand で展開した式に対して hipow を用いた結果を示しています.

```
(%i5) hipow((x+1)^4,x);
(%o5)                                     1
(%i6) hipow(expand((x+1)^4),x);
(%o6)                                     4
```

lopow 関数は < 多項式 > の部分式 < 変数 > の次数で, 明示的に現われものの中で最も低い次数を返します.

多項式の変数に関する函数

```

ratvars(⟨変数1⟩, ..., ⟨変数n⟩)
ratweight(⟨変数1⟩, ⟨重み1⟩, ..., ⟨変数n⟩, ⟨重みn⟩)
showratvars(⟨式⟩)
printlistvar()

```

ratvars 函数は与えられた変数リストに含まれる変数に沿った順序を Maxima に入れる函数です. 変数リストは n 個の変数引数で構成し, ratvars 函数を実行した後, リスト中の一番右側の ⟨変数_n⟩ が有理式にあれば, その変数を有理式の主変数になります. 又, 他の変数の順序はリストの右から左への順番に従います. ある変数が ratvars リストから抜けていれば, その変数は一番左側の ⟨変数₁⟩ よりも低い順序が付けられます. ratvars 函数の引数は変数, 或いは $\sin(x)$ の様な非有理的函数の何れかでなければなりません. 猶, 大域変数 ratvars は, この函数に与えられた引数のリストとなります.

```

(%i26) ratvars(x,y,z);
(%o26)                                     [x, y, z]
(%i27) rat(x+y+z);
(%o27)/R/                                z + y + x
(%i28) rat(a+x+y+z);
(%o28)/R/                                z + y + x + a
(%i29) ratvars(z,y,x);
(%o29)                                     [z, y, x]
(%i30) rat(a+x+y+z);
(%o30)/R/                                x + y + z + a

```

猶, ratvars 函数で指定した変数リストは printlistvar 函数を用いて表示可能です.

ratweight 函数は, ⟨重み_i⟩を ⟨変数_i⟩に割当てます. その重みが大域変数 ratwtlvl の値に従って項が 0 で置き換えられます. 項の重みは項の中の変数の冪を重みに掛けた積の和となります. 例えば, $3 \cdot v_1^2 \cdot v_2$ の重みは $2 \cdot w_1 + w_2$ となります. この切捨ては, 式の CRE 表現の積や冪乗を実行する時のみに生じます.

猶, ratfac 函数と ratweight 函数の手法は互換性がないので両方同時に使えません.

showratvars 函数は ⟨式⟩の大域変数 ratvars のリストを返します.

```

(%i30) exp:x^2+y^2+z^3;
(%o30)                                     3      2      2
                                     z  + y  + x

```

```
(%i31) showratvars(exp);
```

```
(%o31) [x, y, z]
```

printlistvar 関数は内部変数 varlist に束縛された値を返す関数です。この関数は引数が必要としません。

—— 多項式を纏める関数 ——

```
factcomb(<式>)
fasttimes(<多項式1>, <多項式2>)
rootscontract(<式>)
```

factcomb 関数は <式> 中に現われる階乗の係数を階乗それ自体に置換して纏めます。即ち、 $(n+1)*n!$ を $(n+1)!$ にする事です。ここで、大域変数 sumsplifactor が false に設定されていれば、minfactorial 関数が factcomb の後に適用されます。

fasttimes 関数は多項式の積に対する特殊なアルゴリズムを用いて、<多項式₁> と <多項式₂> の積を計算します。これらの多項式は、多変数で各次数に対して係数が 0 でない項が多く、両者共に殆ど同じ大きさされなければ効果があまり出ません。

n と m を多項式の次数とすると、古典的な積では $n*m$ のオーダーで計算を行います。fasttimes 関数を用いると $\max(n, m)^{1.585}$ のオーダーになります。

rootscontract 関数は有理数次数の冪同士の積を大域変数 rootsmode の値に従って纏めます。例えば、rootsmode が true の場合、 $x^{(1/2)}*y^{(3/2)}$ の様な根同士の積を $\text{sqrt}(x*y^3)$ に纏めます。

大域変数 radexpand が true で大域変数 domain が real であれば、rootscontract 関数は abs を sqrt に変換します。即ち、 $\text{abs}(x)*\text{sqrt}(y)$ を $\text{sqrt}(x^2*y)$ に変換します。

rootscontract 関数は logcontract 関数と似た手法で ratsimp 関数を用います。

—— 有理式に関連する関数 ——

```
combine(<式>)
denom(<有理式>)
num(<有理式>)
ratdenom(<有理式>)
ratnumer(<有理式>)
ratdiff(<有理式>, <変数>)
xthru(<式>)
```

combine 関数は <式> に含まれる和の部分式を同じ分母で纏めて一つの項にします。

猶,combine 関数の制御で大域変数 combineflag が用いられています. この大域変数 combineflag の初期値は true で, 有理式の分母が整数の場合, 共通の分母で式を纏めませんが,false の場合はそのままにします.

```
(%i25) combineflag:true$
```

```
(%i26) combine(1/4+(y+1)^4/2);
```

```
(%o26)

$$\frac{2 (y + 1)^4 + 1}{4}$$

```

```
(%i27) combineflag:false$
```

```
(%i28) combine(1/4+(y+1)^4/2);
```

```
(%o28)

$$\frac{(y + 1)^4}{2} + \frac{1}{4}$$

```

denom 関数は〈有理式〉の分母 (DENOMinator) を返します. 猶, 有理式が通常が多項式であれば,1 を返します.

```
(%i40) denom((x^2+1)/(y^2+1)/2);
```

```
(%o40)

$$2 (y^2 + 1)$$

```

```
(%i41) denom(x^2+1);
```

```
(%o41)

$$1$$

```

```
(%i42) denom(1/2*x^2+1/2);
```

```
(%o42)

$$1$$

```

```
(%i43) denom((x^2+1)/2);
```

```
(%o43)

$$2$$

```

num 関数は有理式の分子 (NUMerator) を返します.

ratdenom 関数は〈有理式〉の分母を計算します. 〈有理式〉が一般形式で, 結果も一般形式のものが必要ならば, denom 関数を用いましょう.

ratnumer 関数は〈有理式〉の分子を取り出します. 一般形式の〈有理式〉に対して,CRE 表現の結果が不要であれば, num 関数を使いましょう.

ratdiff 関数は〈有理式〉の微分を〈変数〉で行います. 有理式に対しては diff 関数よりも処理が速く, 計算結果は CRE 表現になります. 猶, ratdiff 関数は因子分解された

CRE 表現には使ってはいけません. 因子分解された式では通常の diff 函数を使いましょう.

xthru 函数は有理式を一つの分母で纏める函数です. この時に, 分母の展開を実行せずに多項式を纏めます.

```
(%i34) xthru(1/x+1/y+1/(x*y));
```

```
(%o34)          y + x + 1
          -----
                x y
```

```
(%i35) xthru(1/(x-1)^2+1/(y+1)^3-(y+1)^2/(x*(x-1)));
```

```
(%o35)          3          2          5
          x ((y + 1)  + (x - 1) ) - (x - 1) (y + 1)
          -----
                2          3
          (x - 1) x (y + 1)
```

終結式に関連する函数

resultant(⟨多項式₁⟩, ⟨多項式₂⟩, ⟨変数⟩)

bezout(⟨多項式₁⟩, ⟨多項式₂⟩, ⟨変数⟩)

eliminate([⟨方程式₁⟩, ⟨方程式₂⟩, ..., ⟨方程式_n⟩], [⟨変数₁⟩, ⟨変数₂⟩, ..., ⟨変数_k⟩])

poly_discliminant(⟨多項式⟩, ⟨変数⟩)

bezout 函数は, ⟨多項式₁⟩ と ⟨多項式₂⟩ に対して, ⟨変数⟩ を主変数とした場合のある係数行列を返します. この係数行列の行列式を取ると終結式に等しくなるものです. この行列 bezout(f,g,x) の determinant が多項式 f と g の終結式になります. 従って, bezout 函数と determinant 函数と組合せれば resultant 函数の代替になります. resultant 函数は二つの多項式 ⟨多項式₁⟩ と ⟨多項式₂⟩ の終結式を計算し, 指定した ⟨変数⟩ を消去します.

ここで, 多項式 f と g の解を各々 α_i, β_j とすると, 多項式 f と g の終結式 $\text{res}(f, g)$ は以下の式と等しくなる事が知られています.

$$\text{res}(f, g) =_{\text{def}} a_m^n b_n^m \prod_{1 \leq i \leq m, 1 \leq j \leq n} (\alpha_i - \beta_j)$$

その為, 終結式は ⟨多項式₁⟩ と ⟨多項式₂⟩ が共通の定数の因子を持つ場合に限って零になる多項式となる事が判ります.

終結式の計算方法は、 $\langle \text{多項式}_1 \rangle$ と $\langle \text{多項式}_2 \rangle$ を $\langle \text{変数} \rangle$ の多項式と看做した場合の係数から構成される行列の行列式から計算出来ます。行列の大きさは、 $\langle \text{多項式}_1 \rangle$ と $\langle \text{多項式}_p \rangle$ の次数を各々 m, n とすると、 $m+n$ 次の正方行列となりますが、bezout では行列操作によって、それよりも小さな行列が得られる場合には、その行列を表示します。具体的には、多項式 f と g を次のものとします。

$$f = \sum_{i=0}^m a_i x^i \quad g = \sum_{i=0}^n b_i x^i$$

すると、

$$\text{resultant}(f, g, x) =_{\text{def}} \det \begin{pmatrix} a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 \\ b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 \end{pmatrix}$$

で終結式は計算出来ます。猶、 $\langle \text{多項式}_1 \rangle, \langle \text{多項式}_2 \rangle$ が因子分解可能であれば resultant 関数を呼び出す前に factor 関数を呼出すと良いでしょう。

eliminate 関数は与えられた方程式、或いは零と等しいと仮定した式から続けて終結式を取る事で、指定された変数の消去を行います。eliminate 関数に引渡した k 個の $\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle$ を消去した $n-k$ 個の式のリストを返します。最初の $\langle \text{変数}_1 \rangle$ は消去されて $n-1$ 個の式を生成し、 $\langle \text{変数}_2 \rangle$ 以降も同様です。

$k=n$ の場合、結果リストは k 個の $\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle$ を持たない一つの式となります。そして最後の変数に対応する終結式を解く為に solve 関数を呼出します。

```
(%i1) exp1:2*x^2+y*x+z$
(%i2) exp2:3*x+5*y-z-1$
(%i3) exp3:z^2+x-y^2+5$
(%i4) eliminate([exp3,exp2,exp1],[y,z]);

      8      7      6      5
(%o4) [7425 x  - 1170 x  + 1299 x  + 12076 x
      4      3      2
      + 22887 x  - 5154 x  - 1291 x
      + 7688 x + 15376]
```

```
(%i5) eliminate([x+y=2,2*x+3*y-5=0],[x,y]);
(%o5) [1]
(%i6) eliminate([x+y=2,2*x+3*y-5=0],[x]);
(%o6) [y - 1]
(%i7) eliminate([x+y=2,2*x+3*y+5=0],[x]);
(%o7) [y + 9]
(%i8) eliminate([x+y=2,2*x+3*y+5=0],[x,y]);
(%o8) [- 9]
```

—— 終結式のアルゴリズムを指定する大域変数 ——

変数名	初期値	可能な値
resultant	subres	[subres,mod,red]

大域変数 resultant は同名の resultant 関数による終結式の計算で用いるアルゴリズムを設定します。指定可能なアルゴリズムを以下に示しておきます。

デフォルト	subres
モジュラー終結式アルゴリズム	mod
縮約 prs	red

殆どの問題では subres が最適です。単変数の大きな次数や 2 変数問題では mod がより良いでしょう。

poly_discriminant 関数は与えられた多項式を指定した変数を主変数として、判別式を計算する関数です。

```
(%i146) poly_discriminant(x^2+1,x);
(%o146) - 4
(%i147) poly_discriminant(x^2+2*x+1,x);
(%o147) 0
(%i148) poly_discriminant(x^2+2*x*y^2+y+1,x);
(%o148) 4
4 y - 4 y - 4
(%i149) poly_discriminant(x^2+2*x*y^2+y+1,y);
(%o149) 3
- 8 x - 8 x + 1
```

共通因子を求める関数

```
gcd(⟨式1⟩,⟨式2⟩,⟨変数1⟩,...)
gcde(⟨多項式1⟩,⟨多項式2⟩,⟨変数⟩)
gcdex(⟨多項式1⟩,⟨多項式2⟩)
gcfactor(⟨Gauß整数⟩)
gfactor(⟨多項式⟩)
gfactorsum(⟨多項式⟩)
ezgcd(⟨多項式1⟩,⟨多項式2⟩,...)
content(⟨多項式⟩,⟨変数1⟩,...,⟨変数n⟩)
```

gcd 関数は与えられた多項式の最大公約因子を計算します。この gcd 関数は多くの関数、例えば, ratsimp 関数や factor 関数等でも利用されています。

gcd 関数に直接影響を及ぼす大域変数として、同名の大域変数 gcd があります。この大域変数 gcd は gcd 関数で用いるアルゴリズムを決定する変数です。

大域変数 gcd に設定可能な値

値	概要
subres	副終結式を利用 (デフォルト値)
ez	ezgcd 関数を利用
eez	eez gcd を利用
red	被約
spmod	剰余
false	gcd 関数は常に 1 を返却

代数的整数を扱う場合、例えば, gcd($x^2-2*\sqrt{2}*x+2, x-\sqrt{2}$) の GCD を計算する為には、大域変数の algebraic が true であり、大域変数 gcd が ez と false 以外の値でなければなりません。

同次多項式に対しては, gcd:subres を用いる事を推奨します。

gcdex 関数は 3 個の多項式を成分とするリスト [a,b,c] を返します。多項式 c が引数の ⟨多項式₁⟩ と ⟨多項式₂⟩ の最大公約因子で, a,b は共に $c = a * \langle \text{多項式}_1 \rangle + b * \langle \text{多項式}_2 \rangle$ を満す多項式となります。この関数が用いているアルゴリズムは Euclid の互除法に基づくものです。

猶、多項式が単変数の場合は ⟨変数⟩ を指定する必要はありませんが、多変数の場合、多項式を ⟨変数⟩ で指定した単変数の多項式と看做して GCD を計算します。

何故、多変数多項式が相手の場合に、変数の指定を行う必要があるかと言えば、多変数

多項式の場合は、二つの多項式の最大公約因子が存在するとは限らないからです。数学的には、最大公約因子は二つの多項式が生成するイデアルの生成元となります。通常、多項式の係数が実数や複素数で、考えている多項式が単変数のみ、即ち、多項式環 $k[x]$ であれば、任意のイデアルは単項イデアル、即ち、一つの多項式だけで生成されるので、この場合は最大公約因子が必ず存在します。その為、多変数の場合には二つの多項式の主変数として変数の一つを選択する必要があります。但し、その選択が妥当でなければ、gcdex は適当な答を返すだけです。

```
(%i16) gcdex(x^2+1,x^3+4);
```

```
(%o16)/R/
          2
      x  + 4 x - 1 x + 4
[- -----, -----, 1]
      17          17
```

```
(%i18) gcdex(x*(y+1),y^2-1,x);
```

```
(%o18)/R/
          1
[0, -----, 1]
          2
        y  - 1
```

```
(%i19) gcdex(x*(y+1),y^2-1,y);
```

```
(%o19)/R/
[1, 0, x y + x]
```

ここで、最後の多変数の例で、gcdex(x*(y+1),y^2-1,x) の結果で、GCD として 1 を返している事に注意して下さい。この場合、多項式環 $K(y)[x]$ で処理を行っているので、共通の因子として期待される $y+1$ にはなりません。ここで、 $K(y)[x]$ は x を主変数とした x と y の多項式環、つまり、 x の多項式で、その係数が体 K 上の y の多項式となるものとして、 x と y の多項式環 $K[x, y]$ を見直したものです。一般的に可換環 K が UFD (Unique Factorized Domain: 一意分解整域) であれば、 $K[x]$ も UFD になる事が知られています。その為、Eculid の互除法が利用可能になるので、gcdex は必ず結果を返します。

`gcdex(x*(y+1),y^2-1,y);` とすれば、多項式環 $K(x)[y]$ の話になるので 1 ではなく $xy + x$ になります。但し、この返却値が良いものとは言い難いものがあります。

gcfactor 関数は Gauss 整数上で $\langle \text{Gauss 整数} \rangle$ の因子分解を行います。猶、Gauss 整数とは、複素数 $a + bi$ で、 a と b が整数になります。因子は a と b を非負とする事で正規化されています。


```
(%i56) gcfactor(5*%i+1);
(%o56) (1 + %i) (3 + 2 %i)
(%i57) gcfactor(2);
2
```

gfactor 関数は Gauss 整数上で〈多項式〉の因子分解を行ないます。これは factor(exp,a^2+1) と同様の結果を返します。

```
(%i3) gfactor(x^4-1);
(%o3) (x - 1) (x + 1) (x - %i) (x + %i)
(%i4) factor(x^4-1,a^2+1);
(%o4) (x - 1) (x + 1) (x - a) (x + a)
(%i5)
```

この例で, factor を用いたものでは方程式 $x^2 + 1 = 0$ の解となる代数的整数 $a(= i)$ を用いて $x^4 - 1$ を因子分解しています。

gfactorsum 関数は factorsum に似ていますが, factor の代りに gfactor が適用されます。

```
(%i58) gfactorsum(x^2+1);
(%o58) (x - %i) (x + %i)
(%i59) factor(x^2+1);
2
(%o59) x + 1
```

ezgcd 関数は, 第一成分が全ての多項式の GCD, 残りの元が GCD で割った値を成分に持つリストを返します。この ezgcd 関数では ezgcd アルゴリズムが常用されています。content 関数は二成分のリストを返し, このリストの第一成分が, 〈変数₁〉を〈多項式〉の主変数とした場合の各係数の最大公約因子, 第二成分を第一成分で多項式を割った monic な多項式となります。

```
(%i43) content(2*x*y+4*x^2*y^2,y);
2
(%o43) [2 x, 2 x y + y]
```

因子分解を行う関数

```
factor(< 式 >)
factor(< 式 >, < p >)
factorsum(< 式 >)
sqfr (< 式 >)
factorout(< 式 >, < 変数1 >, < 変数2 >, ... )
nthroot(< 多項式 >, < n >)
polydecomp(< 多項式 >, < 変数 >)
```

factor 関数は < 式 > を整数上で既約因子に分解します。factor(< 式 >, < p >) の場合、最小多項式が < p > となる代数的数 α を付加した体 $\mathbb{Q}[\alpha]$ 上で式の因子分解を行います。猶、factor 関数には動作に影響を与える大域変数が存在します。この大域変数に関しては、factor に影響を与える大域変数を参照して下さい。

factorsum 関数はグループ単位で < 式 > の因子分解を試みます。このグループの項はそれらの和が因子分解可能なものです。expand((x+y)^2+(z+w)^2) の結果は復元可能ですが、expand((x+1)^2+(x+y)^2) の結果は共通項が存在する為に復元出来ません。sqfr 関数は factor 関数と似ていますが、多項式因子は無平方 (square-free) となります。ここで多項式因子 f が無平方であるとは、定数と異なる多項式 g で、 g^2 が f を割切る様なものが存在しない場合です。特に1変数多項式の場合、 f と $\frac{d}{dx}f$ が共通の零点を持ちません。そこで、多項式 a の無平方因子分解は、多項式 a の分解 $\prod_{i=1}^n a_i^{i_i}$ で、その各因子 a_i は無平方であり、 $i \neq j$ であれば $\gcd(a_i, a_j)=1$ を満すものです。Maxima の sqfr は与えられた式に対して、この無平方因子分解を計算します。

では、無平方因子分解 sqfr と因子分解 factor との違いを $4x^4 + 4x^3 - 3x^2 - 4x - 1$ で比較して確認しましょう。

```
(%i44) sqfr(4*x^4+4*x^3-3*x^2-4*x-1);
                                2    2
(%o44)          (2 x + 1) (x  - 1)
(%i45) factor(4*x^4+4*x^3-3*x^2-4*x-1);
                                2
(%o45)          (x - 1) (x + 1) (2 x + 1)
```

この例で示す様に、factor 関数は式を徹底して分解しているのに対し、sqfr 関数は程々で止めている事です。

この様に、多項式を無平方因子分解は通常因子分解程の手間をかけません。更に、有理多項式の積分計算では、無平方な因子に分母を分解して、各因子を分母に持つ式に

変形して積分を行うアルゴリズムもあります. この無平方分解は幅広く利用されています.

factorout 関数は $\langle \text{式} \rangle$ を $f(\langle \text{変数}_1 \rangle, \langle \text{変数}_2 \rangle, \dots) * g$ の形式の項の和に書換えます. ここで, g は factorout の引数の各変数を含まない式の積で, f は因子分解されたものになります.

nthroot 関数は与えられた整数係数の $\langle \text{多項式} \rangle$ が, ある整数係数の多項式を $\langle \text{正整数} \rangle$ で冪乗したものであれば, その多項式を返します. もし, その様な多項式が存在しなければ, エラーメッセージが表示されます.

この関数は factor 関数や sqfr 関数よりも処理が遥かに速いものです.

```
(%i22) nthroot(x^2+2*x+1,2);
(%o22)
x + 1
(%i23) nthroot(x^3+3*x^2+3*x+1,2);
Not an nth power
-- an error. Quitting. To debug this try debugmode(true);
(%i24) nthroot(1-3*x+3*x^2-x^3,3);
(%o24)
1 - x
```

polydecomp 関数は与えられた多項式を指定された変数の 1 変数多項式として多項式の分解を行い, リストで返す関数です. ここでの分解は因子分解によるものとは異ったものです. polydecomp でリスト $[p_1(x), \dots, p_n(x)]$ が与えられた場合, 本来の多項式 $P(x)$ との関係は λ 表現を用いて以下の様に記述されます.

$$\lambda x.p_1(x)(\lambda x.p_2(x)(\dots(\lambda x.p_n(x))\dots))$$

```
(%i41) polydecomp(x^3+3*x^2+3*x+1,x);
3
(%o41)
[x , x + 1]
```

因子分解に関連する大域変数

変数名	初期値	概要
dontfactor	[]	factor 関数で因子分解しない式のリスト
facexpand	true	factor 関数で返される因子を制御
factorflag	false	式に含まれる整数の因数分解を制御
berlefact	true	因子分解のアルゴリズムを制御
intfaclim	1000	factor で用いる最大の約数の設定
newfac	false	因子分解ルーチンの選択
savefactors	false	式の因子の保存を制御
sumsplitfact	true	factcomb の挙動を制御

大域変数 dontfactor に factor 関数で因子分解しない式を登録します。猶, ratvars 関数で入れた変数の順序に対し、大域変数 dontfactor に割当てられたリストに含まれる変数よりも小さな変数を持つ式の因子分解は実行されません。

大域変数 facexpand は factor 関数で返された既約因子が展開された形式 (デフォルト) か、再帰的 (通常の CRE) 形式であるかを制御します。

大域変数 factorflag が false であれば有理式に含まれる整数の因数分解を抑制します。

大域変数 berlefact が false であれば、factor 関数で Kronecher の因子分解アルゴリズムが利用され、それ以外ではデフォルトの Berlekamp アルゴリズムが使われます。

大域変数 intfaclim は大きな整数の因子分解を行う時に試す最大の約数を指定します。false を指定した場合、即ち、利用者が factor 関数を明示的に呼び出す場合や、整数が fixnum、つまり、1 機械語長に適合する場合、整数の完全な因数分解が試みられます。intfaclim の設定は factor の内部呼び出しで用いられます。

大域変数 intfaclim は大きな整数の因数分解に長時間を費すのを防ぐ為に再設定しても構いません。

大域変数 newfac が true であれば、factor は新しい因子分解ルーチンを用います。

大域変数 savefactors が true であれば、幾つかの同じ因子を含む後の式の展開の処理速度向上の為、式の各因子がある関数で保存されます。

大域変数 sumsplitfact が true の場合、factcomb 関数は minfactorial 関数の呼び出しを行います。

—— 剰余を計算する関数 ——

```
divide(⟨多項式1⟩,⟨多項式2⟩,⟨変数1⟩,...,⟨変数n⟩)
quotient(⟨多項式1⟩,⟨多項式2⟩,⟨変数1⟩,...)
remainder(⟨多項式1⟩,⟨多項式2⟩,⟨変数1⟩,...)
polymod(⟨多項式⟩)
polymod(⟨多項式⟩,⟨整数⟩)
```

divide 関数は $\langle \text{多項式}_2 \rangle$ による $\langle \text{多項式}_1 \rangle$ の商と剰余を計算します. 各多項式は $\langle \text{変数}_n \rangle$ を主変数とし, その他の変数は ratvars 関数に現れるものとしします. 結果はリストで返却され, 第一成分が商, 第二成分が剰余となります.

```
(%i1) divide(x+y,x-y,x);
(%o1) [1, 2 y]
(%i2) divide(x+y,x-y);
(%o2) [- 1, 2 x]
```

quotient 関数は $\langle \text{多項式}_2 \rangle$ による $\langle \text{多項式}_1 \rangle$ の割り算の商を計算します.

remainder 関数は $\langle \text{多項式}_2 \rangle$ による $\langle \text{多項式}_1 \rangle$ の剰余を計算します.

polymod 関数は $\langle \text{多項式} \rangle$ の第二引数の整数や, 大域変数 modulus で指定した値に対する剰余を計算します. ここで大域変数 modulus の初期値は false である為, この大域変数 modulus に整数値が割当てられる迄は, polymod 関数の引数は二つ必要です. 大域変数 modulus が初期値の false のままで一つの引数で計算させようとするとエラーになります.

```
(%i14) polymod(expand((x+2)^3),3);
3
(%o14) x - 1
(%i15) modulus:3;
3
(%i16) polymod(expand((x+2)^3));
3
(%o16) x - 1
(%i17) modulus:false$
(%i18) polymod(expand((x+2)^3));
Maxima encountered a Lisp error:
```

```
MINUSP: NIL is not a real number
```

```
Automatically continuing.
```

```
To reenble the Lisp debugger set *debugger-hook* to nil.
```

```
(%i19)
```

CRE 表現の簡易化に関連する関数

```
ratexpand(< 式 > )
fullratsimp(< 式 >, < 変数1 >, ..., < 変数n > )
fullratsimp(< 式 > n )
ratsimp(< 式 > )
ratsimp(< 式 >, < 変数1 >, ..., < 変数n > )
partfrac(< 式 >, < 変数 > )
```

ratexpand 関数は和の積や指数の和を掛け, 共通の分子で因子を纏め, 分子と分母の共通約数を通分し, 分子を分母によって割られた項へと分割して < 式 > の展開を行います. これは < 式 > を CRE 表現に変換し, それからを一般形式に戻しています.

その為, ratexpand に影響を与える大域変数には ratexpand, ratdenomdivide や keepfloat といった CRE 表現への変換に影響を与える大域変数があります.

fullratsimp 関数は < 式 > に非有理式が含まれていれば, 普通, 簡易化した結果を返す際に, やや非力な非有理的 (一般的) 簡易化に続いて ratsimp を呼出します. 時には, その様な呼出しが一回以上必要であるかもしれません. fullratsimp は, この操作を簡易にしたものです.

fullratsimp 関数は非有理的簡易化に続けて ratsimp を式に変化が生じなくなる迄適用します.

例えば, 式 $\text{exp}:(x^{(a/2)+1})^2 \cdot (x^{(a/2)-1})^2 / (x^a - 1)$ に対し, ratsimp(exp) によって

$(x^{(2 \cdot a)} - 2 \cdot x^{a+1}) / (x^a - 1)$ が得られ, fullratsimp(exp) を実行すれば $x^a - 1$ が得られます.

ratsimp 関数は, 非有理的関数に対して, 有理的に < 式 > とその部分式の全てを引数も含めて, ratexpand の様に簡易化します. 結果は二つの多項式の商として, 再帰的な形式で返されます. 即ち, 主変数の係数は他の変数の多項式となっており, その係数もまた変数の順序に沿って, 主変数の次に順序の高い変数の多項式の係数と纏められています. 変数は ratexpand の様に非有理的関数 (例えば, $\sin x^2 + 1$ を含みますが, ratsimp

で非有理的函数に対する引数は有理的に簡易化されます. ratsimp は ratexpand に影響を与える幾つかの大域変数の影響を受ける事に注意して下さい.

猶,ratsimp(< 式>,< 変数₁>,...,< 変数_n>) で, ratvars に変数の < 変数₁>,... を設定した場合と同様に, この変数の並びの順序で有理的簡易化を行います.

partfrac 函数は, 与えられた有理式を有理式の和に分解する函数で,ratsimp 函数の逆操作を行う様な函数です. 第一引数が分解すべき式で, 第二引数に式の主変数を指定します.

```
(%i135) neko:(y/(x+1)+(y^2+x)/(y*(x+1))+1/(x+1)^2);
```

```
(%o135)
```

$$\frac{y^2 + x}{(x + 1)y} + \frac{y}{x + 1} + \frac{1}{(x + 1)^2}$$

```
(%i136) ratsimp(neko);
```

```
(%o136)
```

$$\frac{(2x^2 + 2)y^2 + y^2 + x^2 + x}{(x^2 + 2x + 1)y}$$

```
(%i137) partfrac(neko,x);
```

```
(%o137)
```

$$\frac{2y^2 - 1}{(x + 1)y} + \frac{1}{y} + \frac{1}{(x + 1)^2}$$

```
(%i138) partfrac(neko,y);
```

```
(%o138)
```

$$\frac{(2x + 2)y + 1}{x^2 + 2x + 1} + \frac{x}{(x + 1)y}$$

trunc 函数

`trunc(⟨式⟩)`

この事を実際に確認して見ましょう.

ところが, CRE 形式や Taylor 級数に対しては動作が異なります.

```
(%i1) t1:taylor(sin(x),x,0,5);
```

$$\frac{x^3}{6} - \frac{x^5}{120} + \dots$$

```
(%o1)/T/
```

```
(%i2) r1:taytorat(t1);
```



```

              5      3
            x  - 20 x  + 120 x
(%o2)/R/
            -----
              120

(%i3) tr1:trunc(t1);

              3      5
             x      x
(%o3)  x  - -- + --- + . . .
              6      120

(%i4) tr2:trunc(r1);

              5      3
            x  - 20 x  + 120 x
(%o4)  -----
              120

(%i5) :lisp $tr1;
((MPLUS SIMP TRUNC) $X
 ((MTIMES SIMP) ((RAT SIMP) -1 6) ((MEXPT SIMP RATSIMP) $X 3))
 ((MTIMES SIMP) ((RAT SIMP) 1 120) ((MEXPT SIMP RATSIMP) $X 5)))

(%i5) :lisp $tr2
((MTIMES SIMP) ((RAT SIMP) 1 120)
 ((MPLUS SIMP) ((MTIMES SIMP RATSIMP) 120 $X)
 ((MTIMES SIMP) -20 ((MEXPT SIMP RATSIMP) $X 3)) ((MEXPT SIMP RATSIMP) $X 5)))

```

この例で示す様に, 内部書式では型の変更が生じています.

6.3 級数の扱い

6.3.1 Maxima に於ける級数の表現

Maxima では級数の扱いは二通りあります. 一つは `sum` 関数を用いた形式的な級数の表記方法です. この場合, 基本的のは多項式と然程の違いはありません. もう一つの方法は, Taylor 級数を用いる方法です.

先ず, 形式的級数を生成する関数として `powerseries` 関数があります. この `powerseries` 関数の構文を挙げておきます.

powerseries 関数の構文

`powerseries(< 関数 >, < 変数 >, < 級数展開を行う点 >)`

```
(%i10) neko:powerseries(sin(2*x)*exp(x),x,0);
      inf      inf
      ====      i8  ====      i8  2 i8 + 1  2 i8 + 1
      \      x      \      (- 1)  2      x
(%o10) ( >      ---) >      -----
      /      i8! /      (2 i8 + 1)!
      ====      ====
      i8 = 0      i8 = 0

(%i11) neko:powerseries(sin(2*log(x)),x,0);
(%o11) Unable to expand
```

`powerseries` 関数は形式的冪級数を求める関数です. 出力される関数は有理式の無限和を `sum` 関数を用いて表現したものになります. 従って, 形式的な級数です. `powerseries` 関数は三角関数や双曲関数の級数展開した情報を持っています. これらの情報を用いて与えられた式の冪級数展開を行いますが, 展開が容易に出来ない場合には, `Unable to expand` を返します.

さて, Maxima の Taylor 級数は通常の多項式表現を拡張したものではなく, より高速な処理の行える CRE 表現を拡張したものとして表現されています.

先ず, Maxima の Taylor 級数は無限の長さを持つ級数として表現されます. そして, データ型が Taylor 級数である事を示す為に, 表示の際に `/T/` が先頭に付けられます. 実際, `sin` 関数を原点 0 の回りで展開した例を以下に示しましょう.

```
(%i5) taylor(sin(x),x,0,10);
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} + \dots$$

```
(%o5)/T/
```

ところで、この式の内部表現は以下の様になっています。

```
(%i7) :lisp $t1
((MRAT SIMP (((%SIN SIMP) $X) $X)
  (cos(1)15731 sin(1)15730 cos(1)15729 sin(1)15728 cos(x0)15727 sin(x0)15725
    X015723 sin(x)15721 X15722)
  (($X ((10 . 1)) 0 NIL sin(1)15730 . 2)) TRUNC)
PS (sin(1)15730 . 2) ((10 . 1)) ((1 . 1) 1 . 1) ((3 . 1) -1 . 6)
((5 . 1) 1 . 120) ((7 . 1) -1 . 5040) ((9 . 1) 1 . 362880))
```

この内部表現から Taylor 級数は CRE 表現を拡張した書式である事が明瞭になります。従って、Taylor 級数から通常の CRE 表現への変換も簡単に行えます。Taylor 級数を CRE 表現に `taytorat` 関数を用いて変換した例を以下に示しておきましょう。

```
(%i7) r1:taytorat(t1);
```

$$\frac{x^9 - 72x^7 + 3024x^5 - 60480x^3 + 362880x}{362880}$$

```
(%o7)/R/
```

```
(%i8) :lisp $r1
((MRAT SIMP (((%SIN SIMP) $X) $X)
  (cos(1)15731 sin(1)15730 cos(1)15729 sin(1)15728 cos(x0)15727 sin(x0)15725
    X015723 sin(x)15721 X15722)
  (sin(1)15730 9 1 7 -72 5 3024 3 -60480 1 362880) . 362880)
```

Maxima で Taylor 級数を生成する事が可能な関数は `taylor` 関数のみです。この `taylor` 関数の構文を以下に纏めておきます。

— Taylor 展開を行う関数 —

```

taylor(<式>, <変数>, <座標>, <正整数値>)
taylor(<式>, [<変数>, <座標>, <正整数値>, asymp])
taylor(<式>, [<変数1>, ..., <変数n>], <座標>, <正整数値>)
taylor(<式>, [<変数1>, ..., <変数n>], [<座標1>, ..., <座標n>], <正整数値>)
taylor(<式>, [<変数1>, ..., <変数n>], [<座標1>, ..., <座標n>], [<正整数値1>, ..., <正整数値n>])
taylor(<式>, [<変数1>, <座標1>, <正整数値1>], ..., [<変数n>, <座標n>, <正整数値n>])

```

基本的に, Taylor 展開では, 関数の変数と展開を行う点を指定し, 級数表示を行う次数の指定を行います. すると, Maxima は指定した次数を越えない次数迄の級数として, Taylor 級数の表示を行います.

6.3.2 Taylor 級数に関連する関数

— Taylor 級数に関連する関数 —

```

taytorat(<式>)
taylor_simplifier(<式>)
taylorinfo(<式>)
taylorp(<式>)

```

taytorat は Taylor 展開の内部表現を CRE 表現に変換する為の関数です.

taylorinfo 関数は与式が Taylor 展開式であればその情報を返し, Taylor 展開式でなければ false を返す関数です.

taylorp 関数は真理値集合を {true, false} とする真理関数です.

taylor 級数に関連する大域変数

大域変数	値	概要
maxtayorder	true	
taylordepth	3	taylor 展開で指定された次数 n に対し $n2^{\text{taylordepth}}$ 次の項までを計算.
taylor_logexpand	true	
taylor_order_coefficients	true	taylor 級数の係数順序を制御
taylor_truncate_polynomials	true	taylor 級数表示を制御

Taylor 級数を変換する関数

`pade( $\langle Taylor \text{ 級数} \rangle, \langle \text{整数}_1 \rangle, \langle \text{整数}_2 \rangle$ )`

pade 関数は与えられた Taylor 級数を近似表現する有理式を返す関数です. 第二引数の整数が³, 近似有理式の分子の多項式の最高次数の上限を定め, 第三引数の整数が近似有理式の分母の多項式の最高次数の上限を定めます.

6.4 式について

Maxima の式は §5.1 の §5.1.5 で述べた様に, Maxima の記号や数値を演算子や関数項で繋ぐ事で構築された対象です.

ところで, Maxima の式は二面性を持っています. 一つは Maxima が表示する数式に近い表示であり, もう一つは Maxima を記述した LISP の処理に適したデータ構造, 則ち, 内部表現です. 実際の Maxima の処理では表に現われる式の操作ではなく, 内部表現を念頭に置いて処理を行う事が必要となります.

6.4.1 変数や文字列の内部表現

先ず, 式を構成する上で基礎となる変数と文字列の内部表現を示しましょう;

Maxima の内部表現：変数

Maxima	内部表現
<code>a</code>	<code>\$A</code>
<code>?a</code>	<code>A</code>
<code>"a"</code>	<code>"a"</code>
<code>'a</code>	<code>((MQQUOTE) \$A)</code>

Maxima の変数 (記号) は内部で先頭に \$ を付けたものになっています. 一方で, Maxima で ? が先頭に付く対象は内部では ? が外されています. その為, LISP の関数を Maxima から実行させる場合, 先頭に ? を付けてやれば使える事になります. 但し, 引数等は Maxima を通して与えられる為, Maxima の関数の様に引数を与える必要があります. 次に, 'a の内部表現は ((MQQUOTE) \$A) となっています. ここで a が \$a となるのは理解出来ませんが, 何故, (MQQUOTE) であって, MQQUOTE でないのでしょうか? 先ず, (MQQUOTE) はリストで, その被演算子 \$a は原子になりますね. こうする事で, 関数と被演算子を区別する事が可能になります. 即ち, 関数は被演算子よりも一階高い対象である事が表現出来るのです. この考え方は Maxima の内部表現では一貫しています.

注意が必要なのは文字列の扱いです. Maxima-5.13.0 以前では Maxima の文字列と LISP の文字列は別物です. 例えば, Maxima の文字列 "ABCD1" は先頭に & を付けて, &ABCD1 と表現されています. ところが, Maxima-5.14.0 以降から表に示す様に, Maxima の文字列と LISP の文字列が一致する仕様に変更されています. この点は使っている環境毎に確認する様にして下さい.

6.4.2 二項演算の内部表現

先ず, 代表的な二項演算子 (内挿表現の演算子) の内部表現を示します:

Maxima の内部表現: 数値演算

Maxima の式	内部表現
$x+y$	<code>((MPLUS SIMP) \$X \$Y)</code>
$x-y$	<code>((MPLUS SIMP) \$X ((MMINUS SIMP) \$Y))</code>
$x*y$	<code>((MTIMES SIMP) \$X \$Y)</code>
x/y	<code>((MTIMES SIMP) \$X ((MEXPT SIMP) \$Y -1))</code>
$x \cdot y$	<code>((MNC TIMES SIMP) \$X \$Y)</code>
x^2 又は $x**2$	<code>((MEXPT SIMP) \$X 2)</code>
$x^{^2}$	<code>((MNCEXPT SIMP) \$X 2)</code>

基本的に `変数1 演算子名 変数2` の内部表現は `((演算子名) 変数1 変数2)` となります。猶, Maxima 組込の演算子名は全て `m` から開始している事に注意して下さい。 `m` を先頭に付けて Maxima の関数と LISP の関数を区別しています。更に, 演算子名は `()` で括られ, 演算子が被演算子よりも一階高い対象である事を明瞭に示しています。更に, 差 $x-y$ は実質的に $x+(-y)$ で内部では表現されている事に注意して下さい。ここで $x-y$ の表現を調べてみましょう。使う関数は LISP の基本の基本, `car` と `cdr` です。Maxima から LISP 関数を利用する方法には, 演算子? を使って Maxima の関数として LISP の関数を利用する方法と, `:lisp` 演算子を使って直接 LISP の関数を入れる方法があります。ここでは, 演算子? を使う方法で調べてみましょう。

```
(%i74) ?car(x-y);
(%o74)          ("+", simp)

(%i75) ?cdr(x-y);
(%o75)          (x, - y)
```

演算子? を用いて LISP の関数を実行すると, その結果は Maxima で解釈された結果が帰って来ます。この例からも, 式 $x-y$ が Maxima 内部では $x+(-y)$ になっている事が分るかと思います。

同様に論理演算子の一覧を示しますが, 内容的には数値演算のものと同様です。

Maxima の内部表現：論理演算子

Maxima の式	内部表現
<code>not a</code>	<code>((MNOT SIMP) \$A)</code>
<code>a or b</code>	<code>((MOR SIMP) \$A \$B)</code>
<code>a and b</code>	<code>((MAND SIMP) \$A \$B)</code>
<code>a = b</code>	<code>((MEQUAL SIMP) \$A \$B)</code>
<code>a > b</code>	<code>((MGREATER SIMP) \$A \$B)</code>
<code>a >= b</code>	<code>((MGEQP SIMP) \$A \$B)</code>
<code>a < b</code>	<code>((MLESSP SIMP) \$A \$B)</code>
<code>a <= b</code>	<code>((MLEQP SIMP) \$A \$B)</code>
<code>a # b</code>	<code>((MNOTEQUAL SIMP) \$A \$B)</code>

6.4.3 割当の演算子の内部表現

割当の演算子にも内部表現があります:

Maxima の内部表現：割当の演算子

Maxima の式	内部表現
<code>a:b</code>	<code>((MSETQ SIMP) \$A \$B)</code>
<code>a::b</code>	<code>((MSET SIMP) \$A \$B)</code>
<code>a(x):=f</code>	<code>((MDEFINE SIMP) ((\$A) \$X) \$F)</code>

最初の `a:b` は内部で `((MSETQ SIMP) $A $B)` と表現されていますが, 余計な括弧と `SIMP` を外してしまえば, `(MSETQ $A $B)` となって LISP の `(setq a b)` に対応する事が分りますね.

関数定義を行う演算子 `:=` も LISP の `(defun a(x) f)` に似た並びで内部表現されている事が判ります. 但し, 関数名は変項よりも高い階数を表現する為に `()` を用いて囲う事が徹底されていますね.

6.4.4 Maxima の関数の内部表現

Maxima の内部表現：関数

Maxima の式	内部表現
<code>a(x)</code>	<code>(((\$A SIMP) \$X)</code>
<code>sin(x)</code>	<code>(((%SIN SIMP) \$X)</code>
<code>diff(y,x)</code>	<code>(((\$DIFF SIMP) \$Y \$X 1)</code>
<code>diff(y,x,2,z,1)</code>	<code>(((\$DIFF SIMP) \$Y \$X 2 \$Z 1)</code>
<code>'diff(y,x)</code>	<code>(((%DERIVATIVE SIMP) \$Y \$X 1)</code>
<code>integrate(a,b,c,d)</code>	<code>(((\$INTEGRATE SIMP) \$A \$B \$C \$D)</code>
<code>'integrate(a,b,c,d)</code>	<code>(((%INTEGRATE SIMP) \$A \$B \$C \$D)</code>

関数名には基本的に\$が先頭に付きますが, 単引用符' が先頭に付いた場合は%で置換えられます. この%が先頭に付いた関数は Maxima で自動的に解釈される事はありません. 則ち,%が内部書式の先頭に付いている事で名詞型と解釈され,Maxima はそれ以上の解釈を行いません. 又, 関数名は()で括られる事で変項よりも一階高い対象である事が表現されています.

微分を行う diff 関数の例に示す様に一階の微分を行う場合, 階数の 1 を省略しても構いませんが内部的には補完されている事が分りますね.

```
(%i93) ?car('diff(x,y));
(%o93) (derivative, simp)
(%i94) ?cdr('diff(x,y));
(%o94) (x, y, 1)
```

ここで名詞型に関連する関数として,nounify 関数と verbify 関数があります.

nounify 関数と verbify 関数

```
nounify(< 記号 >)
verbify(< 記号 >)
```

nounify 関数は Maxima の記号を名詞化して返す関数です. 内部的には先頭の\$を%で置換する関数です. この逆を行う関数が verbify 関数です.

```
(%i9) test1:a$
(%i10) :lisp $test1;
```

```

$A
(%i10) test2:nounify(a)$
(%i11) :lisp $test2;
%A
(%i11) test3:verbify(nounify(a))$
(%i12) :lisp $test3;
$A

```

ここで函数名は記号である為, これらの函数を用いれば函数項の名詞化や動詞化が容易に行える事になります.

6.4.5 配列とリストの内部表現

Maxima の内部表現：函数

Maxima の式	内部表現
<code>a[1,2]</code>	<code>(((\$A SIMP ARRAY) 1 2)</code>
<code>a[1,2](x)</code>	<code>((MQAPPLY SIMP) ((\$A SIMP ARRAY) 1 2) \$X)</code>
<code>[a,b,c]</code>	<code>((MLIST SIMP) \$A \$B \$C)</code>

先ず, 配列の内部表現の第一成分は最初に\$が付けられた配列名があり, その後に毎度の SIMP と最後の ARRAY でデータが配列である事を示しています.

Maxima のリストも LISP のリストとは別の構造で, 他の演算子等と同様に通常のリストに (MLIST SIMP) が先頭に置かれ, 変数には何時もの\$ が先頭に付加されています.

6.4.6 Maxima の制御文の内部表現

ここでは if 文, for 文と block 文の内部表現を示しましょう.

if 文 if 文の場合, LISP の cond と関連付けて mcond となっています:

Maxima の内部表現：if 文

<code>if a then b</code>	<code>((MCOND SIMP) \$A \$B T NIL)</code>
<code>if a then b else c</code>	<code>((MCOND SIMP) \$A \$B T \$C)</code>

for 文 for 文は LISP の do と関連付けて, mdo となっています:

Maxima の内部表現: for 文

```
for i:a thru b step c unless q do f(i)  ((MDO SIMP) $I $A $C
                                         NIL $B $Q (($F SIMP)
                                         $I))
for 1:a next n unless q do f(i)         ((MDO SIMP) $I $A NIL
                                         $N NIL $Q (($F SIMP)
                                         $I))
for i in l do f(i)                      ((MDOIN SIMP) $I $L
                                         NIL NIL NIL NIL (($F
                                         SIMP) $I))
```

block 文

block 文は LISP の prog と関連付けて, mprog となっています:

Maxima の内部表現: block 文

```
block([l1,l2],s1,s2)  ((MPROG SIMP) ((MLIST SIMP) $L1 $L2) $S1
                        $S2)
block(s1,s2)          ((MPROG SIMP) $S1 $S2)
```

表に示す様に, Maxima の各制御文の内部表現は S 式になっています. この様に, S 式で全てが記述されており, その S 式を裏の LISP が解釈しているのです.

6.4.7 表示式と内部表現

式の内部表現では, 式を構成する演算子項や関数項が全て前置式に変換されますが, この他にも, 表示式と内部表現が微妙に異なる書式を取る事があります. 則ち, 表示される式が必ずしも内部表現を反映したものではない事があります.

ここで代表的な表現の違いを次に示しておきましょう:


```

                                x
(%o26)                        -
                                y

(%i27) :lisp $exp1;
((MTIMES SIMP) $X ((MEXPT SIMP) $Y -1))
(%i27) :lisp $exp2;
((MQUOTIENT SIMP) $X $Y)

```

この様に、内部表現で自動的に変換されていた演算子が内部表現で表現される本来の演算子に置換えられています。この `dispform` 関数は与えられた CRE 表現もこの様な外部表現に変換します。

但し、`dispform(<式>)` では、式に含まれた関数の引数は内部形式のままです。この様に、一切の例外を認めずに式全てを外部表現に変換する為、`all` オプションを用いて、`dispform(<式>,all)` で変換を行います。

```

(%i40) expr1:f(sqrt(y/x));

                                y
(%o40)                        f(sqrt(-))
                                x

(%i41) expr2:dispform(expr1);

                                y
(%o41)                        f(sqrt(-))
                                x

(%i42) expr3:dispform(expr1,all);

                                y
(%o42)                        f(sqrt(-))
                                x

(%i43) :lisp $expr1
(($F SIMP)
 ((MEXPT SIMP) ((MTIMES SIMP) ((MEXPT SIMP) $X -1) $Y) ((RAT SIMP) 1 2)))
(%i43) :lisp $expr2
(($F SIMP)
 ((MEXPT SIMP) ((MTIMES SIMP) ((MEXPT SIMP) $X -1) $Y) ((RAT SIMP) 1 2)))
(%i43) :lisp $expr3
(($F SIMP) ((%SQRT) ((MQUOTIENT) $Y $X)))

```

この例では、 $f(\sqrt{\frac{y}{x}})$ を内部表現, `dispform` を作用させたもの, `dispform` に `all` オプション付きで作用させたものの結果を示しています。表示は全て同じものですが、内部表現では、`all` オプションを付けたもの以外は全て同じで、`all` オプションを付けたものは、式全体が外部表現になっています。

大域変数 `display_format_internal`

大域変数	初期値	概要
<code>display_format_internal</code>	<code>false</code>	内部表現に沿った表示への切替を制御

Maxima では大域変数 `display_format_internal` を切替える事で、内部表現に沿った表示を行わせる事が可能です。初期値の `false` であれば、表示の際に前述の様な変換を行う為、内部表現をそのまま表示しません。大域変数 `display_format_internal` を `true` に設定すると、内部表現に沿った表示に切替わります。

```
(%i1) display_format_internal;
(%o1) false
(%i2) [a-b,a/b,sqrt(x),x*4/3,exp(x),1/exp(x)];
          a          4 x    x    - x
(%o2)    [a - b, -, sqrt(x), ---, %e , %e  ]
          b          3
(%i3) display_format_internal:true$
(%i4) [a-b,a/b,sqrt(x),x*4/3,exp(x),1/exp(x)];
          - 1    1/2  4      x    (- 1) x
(%o4)    [a + (- 1) b, a b , x , - x, %e , %e  ]
```

6.4.8 変数と変数項

Maxima で変数や変数項として利用可能な対象は Maxima の記号と文字列です。更に、変数や変数項には項順序 $>_m$ が組込まれています (§5.2 参照)。

束縛変数の処理

Maxima 上で値が割当てられた変数、則ち、束縛変数は大域変数 `values` に登録されます。猶、何等の値が束縛されていない自由変数、Maxima の大域変数や内部変数は大域変数 `values` には登録されません。

Maxima に含まれる変数一覧

大域変数名	初期値	概要
values	[]	利用者の束縛変数が登録されるリスト

```
(%i1) a:1$
(%i2) b::sin(a);
(%o2)                                sin(1)
(%i3) x;
(%o3)                                x
(%i4) values;
(%o4)                                [a, b]
```

この例では大域変数 values には値が束縛された変数 a,b が登録されていますが, 自由変数の x は登録されていませんね.

大域変数 values に登録された束縛変数は remvalues 関数を用いて削除出来ます:

変数値の削除を行う関数

```
remvalue(<変数1> , <変数2> , ... )
remvalue (all)
```

remvalue 関数は指定した大域変数 values に登録された束縛変数を Maxima から削除します. この変数は添字されていても構いません. `remvalue(all)` で, 全ての束縛変数が削除されます.

```
(%i10) b1:sin(y)$
(%i11) x:1$
(%i12) c1:b1*x$
(%i13) values;
(%o13)                                [b1, c1, x]
(%i14) remvalue(x);
(%o14)                                [x]
(%i15) c1;
(%o15)                                sin(y)
(%i16) values;
(%o16)                                [b1, c1]
```

```
(%i17) remvalue(all);
(%o17) [b1, c1]
(%i18) values;
(%o18) []
```

この例では、変数 $b1, x, c1$ に値を割当て、最初に変数 x を削除し、最後に all で全ての変数を削除しています。変数 $c1$ の割当てでは変数 x を利用していますが、入力直後に評価された値が変数 $c1$ に割当てられている為、変数 x を削除しても問題はありません。猶、属性を利用して `remove` 関数で同様の処理が可能です。

式中の変数の処理

Maxima には式中的変数を取り出したりする関数が幾つか用意されています。

式中的変数を扱う関数

```
args(< 式 >)
listofvars(< 式 >)
```

`args` 関数は `< 式 >` が関数項や演算子項であれば、その引数のリストを返し、通常の式であれば項のリストを返します。則ち、内部表現式の引数をリストで返却する関数です。`args` 関数は `substpart("[" , < 式 >, 0)` と同値の処理を実行する関数で、多項式の CRE 表現から変数を取り出す `showratvars` 関数とは違います。

```
(%i21) args(sin(x));
(%o21) [x]
(%i22) args(sin(x+y));
(%o22) [y + x]
(%i23) expand((sin(x)+y)^2);
(%o23) y^2 + 2 sin(x) y + sin^2(x)
(%i24) args(%);
(%o24) [y^2, 2 sin(x) y, sin^2(x)]
```

ここで `args` 関数と `substpart` 関数の両方は大域変数 `inflag` の設定に依存します。`listofvars` 関数は `< 式 >` の変数リストを生成します。ここで、大域変数 `listconstvars` が `true` であれば、`< 式 >` に Maxima の数学定数 `%e, %pi, %i` や定数として宣言した変数で含

まれているものがあれば, `listofvars` 関数が返すリストに, これらの定数が含まれます. 但し, デフォルトの `false` の場合, 定数を除外したリストを返却します.

— `listofvars` の動作を制御する大域変数 —

変数名	初期値	概要
<code>listconstvars</code>	<code>false</code>	Maxima の定数を与式の変数リストに加えるかどうかを制御
<code>listdummyvars</code>	<code>true</code>	疑似変数を与式の変数リストに加えるかどうかを制御

大域変数 `listconstvars` が `true` であれば, 定数として宣言した変数と Maxima の数学定数 `%e`, `%pi`, `%i` が式に含まれていると, `listofvars` 関数はこれらの定数も変数として加えたリストを返します.

大域変数 `listconstvars` が `false` の場合, 数学定数と定数として宣言された変数は除外され, `listvars` が返すリストには含まれません.

```
(%i6) listofvars(x^2*y+aa+%e);
(%o6) [x, y]
(%i7) listconstvars:true;
(%o7) true
(%i8) listofvars(x^2*y+aa+%e);
(%o8) [%e, aa, x, y]
```

大域変数 `listdummyvars` が `false` であれば, 式の疑似変数は `listofvars` 関数が出力するリストの中に含まれません. 猶, 疑似変数は `sum` 関数や `product` 関数等の添字や極限変数や定積分変として利用される変数です.

6.4.9 部分式に分解する関数

Maxima には与えられた式を部分式に分解する関数があります. 先ず, `isolate` 関数と `disolate` 関数は指定した変数を含む式と含まない式に分解して表示します.

— 指定した変数を分離して式を表示する関数 —

```
isolate(< 式 >, < 変数 >)
disolate(< 式 > < 変数1 >, ..., < 変数n >)
```

isolate 関数は〈式〉から〈変数〉との積を持つ部分式と持たない部分式に分けて表示します。〈変数〉を持たない部分式は中間ラベルで置換えられ、式全体は中間ラベルと〈変数〉の項の和で表現されます。猶、isolate 関数は式の展開を行いません。単純に、与えられた式を指定された変数を持つ項と持たない項に分けて表示するだけの関数です。猶、大域変数 isolate_wrt_times が false の場合、〈式〉は〈変数〉との積を持たない部分式と〈変数〉との積を持つ部分式に分解して表示します。

大域変数 isolate_wrt_times が true の場合、isolate は更に〈式〉を分解し、項も〈変数〉の冪とそれ以外の変数との積に分解して表示します。

```
(%i12) isolate_wrt_times:false$
(%i13) exp1:expand((1+a+x)^2);

              2              2
(%o13)      x  + 2 a x + 2 x + a  + 2 a + 1
(%i14) isolate(exp1,x);

              2
(%t14)      a  + 2 a + 1

              2
(%o14)      x  + 2 a x + 2 x + %t14
(%i15) isolate_wrt_times:true$
(%i16) isolate(exp1,x);

              2 a
(%t16)

              2
(%o16)      x  + %t16 x + 2 x + %t14
(%i17) isolate((1+a+x)^2,x);

              a + 1
(%t17)

              2
(%o17)      (x + %t17)
```

disolate 関数は isolate(〈式〉,〈変数〉) に似ていますが、こちらでは利用者が一つ以上の変数を同時に孤立させて表示する事が出来ます

isolate 関数に影響を与える大域変数

変数名	初期値	概要
exptisolate	false	指数項を範囲に含めるかを決定
isolate_wrt_times	false	表示を制御

大域変数 `exptisolate` が `true` であれば, `isolate(〈式〉, 〈変数〉)` の実行で 〈変数〉 に含まれる (%e の様な) 原子の指数項に対しても調べます.

大域変数 `isolate_wrt_times` が `false` の場合, `isolate` 関数は指定した変数を含まない項と含む項に分けて表示を行います.

`true` の場合, 更に式を分解し, 指定した変数を含む項も, 指定した積を除く項と指定した変数の項の積に分解して表示を行います.

```
(%i18) isolate_wrt_times;
(%o18)                                     false
(%i19) exp1:expand((a+b+x)^2);
                                     2          2          2
(%o19)          x  + 2 b x + 2 a x + b  + 2 a b + a
(%i21) isolate(exp1,x);

                                     2          2
(%t21)          b  + 2 a b + a

                                     2
(%o21)          x  + 2 b x + 2 a x + %t21
(%i22) isolate_wrt_times:true$
(%i23) isolate(exp1,x);

                                     2 a
(%t23)

                                     2 b
(%t24)

                                     2
(%o24)          x  + %t24 x + %t23 x + %t21
```

`isolate` 関数では分離して表示するだけでしたが, 指定した変数を含む部分と含まない

部分に分解する関数 `partition` と純虚数を含む項と含まない項に分解する関数 `rectform` があります.

—— 部分式に分解する関数 ——

```
partition(⟨ 式 ⟩,⟨ 変数 ⟩)
rectform(⟨ 式 ⟩)
```

`partition` 関数は与えられた ⟨ 式 ⟩ を分解し、二つの部分式を成分とするリストを返します. これらの部分式は ⟨ 式 ⟩ の第一層に属するもので、第一成分が ⟨ 変数 ⟩ を含まない部分式、第二成分が ⟨ 変数 ⟩ を含む部分式となります.

```
(%i89) part(x+1,0);
(%o89)                                     +
(%i90) partition(x+1,x);
(%o90)                                   [1, x]
(%i91) part((x+1)*y,0);
(%o91)                                     *
(%i92) partition((x+1)*y,x);
(%o92)                                   [y, x + 1]
(%i93) part([x+1,y],0);
(%o93)                                     [
(%i94) partition([x+1,y],x);
(%o94)                                   [[y], [x + 1]]
```

`rectform` 関数は与式を $a+b\%i$ の形式で返します. ⟨ 式 ⟩ が複素数の場合、 a と b は実数になりますが、そうでない場合、純虚数 $\%i$ を持たない部分と純虚数 $\%i$ で括られた式に分解し、内部の項順序に従って出力される為、正確に $a+b\%i$ の書式にはなりません.

```
(%i12) rectform((x+%i)^3);
(%o12)          3          2
              x  + %i (3 x  - 1) - 3 x
(%i13) rectform((10+%i)^3);
(%o13)          299 %i + 970
```

6.4.10 部分式を扱う函数

Maxima の式には函数や演算子を節とする木構造表現が表にあり、それを内部で表現する内部表現の二つがあります。式の木構造では、演算子や函数の情報は比較的分かり易いものですが、内部表現は式の前置式表現を基にしたもので、判り難いものとなっています。Maxima の式を操作する函数には、式の木構造を基に式を操作する函数と、式の内部表現を直接操作する函数の二種類があります。

まず、部分式を取出す函数には、内部表現を直接利用する `inpart` 函数、一般的な木構造表現を用いる `part` 函数の二種類があります。

—— 部分式を取出す函数 ——

```
inpart(⟨ 式 ⟩, ⟨ 整数1 ⟩, ..., ⟨ 整数k ⟩)
inpart(⟨ 式 ⟩, [⟨ 整数1 ⟩, ..., ⟨ 整数k ⟩])
part(⟨ 式 ⟩, ⟨ 整数1 ⟩, ..., ⟨ 整数k ⟩)
part(⟨ 式 ⟩, [⟨ 整数1 ⟩, ..., ⟨ 整数k ⟩])
op(⟨ 式 ⟩)
pickapart(⟨ 式 ⟩, ⟨ 整数 ⟩)
```

`inpart` 函数は ⟨ 式 ⟩ の内部表現に直接作用する函数で、⟨ 整数₁ ⟩, ..., ⟨ 整数_k ⟩ で指定された ⟨ 式 ⟩ の部分式を取出す函数です。この `inpart` 函数は、式の内部表現に直接作用するので、その分、処理が速くなります。

`part` 函数は `inpart` 函数に似ていますが、⟨ 式 ⟩ の木構造表現に対応する部分式を取出します。

部分式の取出し方は、最初に ⟨ 式 ⟩ から ⟨ 整数₁ ⟩ で指定される部分式を取出します。次に、取出した部分式から ⟨ 整数₂ ⟩ で指定される成分を取出します。以降同様に ⟨ 整数_{k-1} ⟩ で指定された部分式から ⟨ 整数_k ⟩ で指定される成分を取出して、この部分式を結果として返します。

```
(%i15) part((x+1)^3+2,1);
```

3

```
(%o15) (x + 1)
```

```
(%i16) part((x+1)^3+2,1,1);
```

```
(%o16) x + 1
```

```
(%i17) part((x+1)^3+2,1,1,1);
```

```
(%o17) x
```

猶、[⟨ 整数₁ ⟩, ..., ⟨ 整数_n ⟩] の様にリストで指定する事も可能ですが、この場合は、第一

層の〈整数₁〉で指定された部分式,以降,〈整数_n〉で指定される部分式が取出され,これらの部分式に `part(〈式〉,0)` で得られる主演算子を作用させた式が返されます.

```
(%i72) expr:x+y+sin(x^2+2*x+1)+cos(z/w);
              z          2
(%o72)      cos(-) + y + sin(x  + 2 x + 1) + x
              w

(%i73) inpart(expr,[2,4]);
              z          2
(%o73)      cos(-) + sin(x  + 2 x + 1)
              w

(%i74) part(expr,[1,4]);
              z
(%o74)      cos(-) + x
              w

(%i75) expr2:x*y*z*sin(x^2+1);
              2
(%o75)      x sin(x  + 1) y z

(%i76) inpart(expr2,[1,4]);
              x z
(%o76)

(%i77) part(expr2,[1,2]);
              2
(%o77)      x sin(x  + 1)

(%i78) inpart(expr,0);
              +
(%o78)

(%i79) inpart(expr2,0);
              *
(%o79)
```

この例で示す様に, リストで指定した場合には部分式を抜き出して主演算子を作用させたものが返されています. その為, 和や積, 一つだけの被演算子を取る演算子 (unary) としての演算子-, 差と商を扱う場合, 部分式の順序に注意が必要になります.

part 函数に関連する大域変数

変数名	初期値	概要
piece		inpart/part 函数で取出した部分式を一時保存
partswitch	false	inpart/part 函数のエラーメッセージを制御

大域変数 piece には inpart 函数や part 函数を用いて取出した最新の部分式が保存されます。

大域変数 partswitch が true に設定していると, 式に指定した成分が存在しない場合に inpart 函数と part 函数は end を返します.false の場合は, エラーメッセージを返します。

op 函数は part 函数の機能省略版とも言えます。実際, `op(<式>)` は `part(<式>,0)` と同じです。

pickapart 函数は <整数> で指定された式の階層に含まれる全ての部分式を %t ラベルに割当て, ラベルを用いた式に <式> を変換します。階層指定は part 函数と同様で, 表示された形式に対して指定を行います。pickapart 函数は大きな式を扱う際に, part 函数を使わずに部分式に変数を自動的に割当てる事にも使えます。

```
(%i49) exp:(x+1)^3;
                                     3
(%o49)                               (x + 1)
(%i50) pickapart(exp,1);
                                     3
(%o50)                               %t48
(%i51) exp2:expand((x+1)^3);
                                     3      2
(%o51)                               x  + 3 x  + 3 x + 1
(%i52) pickapart(exp2,1);
                                     3
(%t52)                               x

                                     2
(%t53)                               3 x
```

```
(%t54)                                     3 x
```

```
(%o54)                                %t54 + %t53 + %t52 + 1
```

6.4.11 総和と積

— Σ と Π —

```
product(〈式〉, 〈添字変数〉, 〈下限〉, 〈上限〉)
sum(〈式〉, 〈添字変数〉, 〈下限〉, 〈上限〉)
lsum(〈式〉, 〈添字変数〉, 〈リスト〉)
```

product 関数は〈添字変数〉の〈下限〉から〈上限〉迄の〈式〉の値の積を与えます. 〈上限〉が〈下限〉より小になると空の積となり, この場合, product 関数はエラー出力ではなく 1 を返します. 評価は sum 関数と似ています.

— product 関数を制御する大域変数 —

変数名	初期値	概要
simpproduct	false	product 関数の簡易化を制御

大域変数 simpproduct は初期値が false になっています. この値を true に変更すると, product 関数の式を自動的に簡易化しようとします.

```
(%i23) product(k,k,1,n);

                                     n
                                   /==\
                                   ! !
(%o23)                               ! ! k
                                   ! !
                                   k = 1

(%i24) simpproduct:true$
(%i25) product(k,k,1,n);
(%o25)                               n!

(%i26) simpproduct:false;
(%o26)                               false
```



```
(%i27) product(k,k,1,n),simpproduct;
```

```
(%o27)                                     n!
```

この例では,simpproduct を true にする事で,product 関数の自動的な簡易化が行われている事に注意して下さい. 猶, この simpproduct は evflag を属性に持つ為,ev 関数による評価のオプションとしても使えます.

sum 関数は〈添字変数〉を〈下限〉から〈上限〉迄の値の〈式〉の和を取ります.〈上限〉と〈下限〉が整数で異なっていれば, 和の各々の項は評価されて互いに加えられます.

sum 関数を制御する大域変数

変数名	初期値	概要
simpsum	false	sum 関数の簡易化を制御
sumexpand	false	総和の積の処理を制御
cauchysum	false	Cauchy 積の適用を制御
genindex	i	sum や product で利用される疑似変数名を設定
gensumnum	false	疑似変数の番号

大域変数 simpsum が true の場合,sum 関数を自動的に簡易化します.

```
(%i33) simpsum;
```

```
(%o33)                                     false
```

```
(%i34) sum(x^n,n,0,m);
```

```

m
====
\      n
>    x
/
====
n = 0
```

```
(%i35) simpsum:true$
```

```
(%i36) sum(x^n,n,0,m);
```

```

m + 1
x    - 1
-----
x - 1
```

```
(%o36)
```

```
(%i37) sum(x^n,n,0,inf);
Is abs(x) - 1 positive, negative, or zero?
```

```
pos;
(%o37)                                     inf
(%i38) sum(x^n,n,0,inf);
Is abs(x) - 1 positive, negative, or zero?
```

```
neg;
(%o38)                                     1
-----
1 - x
```

```
(%i39) simpsum:false;
(%o39)                                     false
(%i40) sum(k,k,1,n),simpsum;
(%o40)                                     2
      n  + n
-----
      2
```

この様に simpsum が false の場合, $\text{sum}(x^n, n, 0, m)$ の簡易化は実行されませんが, simpsum が true となると, 自動的に簡易化が実行されます. $\text{sum}(x^n, n, 0, \text{inf})$ の場合, x の絶対値から 1 を引いたものが正か負か零であるかを利用者が指定する事で簡易化が行えます. 猶, simpsum は evflag 属性を持つ為, ev 函数による評価でも利用可能です. 大域変数 sumexpand が true の場合, sum 函数同士の積を纏めて sum 函数の入れ子にします.

```
(%i1) sum(f(x),x,0,m)*sum(g(x),x,0,n);
(%o1)
      m          n
      ====      ====
      \          \
      ( >  f(x)) >  g(x)
      /          /
      ====      ====
      x = 0      x = 0
```

```
(%i2) sumexpand:true$
(%i3) sum(f(x),x,0,m)*sum(g(x),x,0,n);
```

$$\sum_{i1=0}^m \sum_{i2=0}^n f(i1) g(i2)$$

```
(%o3)
```

但し、この処理は $\inf$ や $\min$ を含む総和に対して利用する事は危険です。

大域変数 `cauchysum` は大域変数 `sumexpand` と対で用います。大域変数 `sumexpand` と `cauchysum` の両方が `true` の場合、総和の掛算を纏める際に、通常の積ではなく Cauchy 積が利用されます。

```
(%i1) sumexpand:true$
(%i2) cauchysum:true$
(%i3) sum(f(x),x,0,m)*sum(g(x),x,0,n);
```

$$\sum_{i1=0}^m \sum_{i2=0}^n f(i1) g(i2)$$

```
(%o3)
```

```
(%i4) sum(f(x),x,0,inf)*sum(g(x),x,0,n);
```

$$\sum_{i3=0}^{\infty} \sum_{i4=0}^n f(i3) g(i4)$$

```
(%o4)
```

```
(%i5) sum(f(x),x,0,inf)*sum(g(x),x,0,inf);
      inf      i5
      ====     ====
      \        \
(%o5)  >        >      g(i5 - i6) f(i6)
      /        /
      ====     ====
      i5 = 0 i6 = 0
```

大域変数 `genindex` と大域変数 `gensumnum` は `sum` 関数内部の疑似変数を生成する為に用いられる大域変数です. 大域変数 `genindex` には疑似変数のアルファベットが, 大域変数 `gensumnum` には疑似変数の番号が各々設定されています. ここで, 大域変数 `gensumnum` は `sum` 関数内部で疑似変数を生成する度に, 一つずつ増加します.

```
(%i1) sumexpand:true$
(%i2) gensumnum;
(%o2) 0
(%i3) sum(f(x),x,0,m)*sum(g(x),x,0,n);
      m      n
      ====     ====
      \        \
(%o3)  >        >      f(i1) g(i2)
      /        /
      ====     ====
      i1 = 0 i2 = 0

(%i4) gensumnum;
(%o4) 2
```

この例では, 大域変数 `sumexpand` を `true` にした為, 総和の積が纏められてしまい, その結果, 二つの疑似変数 `i1` と `i2` が新たに生成されています. この時, `gensumnum` は最初が 0 で, それから二つの疑似変数を生成した為, 2 になっています.

最後の `lsum` 関数は `sum` 関数に似ていますが, 添字変数の定義域をリストで与えます. 但し, この定義域は必ずリストで与える必要は無く, リストで与えられない場合には, 名詞型で式を返します.

```
(%i2) lsum(i,i,[1,2,3,4,5,6]);
(%o2)
21
(%i3) lsum((i*x+1)^2,i,[1,3,5,7,9]);
(%o3)
      2      2      2      2      2
(9 x + 1) + (7 x + 1) + (5 x + 1) + (3 x + 1) + (x + 1)
(%i4) lsum((i*x+1)^2,i,mikeneko);
      ====
      \      2
      >    (i x + 1)
      /
      ====
      i in mikeneko
```

この例で示す様に,mikeneko は単なる原子で Maxima のリストと異なる為に, sum 関数の名詞型として式が返されています.

— sum 関数と prod 関数双方に影響する関数 —

```
bashindices(< 式 >)
niceindices(< 式 >)
```

bashindices 関数は sum 関数と prod 関数の添字を自動的に変更する関数です. この時, 大域変数 genindex の値は j(内部的には\$j) が割当てられる為, 与式の sum 関数や prod 関数の添字は j1, j2, ... で置換えられます.

```
(%i23) neko:sum(sum(sin(k*x)^1,l,1,inf),k,1,inf);
      inf    inf
      ====  ====
      \      \      1
      >      >    sin (k x)
      /      /
      ====  ====
      k = 1 l = 1
```

```
(%i24) bashindices(neko);
```

```

inf      inf
====  ====
\        \          1
(%o24)   >      >    sin (j2 x)
/        /
====  ====
j2 = 1 1 = 1

```

niceindices 関数は後述の大域変数 niceindicespref を利用する関数で, sum 関数や prod 関数による総和や積の添字をこの niceindicespref に登録された添字で置換えます.

```
(%i28) neko;
```

```

inf      inf
====  ====
\        \          1
(%o28)   >      >    sin (k x)
/        /
====  ====
k = 1 1 = 1

```

```
(%i29) niceindices(neko);
```

```

inf      inf
====  ====
\        \          i
(%o29)   >      >    sin (j x)
/        /
====  ====
j = 1 i = 1

```

— sum 関数と prod 関数双方に影響する大域変数 —

大域変数	初期値	概要
niceindicespref	[i,j,k,l,m,n]	sum 関数と prod 関数の双方で用いる添字を指定

大域変数 `niceindicespref` には, `niceindices` 関数で置換えられる添字のリストが割り当てられています. このリストに蓄えられた添字を使い切ると, リストの先頭の添字に 0 から開始する番号を割り当てられます.

```
(%i48) neko:prod(sum(sum(m*sin(k*x)^1,1,1,inf),k,1,inf),m,1,inf);
```

```

      inf      inf      inf
    /====\      =====
      ! !      \      \      1
(%o48)      ! ! m >      >      sin (k x)
      ! !      /      /
      m = 1      =====
                  k = 1 l = 1

```

```
(%i49) niceindicespref:[a,b];
```

```
(%o49) [a, b]
```

```
(%i50) niceindices(neko);
```

```

      inf      inf      inf
    /====\      =====
      ! !      \      \      a
(%o50)      ! ! a0 >      >      sin (b x)
      ! !      /      /
      a0 = 1      =====
                  b = 1 a = 1

```

6.4.12 式の符号を判定する関数

`sign` 関数は文脈を用いて与えられた式の正負を判定する関数です.

sign 関数

sign (< 式 >)

この `sign` 関数が返却する値は, {pos,neg,zero,pz,nz,pnz} になりますが, これらの値の意味を下の表 6.1 に示しておきます.

この `sign` 関数の実例を以下に示しておきましょう.

```
(%i42) assume(neko>0);
```

```
(%o42) [neko > 0]
```

表 6.1: sign 関数の返す値

pos	正の場合
neg	負の場合
zero	零の場合
pz	正か零の場合
nz	負か零の場合
pn	正か負の場合
pnz	判定不能

```
(%i43) sign(neko);
```

```
(%o43)                                pos
```

この例では, 変数 `neko` が 0 より大であると `assume` 関数を用いて仮定した為, `sign` 関数は変数 `neko` が `pos` であると返しています.

6.4.13 式の複製を行う関数

式の複製を行う関数として `copy` 関数があります.

—— copy 関数の構文 ——

```
copy(< 式 >)
```

この `copy` 関数の引数は Maxima の任意の式になります.

猶, 関数 `copy` の実体は内部関数の `copy-tree` 関数です. この関数は他に `copymatrix` 関数, `copylist` 関数でも用いられていますが, 夫々, `matrixp` 関数や `listp` 関数による引数の判定が行われる仕組みになっています. 但し, これらの関数は `copy-tree` 関数を用いて複製を生成する仕組みである為, 実質的に `copy` 関数の適用範囲を狭めたものでしかありません.

```
(%i33) pet:{みけ, ぼち, たま}$
```

```
(%i34) my_pet:copy(pet)$
```

```
(%i35) my_pet;
```

```
(%o35)                                {たま, ぼち, みけ}
```

```
(%i36) eq1:x^2+y+1$
```



```
(%i37) eqx:copy(eq1);
```

```
(%o37)
          2
      y + x  + 1
```

6.4.14 二項演算式の検出を行う関数

assoc 関数

```
assoc(< キーワード >, < リスト >)
assoc(< キーワード >, < リスト >, < 文字列 >)
```

assoc 関数は $x*y$ の様な一つの二項演算子と二つの被演算子の項で構成されたリストに対し、< キーワード > に適合する項があれば、その項の被演算子を返す関数です。又、第三引数に文字列を指定すると、適合に失敗すると false ではなく、指定した文字列を返します。

```
(%i4) assoc(x, [a*b, c^x, x*y]);
```

```
(%o4)
          y
```

```
(%i5) assoc(x, [a*b, c^x, a*x]);
```

```
(%o5)
          false
```

```
(%i6) assoc(x, [a*b, c^x, a*x], "残念でした");
```

```
(%o6)
          残念でした
```

```
(%i7) assoc(x, [a*b, c^x, x*a], "残念でした");
```

```
(%o7)
          残念でした
```

この関数は与式の内部構造に対して検出を行う為、例の $x*y$ と $a*x$ で結果が異なるのに対し、 $a*x$ と $x*a$ の結果が一致する理由となっています。

6.5 代入操作

多項式の計算で, 方程式を求めた結果を早速, 式に代入したい事があります. この場合, 規則による代入や, 直接変数に対して $x:a$ の様に割当てを行う事もありますが, これらとは別に subst 関数等の代入用の関数を用いる方法があります.

この節では最初に通常の代入関数について解説します. ここで通常とは, Maxima での式の表現に注意を払わなくても済む関数の事です. 例えば, 式中の変数 x に 2 を代入する様な関数です. Maxima には式の表現から部分式や演算子を指定して入れ換える関数があります. この関数は与式から部分式を取出す part 関数や inpart 関数に対応するものです.

6.5.1 通常の代入関数

subst 関数

subst($\langle$ 式₁ $\rangle$, $\langle$ 式₂ $\rangle$, $\langle$ 式₃ $\rangle$)

subst 関数は $\langle$ 式₃ $\rangle$ 中の $\langle$ 式₂ $\rangle$ を $\langle$ 式₁ $\rangle$ で置換します.

$\langle$ 式₁ $\rangle$ と $\langle$ 式₂ $\rangle$ は二重引用符で括られた式の演算子や関数名, 或いは, $\langle$ 式₂ $\rangle$ は Maxima の原子や $\langle$ 式₃ $\rangle$ に完全に含まれる部分式でなければなりません.

例えば, $x+y+z$ は $2*(x+y+z)*w$ に完全に含まれる部分式になりますが, $x+y$ は部分式になりません. これは式の木構造を考えると明瞭になります. 部分式は, 式の木構造を考えた時に, 各階層が構成する式を取出したものになるからです. ここで, $\langle$ 式₂ $\rangle$ が $\langle$ 式₃ $\rangle$ の部分式でない場合, subst ではなく, 式の階層を直接指定出来る substpart 関数や ratsubst 関数を使いましょう.

subst 関数では, $\langle$ 式₂ $\rangle$ が 式_a/式_b の様に割算を伴う場合, subst($\langle$ 式₁ $\rangle$ * $\langle$ 式_b $\rangle$, $\langle$ 式_a $\rangle$, $\langle$ 式₃ $\rangle$) が使えます. 又, $\langle$ 式₂ $\rangle$ が $\langle$ 式_a $\rangle^{1/\langle$ 式_b $\rangle}$ の形式であれば, subst($\langle$ 式₁ $\rangle^{\langle$ 式_b $\rangle}$, $\langle$ 式_a $\rangle$, $\langle$ 式₃ $\rangle$) が使えます.

subst に影響を与える大域変数

変数名	初期値	概要
exptsust	false	指数関数の代入操作を制御
opsubst	true	演算子に代入する事を抑制

大域変数 exptsust が true であれば $e^{(a*x)}$ の e^x を y で置換える操作が可能になります.

大域変数 `opsust` が `false` であれば, `subst` 関数は式に含まれる演算子に対して代入を行いません.

例えば, `(opsust:true,subst(x^2,r,r+r[0]))` と `(opsust:false,subst(x^2,r,r+r[0]))` を実行すると, 大域変数 `opsust` が `true` であれば, `subst` 関数は与式 `r+r[0]` の全ての `r` に `x^2` を代入しますが, 大域変数 `opsust` を `false` にすると左側の `r` のみに代入操作が行われ, 項 `r[0]` の `r` には `x^2` が代入されません.

```
(%i63) (opsust:true,subst(x^2,r,r+r[0]));
```

```
(%o63)          2      2
              x  + (x )
                  0
```

```
(%i64) (opsust:false,subst(x^2,r,r+r[0]));
```

```
(%o64)          2
              x  + r
                  0
```

ratsubst 関数

```
ratsubst (<式1>,<式2>,<式3>)
```

`ratsubst` 関数は `<式3>` に含まれる `<式2>` に `<式1>` を代入します. `<式2>` は和, 積, 冪等の演算子でも構いません. `subst` 関数が代入を行う個所で `ratsubst` 関数は式が何を意味するかを判っています. その為, `subst(a,x+y,x+y+z)` は `x+y+z` を返しますが, `ratsubst` 関数は `z+a` を返します.

ratsubst に影響を与える大域変数

変数名	初期値	概要
<code>ratsubstflag</code>	<code>false</code>	冪乗の扱いを制御

大域変数 `ratsubstflag` (`ratsubstflag` ではない事に注意) が `true` の場合, `ratsubst` 関数を使って項の `sqrt` や冪で式の入れ換えが可能となります.

例えば, `a` を `x^(1/3)` とした場合, `x` は `a^3` に等しくなりますが, この様に, 式 `x` の `x^(1/3)` を `a` で置換える事は通常出来ません. 大域変数 `ratsubstflag` が `true` の場合, この様な代入が行えます.

```
(%i5) ratsubstflag:true$
```

```
(%i6) ratsubst(a,x^(1/3),x);
```

```
(%o6)

$$\frac{3}{a}$$

```

fullratsubst 関数

```
fullratsubst(< 式1>, < 式2>, < 式3> )
```

fullratsubst 関数は ratsubst 関数と同じですが、結果が変化しなくなる迄、自分自身を再帰的に呼出します。この関数は、式の置き換えや置き換えられた式が一つ又はそれ以上の変数を共通に持つ場合に便利です。fullratsubst 関数は lratsubst 関数と同じ引数の書き方が出来ます。最初の引数は単一の代入方程式かその様な方程式のリストで、第二の引数は仮定された式となります。

lratsubst 関数

```
lratsubst(< リスト >, < 多項式 > )
```

lratsubst 関数は subst(< 方程式のリスト >, < 式 >) に似ていますが、ratsubst 関数が subst 関数の代りに使われる点で異なります。lratsubst 関数の最初の因子は方程式か方程式のリストで、subst 関数から得られる書式と同一のものでなければなりません。代入は方程式のリストの左から右の順で処理します。

```
(%i1) load ("lrats")$
(%i2) subst ([a = b, c = d], a + c);
(%o2)

$$d + b$$

(%i3) lratsubst ([a^2=b, b=c^2, c^3=d], a^2+b+c^3);
(%o3)

$$d + 2c$$

(%i4) subst ([b=c^2, a-2=b, c^3=d], a^2+b+c^3);
(%o4)

$$d + c^2 + a^2$$

(%i4) lratsubst ([b=c^2, a-2=b, c^3=d], a^2+b+c^3);
(%o4)

$$d + c^2 + b^2 + 4b + 4$$

```

sublis 関数

```
sublis(< リスト >, < 式 > )
```

sublis 関数は〈式〉に〈リスト〉で指定した複数の代入を並行して行ないます。〈リスト〉には、 $a=b$ の書式で式を記述します。演算子=の左辺の a が〈式〉に含まれる原子や関数名を指定し、右辺の b に置換える値や式を設定します。

```
(%i23) sublis([sin=cos,x=2*theta+1],sin(x-1)^2);
2
(%o23) cos (2 theta)
(%i24) sublis([sin=cos,cos=sin],cos(x)^2+sin(x+1)^3);
3 2
(%o24) cos (x + 1) + sin (x)
```

猶、sublis([sin=cos,cos=sin],cos(x)^2+sin(x+1)^3) の様な入れ換えの指定では、〈リスト〉に含まれる式の代入を順番に行うのではなく同時に行う為、cos と sin が入れ換えられている事に注意して下さい。

大域変数 sublis_apply_lambda が sublis 関数を実行した後の簡易化を制御します。

6.5.2 subspart 関数と substinpart 関数

subspart 関数と substinpart 関数は与式の部分式を取出す part 関数と 関数に各々対応する代入関数です。part 関数は式の木構造に対して部分式を取出す関数で、inpart 関数は Maxima の式の内部表現に対して部分式を取出す関数です。subspart 関数と substinpart 関数の対応もこれと同様です。その為、大域変数 inflag を true に設定して part/subspart 関数を呼出す事は、inpart/substinpart 関数を呼出す事と同じ事になります。

— subspart と substinpart 関数 —

```
subspart(〈式1〉,〈式2〉,〈正整数1〉,⋯,〈正整数n〉)
substinpart(〈式1〉,〈式2〉,〈正整数〉,⋯)
```

subspart 関数は〈式₂〉から part 関数の様に〈正整数₁〉,⋯,〈正整数_n〉で抽出した部分式に〈式₁〉を代入します。〈式₁〉に演算子を入れる場合には、二重引用符で subspart("+",a*b,0) の様に括る必要があります。

具体的な例で説明しましょう。式 $\frac{1}{x^3+3x^2+1}$ の成分の入れ換えを行いましょ。この式の構造は、図 6.1 に示すものになります。

subspart 関数はこの構造に従って入れ換えを行います。その為、演算子や式で成分を置換える事も可能です。

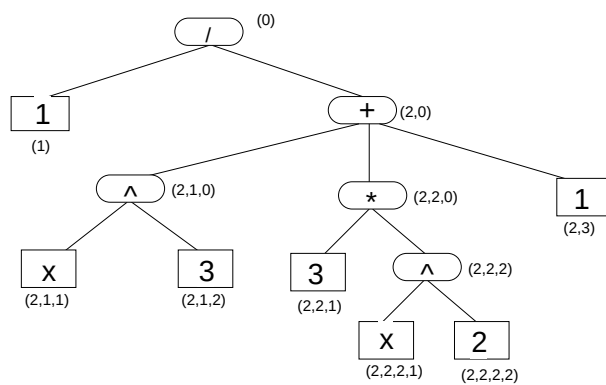


図 6.1: $\frac{1}{x^3+3x^2+1}$ の構造

```
(%i7) 1/(x^3+3*x^2+1);
```

```
(%o7)
```

$$\frac{1}{x^3 + 3x^2 + 1}$$

```
(%i8) substpart(4,%,2,1,2);
```

```
(%o8)
```

$$\frac{1}{x^4 + 3x^2 + 1}$$

```
(%i9) substpart(1,%,2,2,2,2);
```

```
(%o9)
```

$$\frac{1}{x^4 + 3x + 1}$$

```
(%i10) substpart(x,%,1);
```

```
(%o10)
```

$$\frac{x}{x^4 + 3x + 1}$$

```
(%i11) substpart("^",%,0);
```

```

              4
            x  + 3 x + 1
(%o11)      x
(%i12) substpart(sin(x),%,1);

              4
            x  + 3 x + 1
(%o12)      sin(x)
(%i13) substpart(y,%,2);

              y
(%o13)      sin (x)

```

substpart 関数は substpart 関数に似ていますが,substpart 関数と違い, 式の内部表現に対して作用します.

6.6 式の展開と簡易化

6.6.1 自動展開を行う大域変数

Maxima では $x+1-x*2+4$ の様な式を入力すると同時に簡易化が行われます. これは入力式の簡易化を行う内部関数 `simplify` が, 大域変数 `simp` で制御されている為で, 大域変数 `simp` が初期値の `true` であれば, 内部関数 `simplify` は入力式に含まれる関数の `operators` 属性や演算子に設定された属性に対応する内部関数を用いて自動簡易化や評価が実行される仕組みになっています (§5.8.1 参照).

```
(%i1) x^2+2-x^2*2+4+4/2;
                                     2
(%o1)                               8 - x
(%i2) simp;
(%o2)                               true
(%i3) simp:false$
(%i4) x^2+2-x^2*2+4+4/2;
                                     2      2      4
(%o4)                               x  + 2 + (- x ) 2 + 4 + -
                                     2
```

実際, 大域変数 `simp` の値を `false` に設定すると式の簡易化が一切実行されていません. この様な式の簡易化に関連する大域変数が数多くあります. 次に自動評価に関連する大域変数を纏めておきましょう.

自動展開に関連する大域変数

変数名	初期値	概要
<code>demoivre</code>	<code>false</code>	指数関数の表示を制御
<code>%emode</code>	<code>true</code>	指数関数の簡易化を制御
<code>%enumer</code>	<code>false</code>	自然底の浮動小数点への自動変換
<code>negdistrib</code>	<code>true</code>	$-(A+B+\dots+Z)$ を $-A-B\dots-Z$ に自動変換
<code>numer</code>	<code>false</code>	数値を含む式の自動評価を実行
<code>simp</code>	<code>true</code>	入力式の数値部分の自動的簡易化を実行
<code>sumexpand</code>	<code>false</code>	<code>sum</code> 関数の簡易化を実行
<code>simplproduct</code>	<code>false</code>	<code>product</code> 関数の簡易化を制御

先ず, 大域変数 `demoivre` を `true` で, 与式 $e^{(a+b*i)}$ の b が実数の場合, 与式は $e^a * (\cos(b) + i * \sin(b))$ に自動的に展開されます.

```
(%i18) exp(a+b*i);
                                %i b + a
(%o18)                                %e
(%i19) demoivre:true$
(%i20) exp(a+b*i);
                                a
(%o20) %e (%i sin(b) + cos(b))
```

大域変数 `emode` が `true` であれば, $e^{(\pi*i*x)}$ が次の様に簡易化されます.

- x が整数, 或いは $1/2, 1/3, 1/4$ や $1/6$ と整数の積であれば, $\cos(\pi*x) + i * \sin(\pi*x)$ となります.
- その他の数値の場合, $e^{(\pi*i*y)}$ となります. ここで y は $x - 2*k$, k は $\text{abs}(y) < 1$ を満す整数です.

大域変数 `emode` が `false` の場合, $e^{(\pi*i*x)}$ の簡易化は何も実行されません.

```
(%i25) %emode:true$
(%i26) exp(%pi*i/2);
                                %i
(%o26)                                %i
(%i27) %emode:false$
(%i28) exp(%pi*i/2);
                                %i %pi
                                -----
                                2
(%o28)                                %e
```

大域変数 `enumer` が `true` であれば, e は $2.718\dots$ に変換されます. e^x の指数が整数の場合に限り, この変換が行なわれます.

大域変数 `negdistrib` は -1 の積を分配するかどうかを決定する大域変数です. `true` の場合には -1 の積が分配されます.

```
(%i1) negdistrib;
(%o1)                                true
```

```
(%i2) -(a+b+c-d);
(%o2)          d - c - b - a
(%i3) negdistrib:false$
(%i4) -(a+b+c-d);
(%o4)          - (- d + c + b + a)
```

大域変数 `numer` は入力された式の数値を自動的に浮動小数点数に変換させる大域変数です. この大域変数は後述の大域変数 `float` に影響を及ぼしますが, `float` の設定の影響は受けません.

大域変数 `simp` は `true` の場合に, 和や差の演算子の簡易化を自動実行します.

大域変数 `sumexpand` と大域変数 `prodexpand` の詳細に関しては, §6.4 の `sum` 関数と `product` 関数を参照して下さい.

これらの変数にはフラグを持っていませんが, 大域変数の中には, 属性を持っている関数があります.

属性を持つ自動評価に関連する大域変数

変数名	初期値	属性	概要
<code>float</code>	<code>false</code>	<code>evflag</code>	<code>true</code> の場合, 非整数, 及び非整数を含む式を自動的に浮動小数点数へ変換する
<code>expop</code>	0	<code>fixnum</code>	正の冪乗の自動展開の上限を定める.
<code>expon</code>	0	<code>fixnum</code>	負の冪乗の自動展開の下限を定める.

大域変数 `float` は `numer` に似た大域変数です. `true` の場合, Maxima は整数以外の数値を自動的に浮動小数点数に変換します.

大域変数 `float` は大域変数 `numer` の影響を受けます. 実際, `numer:true` により, 自動的に `float:true` となり, 同様に `muner:false` と設定する事で, `float` も `false` に設定されます. しかし, 大域変数 `float` の設定に大域変数 `numer` は影響を受けません. 大域変数 `expon` は `expand` 関数とは別個に, Maxima が自動的に展開する式に含まれる負の冪の次数を定めます. 同様に, 大域変数 `expop` は自動的に展開される正の最も高い次数を定めます.

大域変数 `expon` と `expop` は初期値が 0 に設定されている為, $(x+1)^0$ の様に冪の次数が零であれば自動的に 1 に変換されます. 大域変数 `expop` を例えば 4 に変更すると, 冪の次数が 0 以上, 4 以下であれば Maxima は冪を自動的に展開します. 又, `expon` を 4 にすると, 負の冪の次数の絶対値が 0 以上, 4 以下であれば自動的に冪を展開します. 以下に簡単な例を示しましょう.

```
(%i38) expon:4$
```

```
(%i39) (x+1)^(-3);
```

```
(%o39)
```

$$\frac{1}{x^3 + 3x^2 + 3x + 1}$$

```
(%i40) (x+1)^(-5);
```

```
(%o40)
```

$$\frac{1}{(x + 1)^5}$$

```
(%i41) expop:4$
```

```
(%i42) (x+1)^4;
```

```
(%o42)
```

$$x^4 + 4x^3 + 6x^2 + 4x + 1$$

```
(%i43) (x+1)^5;
```

```
(%o43)
```

$$(x + 1)^5$$

6.6.2 指数関数の展開に関連する関数

指数関数の表示

```
demoivre(< 式 >)
exponentialize(< 式 >)
```

demoivre 関数は大域変数 demoivre の設定や ev 関数による式の再評価なしで変換を行います。

exponentialize 関数は < 式 > を指数関数形式に変換します。

6.6.3 式の展開に関連する函数

expand 函数

```

expand(< 式 >, < p >, < n >)
expand(< 式 >)
expan(< 式 >, p, n)
expandwrt(< 式 >, < 変数1 >, ..., < 変数n >)
expandwrt_factored(< 式 >, < 変数1 >, ..., < 変数n >)
pfet(< 式 >)

```

expand 函数は和の積や指数函数内の和を展開し、有理式の分子を各々の項に分離し、可換積と非可換積の両方の積を < 式 > の全ての階層で加法に対して分配します。

猶、多項式に対してはより効率的なアルゴリズムを用いる ratexpand を通常用いるべきです。

大域変数の maxnegex と maxposex は Maxima が展開する式の負と正の冪の次数の最大値を設定します。

expand 函数を制御する大域変数

変数名	初期値	属性	概要
maxnegex	1000	fixnum	expand で展開される負の冪の次数
maxpogex	1000	fixnum	expand で展開される正の冪の次数

大域変数 maxnegex は expand 函数で展開される絶対値が最大となる負の冪の次数です。正の冪の最大次数は maxposex です。

大域変数 maxposex は expand 函数で展開される最大の正の冪の次数です。猶、負の冪の最大次数は maxnegex です。

expan 函数は、expan(< 式 >, p, n) の場合、p を大域変数 maxposex, n を大域変数 maxnegex に割当てて < 式 > の展開を行います。

expandwrt 函数は < 変数₁ >, ..., < 変数_n > に対し、< 式 > を展開します。< 変数_i > を含む全ての積は明示的に現れます。返される形式は < 変数_i > を持つ式の和の積を持たないものとなります。< 変数_i > は変数、演算子や式でも構いません。

デフォルトで分母は展開されませんが、大域変数の expandwrt_denom でこれは制御が出来ます。この函数を使う為には予め `load(stopex);` で読込を実行します。expandwrt_factored 函数は expndwrt 函数に似ていますが、幾分違った式の積を扱います。ex-

pand_factored は要求される展開を処理しますが、引数リストの中の変数に含まれる〈式〉の因子に対してのみ処理を行います。予め、`load(stopex)` で読込を実行する必要があります。

pfet 関数は expand 関数に似た関数で、式の展開を行います。猶、pfet 関数では ratexpand 関数と内部関数の sssqfr 関数が用いられています。この関数の特徴は主変数に対して式の展開を行う事です。従って、expand 関数の様に式が完全に展開されたものにはなりません。

```
(%i61) pfet((x+y+6)^2+6+z);
```

```
(%o61)          2          2
      z + y  + 2 (x + 6) y + x  + 12 x + 42
```

```
(%i62) pfet(((x+y)*z+6)^2+6+z);
```

```
(%o62)          2  2
      (y + x)  z  + 12 (y + x) z + z + 42
```

```
(%i63) pfet(((x+z)*x+6)^2+6+z);
```

```
(%o63)          2  2          2          4          2
      x  z  + 2 x (x  + 6) z + z + x  + 12 x  + 42
```

この例では、最初の式の主変数は Maxima の項順序 $>_m$ により y となります。その為、 y に対しては展開されますが、その係数は展開されません。後の二つの例では主変数が z となる為、 z に対してのみ展開が行われます。

演算子の分配に関連する関数

```
distrib(〈式〉)
multthru(〈式1〉,〈式2〉)
multthru(〈式〉)
```

distrib 関数は可換積演算子 $*$ に対し和 $+$ を分配します。expand との違いは、式の最上層のみで作用する点です。又、multthru と最も上層の全ての和を展開する点でも異なります。

multthru 関数は〈式〉の部分展開を行います。

即ち、〈式〉が $f_1 * f_2 * \dots * f_n$ の形式で、各因子の中で、冪乗でない〈式〉中で最も左側の因子を f_i とすると、〈式〉の f_i 以外の因子と f_i の項との積の和に分解します。例えば、 $(x+1)^2 * (z+1) * (y+1)$ の場合、最も左側の因子 $y+1$ で式が展開され、 $(x+1)^2 * (y+1) * z + (x+1)^2 * (y+1)$ となります。

$\text{multthru}(\langle \text{式}_1 \rangle, \langle \text{式}_2 \rangle)$ の場合, $\langle \text{式}_2 \rangle$ の各項を $\langle \text{式}_1 \rangle$ 倍にします. つまり, $\text{multthru}(\langle \text{式}_1 \rangle * \langle \text{式}_2 \rangle)$ と同値です.

猶, $\langle \text{式}_2 \rangle$ には方程式を指定出来ます. この場合, 演算子=の二つの被演算子に $\langle \text{式}_1 \rangle$ との積が返されます.

猶, multthru は冪乗された和の展開は行いません. この函数は和に対する可換, 或いは, 非可換積の分配に関して最も速いものです.

```
(%i18) multthru((x+1)^2*(z+1));
```

```
(%o18)          2          2
      (x + 1) z + (x + 1)
```

```
(%i19) multthru((x+1)^2*(y+1)^2*(z+1)^2,z+1);
```

```
(%o19)          2          2          2          2          2          2
      (x + 1) (y + 1) z (z + 1) + (x + 1) (y + 1) (z + 1)
```

```
(%i20) multthru((x+1)^2*(y+1)^2*(z+1)^2,x+1);
```

```
(%o20)          2          2          2          2          2          2
      x (x + 1) (y + 1) (z + 1) + (x + 1) (y + 1) (z + 1)
```

```
(%i21) multthru((x+1)^2*(z+1)*(y+1));
```

```
(%o21)          2          2
      (x + 1) (y + 1) z + (x + 1) (y + 1)
```

```
(%i22) multthru((x+1)^2*(y+1)^2*(z+1)^2,x^2+1=0);
```

```
(%o22)          2          2          2          2          2          2
      x (x + 1) (y + 1) (z + 1) + (x + 1) (y + 1) (z + 1) = 0
```

distrib,multthru,expand の比較

$\text{distrib}((a+b)*(c+d))$	$\Rightarrow$	$b*d + a*d + b*c + a*c$
$\text{expand}((a+b)*(c+d))$	$\Rightarrow$	$b*d + a*d + b*c + a*c$
$\text{multthru}((a+b)*(c+d))$	$\Rightarrow$	$(b + a)*d + (b + a)*c$
$\text{distrib}(1/((a+b)*(c+d)))$	$\Rightarrow$	$1/((b + a)*(d + c))$
$\text{expand}(1/((a+b)*(c+d)),1,0)$	$\Rightarrow$	$1/(b*d + a*d + b*c + a*c)$

6.6.4 sum 関数の簡易化に関連する関数

sumcontract

sumcontract(<式>)

sumcontract 関数は上限と下限の差が定数となる加法の全ての総和を結合します. 結果は, 各総和の集合に対して, 全ての適切な外の項を加えて一つの総和にしたものを含む式になります. sumcontract は全ての互換な総和を結合し, 可能であれば, 総和の一つから添字の一つを用います. sumcontract を実行する前に intosum(<式>) の実行が必要かもしれません.

```
(%i18) sum(1/n^2,n,1,m)+sum(1/n^3,n,1,m);
```

```
(%o18)
      m      m
      ====
      \      1      \      1
      >      -- + >      --
      /      2      /      3
      ==== n      ==== n
      n = 1      n = 1
```

```
(%i19) sumcontract(%);
```

```
(%o19)
      m
      ====
      \      1      1
      >      ( -- + -- )
      /      2      3
      ==== n      n
      n = 1
```

intosum 関数

intosum(<式>)

総和の乗法がなされる全ての物を取り, それらを総和の内部に置きます. 添字が式の外側で用いられていれば, この関数は sumcontract に対して実行するのと同様に適切な添字を探そうとします. これは本質的に総和の outative 属性の観念の逆になりますが, この属性を取り除かずに素通りするだけである事に注意して下さい.

intosum 函数を用いる前に scanmap(multthru, <式>) を実行しなければならない場合もあります.

6.6.5 簡易化を行う函数

radcan 函数

```
radcan(<式>)
```

radcan 函数の引数 <式> は, 対数函数, 指数函数と冪乗根を含んでも構いません. <式> をある変数順序に対する CRE 表現に変換し, 簡易化を行います. 特定の変数順序に対し, CRE 表現は一意に定まります (従って, CRE 表現は式の正準表現になります). その為, radcan を用いた簡易化も一意に定まります. 但し, radcan は時間を多く消費します. これは因子分解と指数の部分分数展開を基本とした簡易化の為, 式の成分の間の関係を探索する為です.

scsimp 函数

```
scsimp(<式>, <規則1>, ..., <規則n>)
```

scsimp(=Sequential Comparative SIMPlification) は, 式 (その最初の引数), 同一性や規則 (その他の引数) の集合を取って簡易化を試みます. より小さな式が得られると, その処理が繰返されます. そうでなければ, 全ての簡易化が試みられた後に, もとの式が返却されます.

6.6.6 簡易化に関する補助的函数

askinteger 函数

```
askinteger(<式>, <オプション引数>)  
askinteger(<式>)
```

<式> は任意の有効な Maxima の式で, <オプション引数> は even(偶数), odd(奇数), integer(整数) の何れか一つで, 省略された場合は内部で integer が設定されます. この函数は Maxima に蓄えられた情報から <式> が even, odd, 或いは integer であるかを決定しようとします. Maxima に蓄えられた情報では不十分な場合, 利用者に質問して, Maxima に情報を蓄えます.


```
(%i16) aa:1$
(%i17) askinteger(aa);
(%o17)                                     yes
(%i18) askinteger(aa,odd);
(%o18)                                     yes
(%i19) askinteger(aa,even);
(%o19)                                     no
(%i20) askinteger(yy);
Is yy an integer?

yes;
(%o20)                                     yes
(%i21) askinteger(yy,odd);
Is yy an odd number?

no;
(%o21)                                     no
(%i22) askinteger(yy,even);
(%o22)                                     yes
(%i23) askinteger(zz,even);
Is zz an even number?

no;
(%o23)                                     no
(%i24) askinteger(zz,odd);
Is zz an odd number?

no;
(%o24)                                     no
(%i25) askinteger(zz,integer);
Is zz an integer?

yes;
(%o25)                                     yes
(%i26) askinteger(zz+yy+aa,integer);
```

```
(%o26)                                     yes
(%i27) askinteger(zz,integer);
(%o27)                                     yes
(%i28) askinteger(zz,even);
Is zz an even number?

yes;
(%i29) askinteger(zz*2+aa,even);
(%o29)                                     no
```

ここでの例で示す様に yy が integer であると指定すると、それから yy は integer となります。更に、 yy が odd であると宣言すれば、自動的に even になります。但し、 zz が odd でなく、even ではないと宣言しても、askinteger(zz) は no とはならず尋ねて来ます。ここで、yes とすれば、それまで入力した odd でも even でもない事が消去されます。askinteger 関数は $zz+yy+aa$ や $2*zz+aa$ の様な式に対しても、それ以前の入力情報から、整数、奇数か偶数であるかを判断します。

———— asksign 関数 ————

asksign (〈式〉)

asksign 関数は〈式〉が、正、負、或いは零であるかを決定します。この際に、Maxima に蓄えられた情報をもとに決定しようとしませんが、情報が不十分で決定出来なければ、その演繹を完遂する為に必要な質問を利用者に対して行います。この利用者の答は Maxima に記録されます。ここで、asksign が尋ねる値は pos(正值), neg(負値), zero(零) の何れか一つです。

———— numerval 関数 ————

numerval(〈変数₁〉,〈式₁〉, ...,〈変数_n〉,〈式_n〉)

numerval 関数は〈変数_{*i*}〉を〈式_{*i*}〉の数値変数として宣言し、〈式_{*i*}〉は大域変数 numer が true であれば、任意の式に現われる変数に対して評価と代入が行われます。

6.6.7 共通の項で纏める関数

facout 関数

$$\text{facout}(\langle \text{式}_1 \rangle, \langle \text{式}_2 \rangle)$$

facout 関数は第二引数の $\langle \text{式}_2 \rangle$ の各項を第一引数の $\langle \text{式}_1 \rangle$ で割って、全体を第一引数の積で纏めた形式で返す関数です。この関数の性質上、第二引数が単項式であれば、第二引数の式がそのまま返却されます。

```
facout(sin(x),cos(x)+sin(x));

                                cos(x)
(%o42)                        (----- + 1) sin(x)
                                sin(x)

(%i43) facout(sin(x),cos(x));
(%o43)                        cos(x)
```

6.7 リスト

6.7.1 Maxima のリスト

Maxima にはリストと呼ばれるデータ型があります。これは配列に似たデータ構造ですが、より柔軟で視覚的にも把握し易い構造を持っており、多くの数式処理で採用されています。

Maxima のリストは $[1, 2, 7, x+y]$ の様に、各成分をコンマ、で区切った列を大括弧 $[]$ で括ったものです。猶、リストは集合とは違い、列の順序に意味があり、同じ成分で構成されたリストでも列の順序が異なると異ったリストになります。例えば、 $[1, 2, 3], [1, 3, 2], [3, 2, 1]$ は、全て $1, 2, 3$ で構成されたリストですが、成分の並びが全て異なる為、全て異なったリストになります。但し、集合の場合は成分の並び順は無関係で、 $\{2, 3, 1\}, \{1, 3, 2\}$ は $\{1, 2, 3\}$ と等しい集合になります。集合はリストと重なる部分も多い為、必要に応じて §6.8 も参照して下さい。

又、Maxima のリストでは、LISP のリストの様に成分を単純に空行で区切るのではない事に注意して下さい。但し、Maxima のリストでも $[1, 2, [3, 4], [4, [5]]]$ の様にリストを入れ子にしても構いません。

ここで Maxima は LISP で記述されたシステムの為、Maxima の式やプログラムも内部では LISP の S 式と呼ばれるリストで表現されています。更に、Maxima の演算子が前置式でなくても内部では前置表現になっています。その為、Maxima 内部表現では演算子の部分が 0、それ以降の成分に $1, 2, \dots$ と番号が振られた階層構造を持っています。この事から一見してリスト処理専用の函数であっても、Maxima の殆どの式で利用可能な事が多くあります。この Maxima の内部表現に関しては §6.4 を参照して下さい。

6.7.2 リストの生成を行う函数

出力がリストの函数は沢山あります。ここでは雛形となる式を用いて、リストを生成する函数を幾つか紹介しておきましょう。

—— リストを生成する函数 ——

```
create_list(< 式 >, < 変数1 >, < リスト1 >, ..., < 変数n >, < リストn >)
makelist(< 式 >, < 変数 >, < 正整数1 >, < 正整数2 >)
makelist(< 式 >, < 変数 >, < リスト >)
```

create_list 函数は与えられた < 式 > に対し、式中的変数にリストで指定した値を代入した結果で構成された平リストを返す函数です。

```
(%i30) create_list(x^2,x,[1,3,x1,sin(x)]);
                                2      2
(%o30) [1, 9, x1 , sin (x)]
```

create_list 関数は後述の makelist 関数と違い複数の変数を扱う事が可能です. この場合, create_list 関数の引数を左端から順に作用させてゆく形になります.

例えば, create_list($f(x_1, \dots, x_n), x_1, L_1, \dots, x_n, L_n$) は
 flatten(create_list($\dots$ (create_list($f(x_1, \dots, x_n), x_n, L_n$), $\dots, x_1, L_1$))) と同じ値となります.

ここで flatten 関数はリストを平リストにする関数です.

実際に, この事を確認しておきましょう.

```
(%i28) create_list(i^j+sin(k),i,[0,1,2,3,4],j,[2,3],k,[x1,x2,x3]);
(%o28) [sin(x1), sin(x2), sin(x3), sin(x1), sin(x2), sin(x3), sin(x1) + 1,
sin(x2) + 1, sin(x3) + 1, sin(x1) + 1, sin(x2) + 1, sin(x3) + 1, sin(x1) + 4,
sin(x2) + 4, sin(x3) + 4, sin(x1) + 8, sin(x2) + 8, sin(x3) + 8, sin(x1) + 9,
sin(x2) + 9, sin(x3) + 9, sin(x1) + 27, sin(x2) + 27, sin(x3) + 27,
sin(x1) + 16, sin(x2) + 16, sin(x3) + 16, sin(x1) + 64, sin(x2) + 64,
sin(x3) + 64]
(%i29) flatten(create_list(create_list(create_list(i^j+sin(k),k,[x1,x2,x3]),
j,[2,3]),i,[0,1,2,3,4]));
(%o29) [sin(x1), sin(x2), sin(x3), sin(x1), sin(x2), sin(x3), sin(x1) + 1,
sin(x2) + 1, sin(x3) + 1, sin(x1) + 1, sin(x2) + 1, sin(x3) + 1, sin(x1) + 4,
sin(x2) + 4, sin(x3) + 4, sin(x1) + 8, sin(x2) + 8, sin(x3) + 8, sin(x1) + 9,
sin(x2) + 9, sin(x3) + 9, sin(x1) + 27, sin(x2) + 27, sin(x3) + 27,
sin(x1) + 16, sin(x2) + 16, sin(x3) + 16, sin(x1) + 64, sin(x2) + 64,
sin(x3) + 64]
```

makelist 関数は $\langle$ 変数 $\rangle$ で $\langle$ 式 $\rangle$ の変数を指定し, その変数に対して値を代入する事でリストを生成する関数です. 先ず, 引数が 4 個の場合, 第三と第四引数は正整数で, 第二引数で指定した変数の値域はこの第三と第四引数で指定された閉区間に含まれる整数になります. ここで, 第三引数は第四引数以下でなければなりません.

makelist 関数の引数が 3 個の場合, 変数の値域は第三引数のリストになります.

```
(%i5) makelist(x^2+i*x+1=i^2,i,1,4);
```

```
(%o5) [x2 + x + 1 = 1, x2 + 2 x + 1 = 4, x2 + 3 x + 1 = 9, x2 + 4 x + 1 = 16]
```

```
(%i6) makelist(x^2+i*x+1=i^2,i,[1,2,4]);
```

```
(%o6) [x2 + x + 1 = 1, x2 + 2 x + 1 = 4, x2 + 4 x + 1 = 16]
```

6.7.3 リスト処理に関連する大域変数

—— リストに関連する大域変数 ——

変数名	初期値	取り得る値	概要
listarith	true	[true,false]	算術演算子によるリスト評価を制御
inflag	false	[true,false]	内部表現への作用を制御

大域変数 listarith は true であれば算術演算子によるリスト評価を実行します. 大域変数 inflag は true の場合, 成分を取り出す関数は与えられた式の内部表現に対して処理を行います. 簡易化は式の再並び換えを行う事に注意が必要になります. その為, `first(x+y)` は inflag が true であれば `x` となりますが, 大域変数 inflag が false ならば `y` となります.

猶, 大域変数 `inflag` に影響を受ける函数に以下のものがあります.

— `inflag` の影響を受ける函数 —

函数	概要
<code>part</code>	式の成分を取出す函数
<code>substpart</code>	式の成分の代入を行う函数
<code>first</code>	リストの先頭を取出す函数
<code>rest</code>	リストの残りを取出す函数
<code>last</code>	リストの末尾を取出す函数
<code>length</code>	リストの長さを返す函数
<code>for-in</code> 構文	リストを用いるループ文
<code>map</code>	リストに函数を作用させる函数
<code>fullmap</code>	リストに函数を作用させる函数
<code>maplist</code>	リストに函数を作用させる函数
<code>reveal</code>	式の置換えを行う函数
<code>pickapart</code>	成分を取出す函数

6.7.4 リスト処理に関連する主な函数

— リストに関連する真理函数 —

函数	<code>true</code> を返す条件
<code>atom(<変数>)</code>	原子の場合
<code>symbolp(<変数>)</code>	原子であり, 数値でない場合
<code>listp(<変数>)</code>	リストの場合
<code>member(<式₁>, <式₂>)</code>	<式 ₁ > が <式 ₂ > に含まれている場合

<変数> が原子かリストであるかは `atom` 函数と `listp` 函数で調べる事が出来ます. `atom` 函数は引数が原子であれば `true`, それ以外は `false` を返します. `listp` 函数は引数がリストであれば `true`, そうでなければ `false` を返します.

猶, ここでの判定では内部表現は無関係です. その為, `sin(x)` や `x+y` の様な式では各変数に値が束縛されていなければ原子でもリストにもなりません.

`member` 函数は二つの引数を取り, <式₁> が <式₂> に含まれていれば `true`, それ以外は `false` を返します.

```
(%i34) member(sin(x),cos(x)+sin(x)+x^2);
(%o34) true
(%i35) member(sin(x),[cos(x)+sin(x)+x^2]);
(%o35) false
(%i36) member(sin(x),[cos(x),sin(x),x^2]);
(%o36) true
(%i37) member(sin(x),[cos(x),sin(x)+x^2]);
(%o37) false
(%i38) member(sin(x),f(cos(x),sin(x),x^2));
(%o38) true
(%i39) member(sin(x),f(cos(x),sin(x)+x^2));
(%o39) false
```

—— リストの基本処理を行う関数 ——

length($\langle$ リスト $\rangle$)	$\langle$ リスト $\rangle$ の長さを返却
copylist($\langle$ リスト $\rangle$)	$\langle$ リスト $\rangle$ の複製
reverse($\langle$ リスト $\rangle$)	$\langle$ リスト $\rangle$ の並びを逆にしたリストを返却
append($\langle$ リスト ₁ $\rangle$, $\langle$ リスト ₂ $\rangle$, ...)	複数のリストの結合を実行
cons($\langle$ 式 ₁ $\rangle$, $\langle$ 式 ₂ $\rangle$)	$\langle$ 式 ₂ $\rangle$ を $\langle$ 式 ₁ $\rangle$ に追加
endcons($\langle$ 式 $\rangle$, $\langle$ リスト $\rangle$)	$\langle$ 式 $\rangle$ を $\langle$ リスト $\rangle$ の後に結合
delete($\langle$ 式 ₁ $\rangle$, $\langle$ 式 ₂ $\rangle$, $\langle n \rangle$)	$\langle$ 式 ₁ $\rangle$ を $\langle$ 式 ₂ $\rangle$ の先頭から $\langle n \rangle$ 個削除
sort($\langle$ リスト $\rangle$, $\langle$ 述語 $\rangle$)	$\langle$ 述語 $\rangle$ による置換
sort($\langle$ リスト $\rangle$)	順序 $>_m$ を用いてリストの成分の置換を実施

リストの長さは length を用いて調べられます。この length は LISP の同名の関数と同じ動作をしています。但し,Maxima の任意の式は内部表現では LISP のリスト構造を持ち,length 関数はその内部表現で、先頭の式の識別子を除いたリストの長さを返す関数です。

```
(%i22) length([1,2,3]);
(%o22) 3
(%i23) length([1,2,[1,2,[3,4,5]]]);
(%o23) 3
```



```
(%i43) endcons(x+y,[1,2,3,4]);
(%o43)
[1, 2, 3, 4, y + x]
(%i44) endcons(x+y,sin(x)+cos(y));
(%o44)
cos(y) + y + sin(x) + x
(%i45) endcons(x+y,sin(x)/cos(y));
(%o45)
sin(x) (y + x)
-----
cos(y)
(%i46) endcons(x+y,sin(x)*cos(y));
(%o46)
sin(x) (y + x) cos(y)
(%i47) endcons(x+y,sin(x)-cos(y));
(%o47)
- cos(y) + y + sin(x) + x
```

この様に endcons 関数はリストではなく、式に追加する場合はその式の内部表現に依存します。

delete 関数は $\langle \text{式}_2 \rangle$ に含まれる $\langle \text{式}_1 \rangle$ を式の先頭から $\langle n \rangle$ 個削除します。但し、 $\langle \text{式}_1 \rangle$ が関数、或いは単項式でなければ除去出来ません。

猶、 $\langle n \rangle$ はオプションで、指定しない場合と $\langle \text{式}_2 \rangle$ に含まれる $\langle \text{式}_1 \rangle$ が $\langle n \rangle$ 個よりも少ない場合、 $\langle \text{式}_1 \rangle$ を全て削除します。

sort 関数は第一引数のリストの並び換えを行います。引数が一つの場合、各成分を Maxima の順序 $>_m$ を用いて並び換えを行います。ここで、引数が二個の場合、第二引数が 2 変数の真理関数となります。

```
(%i47) sort([1,2,3,4,x,y,z]);
(%o47)          [1, 2, 3, 4, x, y, z]
(%i48) sort([1,2,3,4,x,y,z],greaterp);
(%o48)          [z, y, x, 4, 3, 2, 1]
```

指定した部分式を取出す関数

```
first( $\langle \text{式} \rangle$ )
last( $\langle \text{式} \rangle$ )
rest( $\langle \text{式}_1 \rangle, \langle n \rangle$ )
sublist( $\langle \text{リスト} \rangle, \langle \text{関数} \rangle$ )
substpart( $\langle x \rangle, \langle \text{式} \rangle, \langle n_1 \rangle, \dots, \langle n_k \rangle$ )
```

first 関数は $\langle \text{式} \rangle$ の最初の成分を返します。これに対し、last 関数は $\langle \text{式} \rangle$ の最後の成分を返します。

rest 関数、first 関数と last 関数は、Maxima の大域変数 inflag によって、与式の内部表現に対する操作に変更出来ます。inflag の初期値が false の為、内部表現に対して操作したければ、`inflag:true;` を実行し、inflag の値を true に変更する必要があります。

rest 関数は、 $\langle n \rangle$ が正整数であれば、 $\langle \text{式}_1 \rangle$ の先頭から n 個の成分を除いた式を返します。ここで n が負整数であれば、 $\langle \text{式}_1 \rangle$ の後から $\langle n \rangle$ 個の成分を除いた式を返します。

```
(%i52) rest(x+y+z,2);
(%o52)          x
(%i53) rest(x+y+z,-2);
(%o53)          z
(%i54) rest([x+y+z,sin(x)+cos(x),exp(x)],-2);
(%o54)          [z + y + x]
```

```
(%i55) rest([x+y+z,sin(x)+cos(x),exp(x)],2);
```

```

                                x
(%o55)                        [%e ]

```

sublist 関数は〈真理関数〉が true を返す〈リスト〉に含まれる成分を抽出したリストを返します。

例えば, `sublist([1,2,3,4],evenp);` は `[2,4]` を返します。

substpart 関数は〈式〉の $\langle n_1 \rangle, \dots, \langle n_k \rangle$ で指定した成分を $\langle x \rangle$ で置換えます。〈 x 〉は式, 原子, 演算子が指定可能です。

$\langle n_1 \rangle, \dots, \langle n_k \rangle$ の指定方法は, 〈式〉がリストで, 第 m 番目の成分であれば m を指定します。更に, リストの m 番目がリストで, その n 番目の成分を入れ替えたければ, 列 m, n で指定します。

```
(%i13) substpart(x,[1,2,3,4],2);
```

```
(%o13)                        [1, x, 3, 4]
```

```
(%i14) substpart(x,[1,[2,3],4],2);
```

```
(%o14)                        [1, x, 4]
```

```
(%i15) substpart(x,[1,[2,3],4],2,2);
```

```
(%o15)                        [1, [2, x], 4]
```

最初の例ではリスト `[1,2,3,4]` の第二成分の 2 を x で置換える為, 2 を指定しています。複合リスト `[1,[2,3],4]` の場合, 第二成分はリスト `[2,3]` なので, 第二成分を x で置換すれば `[1,x,4]` になります。第二成分に含まれる 3 を x で置換えたければ, 第二成分のリストの第二成分を指定すれば良い事になります。

この事は Maxima の一般の式に対しても適応が可能です。但し, Maxima の場合は入力した式の順番と, Maxima に入力された後の順番が異なる事がある為, 注意が必要になります。

```
(%i16) expr:(x+1)/(x^2+x+1)+exp(x);
```

```

                                x      x + 1
(%o16)                        %e  +  -----
                                2
                                x  + x + 1

```

```
(%i17) substpart(sin(x),expr,1);
```

```
(%o17)
              x + 1
sin(x) + -----
              2
          x  + x + 1
```

```
(%i18) substpart(sin(x),expr,2);
```

```
(%o18)
              x
sin(x) + %e
```

```
(%i19) substpart(sin(x),expr,2,2);
```

```
(%o19)
          x + 1      x
----- + %e
          sin(x)
```

```
(%i20) substpart(sin(x),expr,2,1);
```

```
(%o20)
          sin(x)      x
----- + %e
          2
          x  + x + 1
```

```
(%i21) substpart("+",expr,2,0);
```

```
(%o21)
          x      2
%e  + x  + 2 x + 2
```

この例では式 `expr` の指定した成分を `sin(x)` で置換える操作を行っています。ここで第一成分は入力した順序とは異なって `%e^x` となっている事に注意して下さい。次に、第二成分は有理式全体となりますが、この第二成分の構造は (割算, `x+1`, `x^2+x+1`) となっています。Maxima の内部表現では式はリストで表現され、演算子が先頭の第0成分となり、以下にその引数が続く構造となっています。その為、2,1 で有理式の分子、2,2 で有理式の分母、最後の 2,0 が演算子となります。そこで、`substpart("+",expr,2,0)` を実行すると割算が和に置換えられてしまい、有理式が `x^2+2*x+2` で置換えられてしまいます。

6.7.5 map 函数族

map 函数は LISP ではお馴染の函数で, 基本的に函数をリストに作用させるものです. これは Mathematica や Maple でも採用されており, 非常に便利な函数です. Maxima では, 式は内部的にリスト構造を持っていますが, 表にはそれが現われていない為, map 函数の動作が判り難い側面もあります.

ここでは, 内部形式と絡めて, map 函数のお仲間について解説します.

Maxima の map 函数のお仲間は全て Maxima の式に函数を式やリスト, 或いは行列等に作用させる働きがあります. この map 函数の基本的な考えは, 函数をリストの各成分に作用させたリストを計算させるものです. ところが Maxima の式は内部的に全てリスト (S 式) である為, この map 函数のお仲間による式への作用は非常に有効な手段です. そこで, 作用させる函数の特性, 作用させる対象の構造といった諸条件に対応する為に, 様々な map 函数のお仲間が Maxima にはあります.

さて, ここで最初に map 函数の例を示しておきましょう.

```
(%i34) map(sin,x*y);
(%o34)                sin(x) sin(y)
(%i35) map(sin,x*y+y);
(%o35)                sin(x y) + sin(y)
(%i36) map(sin,factor(x*y+y));
(%o36)                sin(x + 1) sin(y)
(%i37) map(lambda([x,y],x*y),x+y,w+z);
(%o37)                y z + w x
```

最初の $x*y$ の場合, 主演算子は $*$, 第一層には被演算子の x と y がある為, $\sin$ は式 $x*y$ の第一層の部分式 x と y に作用しますが, 演算子の置換を行わない為に $\sin(x)+\sin(y)$ が得られます. 式 $x*y+y$ の場合, この式の第一層には二つの部分式 $x*y$ と w があるので今度は $\sin(x*y)+\sin(y)$ となります.

ところが同値な式でも内部表現が異なると異った結果になります. 次の例では $\text{factor}(x*y+y)$ の結果に map 函数で $\sin$ を作用させていますが, $\text{factor}(x*y+y)$ が $(x+1)*y$ となる為, この式の第一層には部分式 $x+1$ と y があり, 主演算子が $*$ なので, $\sin(x+1)*\sin(y)$ が得られます.

次の maplist 函数は基本的に map 函数と同様ですが, こちらは主演算子をリストに置換します.

```
(%i15) map(sin,factor(x*y+y));
(%o15)          sin(x + 1) sin(y)
(%i16) maplist(sin,factor(x*y+y));
(%o16)          [sin(x + 1), sin(y)]
(%i17) :lisp %o15;
((MTIMES SIMP) ((%SIN SIMP) ((MPLUS SIMP) 1 X)) ((%SIN SIMP) Y))
(%i17) :lisp %o16;
((MLIST SIMP) ((%SIN SIMP) ((MPLUS SIMP) 1 X)) ((%SIN SIMP) Y))
```

上の例で示す様に、内部表現で MTIMES が MLIST の変化している事に注目して下さい。

6.7.6 map 関数族に関連する大域変数

大域変数 maperror

変数名	初期値	概要
maperror	true	map や maplist 関数を制御します

大域変数 maperror は map 関数と maplist 関数の動作に影響します。先ず、map 関数と maplist 関数の引数は $\langle \text{関数} \rangle, \langle \text{式}_1 \rangle, \dots, \langle \text{式}_n \rangle$ で、 $\langle \text{関数} \rangle$ は n 変数の関数です。ここで、大域変数 maperror の値が true の場合、各 $\langle \text{式}_i \rangle$ の主演算子は基本的に同じもので、成分の個数も同じ個数でなければなりません。大域変数 maperror が false の場合、それ以外の引数でも適用可能になり、以下の動作となります。

1. 全ての $\langle \text{式}_i \rangle$ が同じ長さでなければ、最短の $\langle \text{式}_j \rangle$ を終えた時点で停止します。
2. $\langle \text{式}_i \rangle$ の主演算子が全て同じものでなければ、 $[\langle \text{式}_1 \rangle, \dots, \langle \text{式}_n \rangle]$ に $\langle \text{関数} \rangle$ を作用させ、apply と同じ動作になります。

大域変数 maperror が true の場合、上の二つの状況であればエラーメッセージが出力されます。

```
(%i40) maperror:false;
(%o40)          false
(%i41) map(lambda([x,y],x*y),x+y+a,w+z);
‘map’ is truncating.
```

```
(%o41)
y z + w x
(%i42) map(lambda([x,y],x*y),x+y+a,w*z);
'map' is doing an 'apply'.
(%o42)
w (y + x + a) z
(%i43) maperror:true;
(%o43)
true
(%i44) map(lambda([x,y],x*y),x+y+a,w*z);
Arguments to 'mapl' not uniform - cannot map.
-- an error. Quitting. To debug this try debugmode(true);
```

6.7.7 map 関数いろいろ

map 関数に関連する真理関数

```
mapatom(< 式 >)
```

mapatom 関数は、< 式 > が map 関数によって原子として扱われる時、true を返します

map 関数

```
map(< 関数 >, < 式1 >, ..., < 式n >)
maplist(< 関数 >, < 式1 >, < 式2 >, ...)
```

map 関数は n 個の式の列 $\langle \text{式}_1 \rangle, \dots, \langle \text{式}_n \rangle$ から n 個の成分を取出し、 n 個の引数を取る $\langle \text{関数} \rangle$ を作用させた結果を返します。

maplist 関数は map 関数に似ています。 $\langle \text{式}_i \rangle$ の各成分に $\langle \text{関数} \rangle$ を作用させたリストを生成します。 $\langle \text{関数} \rangle$ は関数の名前や lambda 式となります。

maplist が map 関数と異なる点は、map の場合、 $\langle \text{式}_i \rangle$ の主演算子で式を纏めたものが出力されるのに対して、maplist では、主演算子の代りに Maxima のリスト演算子が置かれる事です。

```
(%i27) maplist(sin,x+y);
E(%o27)
[sin(y), sin(x)]
(%i28) map(sin,x+y);

(%o28)
sin(y) + sin(x)
(%i29) maplist(lambda([x,y],x*y),x+y,w+z);
```

```
(%o29) [y z, w x]
(%i30) map(lambda([x,y],x*y),x+y,w+z);
(%o302) y z + w x
```

———— scanmap 関数 ————

```
scanmap(< 関数 >, < 式 >)
scanmap(< 関数 >, < 式 >, bottomup)
```

scanmap(< 関数 >, < 式 >) の場合, 再帰的に < 関数 > を < 式 > の内部表現の頂上から適用して行きます. これは徹底した因子分解が望ましい時には特に便利です.

例えば, $(a^2 + 2a + 1)y + x^2$ を factor で因子分解しても, そのまま元の式が返却されるのですが, scanmap 関数を使って factor 関数を式に作用させると, 与式の部分式 $a^2 + 2a + 1$ が因子分解された結果が返されます.

```
(%i3) exp:(a^2+2*a+1)*y + x^2
(%i4) factor(exp);

          2          2
(%o4)      a y + 2 a y + y + x
(%i5) scanmap(factor,exp);

          2      2
(%o5)      (a + 1) y + x
```

scanmap 関数による結果は式の内部表現に依存します. 上の例の式の内部表現を以下に示します.

```
((MPLUS SIMP)
  ((MEXPT SIMP) X 2)
  ((MTIMES SIMP)
    ((MPLUS SIMP) 1 ((MTIMES SIMP) 2 A)
      ((MEXPT SIMP) A 2)) Y))
```

この内部表現に対して, scanmap 関数は factor 関数を上の階層から各部分式に対して作用させます. この式の場合, 最初に x^2 と $(a^2 + 2a + 1)y$ に factor 関数を作用させ, その次に x^2 の x と 2, $a^2 + 2a + 1$ と y 等々と下の階層の成分に作用します. その結果, $a^2 + 2a + 1 (= (+ 1 (* 2 a) (^ a 2)))$ の展開以外はそのままとする為, 上記の結果を得ています. この作用の様子は factor 関数の代りに 'f を与えると判り易くなります.


```
(%i18) scanmap('f,exp);
              f(2)                                f(2)
(%o18) (f(f(f(f(a)      ) + f(f(2) f(a)) + f(1)) f(y)) + f(f(x)      ))
```

この性質がある為に式を全て展開してしまうと、各項に factor 関数に作用させる為に、入力値がそのまま返却されてしまいます。

```
(%i16) expand(exp);
              2              2
(%o16)      a y + 2 a y + y + x
(%i17) scanmap(factor,expand(exp));
              2              2
(%o18)      a y + 2 a y + y + x
```

scanmap(⟨ 関数 ⟩,⟨ 式 ⟩,bottomup) は scanmap(⟨ 関数 ⟩,⟨ 式 ⟩) とは逆に、内部表現の最下層から ⟨ 関数 ⟩ を作用させます。

— fullmap 関数と fullmapl 関数 —

```
fullmap(⟨ 関数 ⟩,⟨ 式1 ⟩,⋯,⟨ 式n ⟩,⋯)
fullmapl(⟨ 関数式 ⟩,⟨ 式1 ⟩,⋯,⟨ 式n ⟩,⋯)
```

fullmap 関数は ⟨ 関数 ⟩ を第二引数以降の式に対して作用させる関数です。

```
(%i38) fullmap(sin,[1,2,3]);
(%o38)      [sin(1), sin(2), sin(3)]
(%i39) fullmap(lambda([x,y],x-y),[1,2,3],[3,2,1]);
(%o39)      [- 2, 0, 2]
(%i40) fullmap(sin,a+b+c*d);
(%o40)      sin(c) sin(d) + sin(b) + sin(a)
```

fullmapl 関数は fullmap 関数と似た働きをしますが、fullmap 関数の第二引数以降は全てリスト、或いは行列である点で異なります。

```
(%i41) fullmapl(sin,[1,2,3]);
(%o41)      [sin(1), sin(2), sin(3)]
(%i42) fullmapl(lambda([x,y],x-y),[1,2,3],[3,2,1]);
(%o42)      [- 2, 0, 2]
(%i43) fullmapl(sin,[a+b+c*d]);
```

```
(%o43) [sin(c d + b + a)]
(%i44) fullmap1(sin, [[a],[b],[c*d]]);
(%o44) [[sin(a)], [sin(b)], [sin(c d)]]
```

———— outermap 関数 ————

outermap($\langle$ 関数 $\rangle$, $\langle$ 式₁ $\rangle$, $\dots$, $\langle$ 式_n $\rangle$)

outermap 関数は与えられた $\langle$ 関数 $\rangle$ を $\langle$ 式₁ $\rangle, \dots, \langle$ 式_n $\rangle$ 分配する事で生成した各成分の長さが n , 成分の総数が $\text{length}(\langle$ 式₁ $\rangle) \cdot \text{length}(\langle$ 式₂ $\rangle) \cdots \text{length}(\langle$ 式_n $\rangle)$ 個のリストに作用させる関数です.

```
(%i1) outermap(f, [1,2,3], [4,5], [a,b,c,d,e]);
(%o1) [[[f(1, 4, a), f(1, 4, b), f(1, 4, c), f(1, 4, d), f(1, 4, e)],
[f(1, 5, a), f(1, 5, b), f(1, 5, c), f(1, 5, d), f(1, 5, e)]],
[[f(2, 4, a), f(2, 4, b), f(2, 4, c), f(2, 4, d), f(2, 4, e)],
[f(2, 5, a), f(2, 5, b), f(2, 5, c), f(2, 5, d), f(2, 5, e)]],
[[f(3, 4, a), f(3, 4, b), f(3, 4, c), f(3, 4, d), f(3, 4, e)],
[f(3, 5, a), f(3, 5, b), f(3, 5, c), f(3, 5, d), f(3, 5, e)]]]
```

6.8 集合について

6.8.1 概要

Maxima での集合は、原子、式、リストや集合等の成分を中括弧 ($\{\}$) で括った対象です。この中括弧 $\{\}$ は matchfix 型演算子として宣言されており、内部では第一成分に ($\$SET SIMP$) が置かれたリストとして表現されています。

集合の生成は、単純に成分を中括弧で括ったものを直接入力して生成するか、setify 関数や fullsetify 関数を用いてリストから生成します。

Maxima での集合は orderlessp 関数を用いた順序で並べ替えたものになります。その為、集合の各成分を orderlessp 関数で比較した場合、集合で左側のものが右側のものよりも小さくなります。この順序は第 5.2 節の Maxima の順序 $>_m$ です。

集合を生成する際に、各成分の評価が行われ、同値なものが存在すれば、代表のみが選択され、Maxima の順序 $>_m$ に従って成分の並び換えが行われます。但し、論理式は評価が行われません。

```
(%i117) {1,2,3,4,5,4/2,cos(0)};
(%o117) {1, 2, 3, 4, 5}

(%i118) a:1;b:sin(%pi/4)$
(%i120) {1,{2},{a}},b,{a,{b,a*b}}};
(%o120) {1, -----, {1, {-----, {-----}}}}, {2}, {{1}}
          sqrt(2)      sqrt(2)      sqrt(2)

(%i121) S1:{1>3,3<54,5=5};
(%o121) {1 > 3, 3 < 54, 5 = 5}

(%i122) S2:setify([1>3,3<54,5=5]);
(%o122) {1 > 3, 3 < 54, 5 = 5}

(%i123) S3:map(lambda([x],ev(x,pred)),{1>3,3<54,5=5});
(%o123) {false, true}
```

集合はリストに似ている為、リストの様な処理が行えそうに見えますが、リストで出来る操作が集合に対して行えるとは限りません。

以下に利用可能な関数名を列記しておきますが、一応使えなくもない程度で、将来の事も考えると、希望する関数が存在しない場合には、集合を listify 関数や full.listify 関数で一旦リストに変換して、リスト処理の関数を用いる方が無難であると思います。

- 集合に対して適用可能な関数
length, member, append, cons, delete, endcons, first, rest, last, map, maplist, scanmap
- 無意味な関数
reverse, substpart

6.8.2 集合の生成に関連する関数

集合の生成は、その成分で構成された列を演算子 $\{\}$ で括る事でも生成出来ませんが、定型的な式で構築可能な場合には、その式を用いた方法で、集合やリストを生成したり、既存のリストから集合を生成する方法があります。

—— 集合やリストを生成する関数 ——

```
makeset(⟨ 式 ⟩, ⟨ リスト1 ⟩, ⟨ リスト2 ⟩)
cartesian_product(⟨ 集合1 ⟩, ..., ⟨ 集合n ⟩)
divisors(⟨ 正整数 ⟩)
```

makeset 関数は $\langle \text{式} \rangle$ から集合を生成する関数です。先ず、第二引数の $\langle \text{リスト}_1 \rangle$ は $\langle \text{式} \rangle$ に含まれる変数で構成されたリストで、ここで指定した各変数の値域が $\langle \text{リスト}_2 \rangle$ で指定するリストになります。猶、 $\langle \text{リスト}_2 \rangle$ の各成分は $\langle \text{リスト}_1 \rangle$ で指定した変数と対応が付くものでなければなりません。

```
(%i15) makeset(i+j*x, [i, j], [[1, 2], [3, 1], [4, 3]]);
(%o15) {x + 3, 2 x + 1, 3 x + 4}
(%i16) makeset(i+j*x, [x, j], [[1, 2], [3, 1], [4, 3]]);
(%o16) {i + 2, i + 3, i + 12}
(%i17) makeset(i+j*x, [i], [[1], [3], [4]]);
(%o17) {j x + 1, j x + 3, j x + 4}
(%i18) makeset(i+j*x, [i], [[1]]);
(%o18) {j x + 1}
```

cartesian_product 関数は直積集合を生成する関数です。集合を $S_1, \dots, S_n$ とすると、cartesian_products はリスト $[s_1, \dots, s_n] \in S_1 \times \dots \times S_n$ で構成された集合を返します。

猶、集合の成分には順序がありませんが、直積集合の各成分はリストである為に、各成分には順序があります。

```
(%i26) cartesian_product({1,2,3},{4,5,6});
(%o26) {[1, 4], [1, 5], [1, 6], [2, 4], [2, 5], [2, 6], [3, 4], [3, 5], [3, 6]}
(%i27) cartesian_product({1,2,3},{4,5});
(%o27) {[1, 4], [1, 5], [2, 4], [2, 5], [3, 4], [3, 5]}
```

divisors 関数は与えられた整数を割切る正整数 (因子) で構成されたリストを返す関数です. 負の整数が与えられた場合, 内部で絶対値を取って処理を行います. 猶, この関数の内部では factor 関数が用いられています.

```
(%i36) divisors(-290348);
(%o36) {1, 2, 4, 29, 58, 116, 2503, 5006, 10012, 72587, 145174, 290348}
(%i37) divisors(2903488432);
(%o37) {1, 2, 4, 8, 13, 16, 26, 52, 104, 208, 13959079, 27918158, 55836316,
111672632, 181468027, 223345264, 362936054, 725872108, 1451744216, 2903488432}
```

—— リストから集合の生成を行う関数 ——

```
setify(< リスト >)
fullsetify(< リスト >)
```

setify 関数は与えられたリストから集合を生成する関数です.

```
(%i18) a:setify([ 1,2,3,4,5]);
(%o18) {1, 2, 3, 4, 5}
(%i19) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
```

```
MAXIMA> $a
(($SET SIMP) 1 2 3 4 5)
MAXIMA> (to-maxima)
Returning to Maxima
(%o19) true
(%i20) b:setify([ 10,x,3,104,-5]);
(%o20) {- 5, 3, 10, 104, x}
```

又, 生成する際に, LISP の sort 関数を用いて成分が再配置されていますが, その際の順序として orderlessp が用いられます. その結果, orderlessp 関数で最小ものが左端に置かれ, 以降右に行くに従って大きくなります.

猶, setify 命令では最上部が集合になるだけで, 各成分は以前の型をそのまま保ちます. その為, 全てを集合にする為には, fullsetify 命令を用います.

```
(%i24) setify([[1,2],3,[4],5]);
(%o24)          {3, 5, [1, 2], [4]}
(%i25) fullsetify([[1,2],3,[4],5]);
(%o25)          {3, 5, {1, 2}, {4}}
```

—— 集合からリストを生成する函数 ——

```
listify(< 集合 >)
full.listify(< 集合 >)
```

setify 函数と fullsetify 函数の逆の操作を行う函数が listify 函数と fulllistify 函数です.

```
(%i28) listify(b1);
(%o28)          [3, 5, {1, 2}, {4}]
(%i29) listify(a1);
(%o29)          [3, 5, [1, 2], [4]]
```

—— リスト操作の函数 ——

```
flatten(< リスト >)
unique(< リスト >)
```

flatten 函数は与えられた Maxima の複合リストを Maxima の平リストに変換する函数です. 但し, Maxima のリストに対してのみ効果があり, 集合や式は無変換で, そのままの式を返します.

```
(%i6) flatten([1,2,3,[4,[5,4],4]]);
(%o6)          [1, 2, 3, 4, 5, 4, 4]
(%i7) flatten({1,2,3,[4,[5,4],4]});
(%o7)          {1, 2, 3, [4, [5, 4], 4]}
```

unique 函数はリストの各成分に対して, orderlessp 函数を述語として, 順序 $>_m$ で成分の並び換えを実行する函数です. 左から右への順序が, $>_m$ の小から大の順に対応します.

但し, unique 函数はリストの第一層の並び換えを行い, より深い階層に作用はしません.

```
(%i3) unique([1,2,34]);
(%o3)          [1, 2, 34]
(%i4) unique([2,34,1]);
(%o4)          [1, 2, 34]
(%i5) unique([2,34,[3,1]]);
(%o5)          [2, 34, [3, 1]]
```

6.8.3 集合演算の関数

集合演算の関数

演算子	構文
$\cup$	<code>union(<集合₁>, ..., <集合_n>)</code>
$\cap$	<code>intersection(<集合₁>, ..., <集合_n>)</code>
$\cap$	<code>intersect(<集合₁>, ..., <集合_n>)</code>
$\ominus$	<code>symmdifference(<集合₁>, ..., <集合_n>)</code>
$\setminus$	<code>setdifference(<集合₁>, <集合₂>)</code>

`union` 関数は引数の集合全ての和集合を返す関数です.

```
(%i21) union({1,2,3},{3,4,5});
(%o21)          {1, 2, 3, 4, 5}
(%i22) union({{1,2,3}},{3,4,5});
(%o22)          {3, 4, 5, {1, 2, 3}}
(%i23) union({1,2,3},{3,4,5},{a,b,c});
(%o23)          {1, 2, 3, 4, 5, a, b, c}
```

`intersection` は一つ以上の集合を引数にし、与えられた集合の共通集合を返します. 猶、`intersect` 関数は `intersection` 関数の別名といっても良い程で、`intersect` 関数の内部では `apply` 関数を用いて `intersection` 関数を作作用させているだけです.

```
(%i26) intersection({1,2,3,4},{4,5,6});
(%o26)          {4}
(%i27) intersection({1,2,{3,4}},{4,5,6});
(%o27)          {}
```

setdifference 関数は第一引数の集合から第二引数の集合を除去した集合を返します. ここで第一引数の集合が空集合 $\{\}$ ($= \emptyset$) であれば, 結果は空集合 $\{\}$ になります. 猶, 引数が集合でない場合にはエラーを返します.

```
(%i106) setdifference({1,2,3},{3,4});
(%o106) {1, 2}
(%i107) setdifference({1,2,3},{});
(%o107) {1, 2, 3}
(%i108) setdifference({}, {3,4});
(%o108) {}
(%i109) setdifference({1,2,3},{1});
(%o109) {1, 2, 3}
```

symmmdifference 関数は与えられた複数の集合の共通部分を削除した集合を返却します.

```
(%i87) S_1 : {a, b, c};
(%o87) {a, b, c}
(%i88) S_2 : {1, b, c};
(%o88) {1, b, c}
(%i89) S_3 : {a, b, z};
(%o89) {a, b, z}
```

6.8.4 集合に関連する関数

—— 集合の濃度/基数を返す関数 ——

cardinality(⟨ 集合 ⟩)

cardinality 関数は与えられた集合の基数/濃度を返す関数です.

```
(%i58) cardinality({1,2,3,4});
(%o58) 4
(%i59) cardinality({},{});
(%o59) 2
(%i60) cardinality({1,2,{3,4}});
(%o60) 3
```


—— 集合やリストに含まれる元の最大値や最小値を返す函数 ——

```
lmax(< リスト >)
lmax(< 集合 >)
lmix(< リスト >)
lmix(< 集合 >)
```

`lmax` 函数と `lmin` 函数は引数として集合やリストを取り, その中で最大値, 或いは最小値を返す函数です. これらの函数の実体は, `lmax` 函数が `max` 函数, `lmin` 函数が `min` 函数で, 単純にリストの演算子 `[]` や集合の演算子 `{}` を外した数列を `max` 函数や `min` 函数に引き渡すだけです.

```
(%i38) lmax({1,-2,3,-4});
(%o38)                                     3
(%i39) lmin({1,-2,3,-4});
(%o39)                                    - 4
(%i40) lmin({{1,-2},{3},-4});
(%o40)                                min(- 4, {- 2, 1}, {3})
(%i41) lmax({{1,-2},{3},-4});
(%o41)                                max(- 4, {- 2, 1}, {3})
```

—— reduce 函数族 ——

```
lreduce(< 函数 >, < 式 >)
lreduce(< 函数 >, < 式1 >, < 式0 >)
rreduce(< 函数 >, < 式 >)
rreduce(< 函数 >, < 式1 >, < 式0 >)
xreduce(< 函数 >, < 式 >)
xreduce(< 函数 >, < 式1 >, < 式0 >)
tree_reduce(< 函数 >, < 式 >)
tree_reduce(< 函数 >, < 式1 >, < 式0 >)
```

`reduce` 函数族は基本的に第一引数の函数を第二引数のリストや集合に作用させる函数ですが, この作用のさせ方の違いから, 4 種類の函数になっています.

`lreduce` 函数は `lreduce(F, L)` の時, $F(\cdots, F(F(l_1, l_2), l_3), \cdots, l_n)$ を返し, `lreduce(F, L, X)` に対しては $F(\cdots, F(F(X, l_1), l_2), \cdots, l_n)$ を返す函数です.

```
(%i42) lreduce(f, [x_1,x_2,x_3,x_4]);
```

```
(%o42)          f(f(f(x_1, x_2), x_3), x_4)
(%i43) lreduce(f, [x_1,x_2,x_3,x_4],X);
(%o44)          f(f(f(X, x_1), x_2), x_3), x_4)
```

rreduce 関数は $\text{rreduce}(F, L)$ の時、 $F(l_1, \dots, F(l_{n-2}, F(l_{n-1}, l_n)))$ を返し、 $\text{rreduce}(F, L, X)$ に対しては $F(l_1, \dots, F(l_{n-1}, F(l_n, X)))$ を返す関数です。

```
(%i44) rreduce(f, [x_1,x_2,x_3,x_4]);
(%o44)          f(x_1, f(x_2, f(x_3, x_4)))
(%i45) rreduce(f, [x_1,x_2,x_3,x_4],X);
(%o45)          f(x_1, f(x_2, f(x_3, f(x_4, X))))
```

xreduce 関数は第一引数の演算子が nary 型である場合を除いて、lreduce 関数と全く同じです。

F を nary 型の演算子する時、 $\text{xreduce}(F, L)$ は $F((l_1, l_2, l_3, \dots, l_n)$ を返し、 $\text{xreduce}(F, L, X)$ に対しては $F(X, l_1, l_2, \dots, l_n)$ を返します。

```
(%i66) xreduce(g, [x_1,x_2,x_3,x_4,x_5]);
(%o66)          g(g(g(g(x_1, x_2), x_3), x_4), x_5)
(%i67) xreduce(g, [x_1,x_2,x_3,x_4,x_5],X);
(%o67)          g(g(g(g(X, x_1), x_2), x_3), x_4), x_5)
(%i68) declare(g,nary);
(%o68)          done
(%i69) xreduce(g, [x_1,x_2,x_3,x_4,x_5]);
(%o69)          f(x_1, x_2, x_3, x_4, x_5)
(%i70) xreduce(g, [x_1,x_2,x_3,x_4,x_5],X);
(%o70)          f(X, x_1, x_2, x_3, x_4, x_5)
```

tree_reduce 関数は $\text{tree\_reduce}(F, L)$ の時、 $F(F(\dots, F(F(l_1, l_2), F(l_3, l_4)), \dots), l_n)]$ を返し、 $\text{tree\_reduce}(F, L, X)$ に対しては、 $F(F(\dots, F(F(X, l_1), F(l_2, l_3)), \dots), l_n)]$ を返す関数です。

```
(%i49) tree_reduce(f, [x_1,x_2,x_3,x_4,x_5]);
(%o49)          f(f(f(x_1, x_2), f(x_3, x_4)), x_5)
(%i50) tree_reduce(f, [x_1,x_2,x_3,x_4,x_5],X);
(%o50)          f(f(f(X, x_1), f(x_2, x_3)), f(x_4, x_5))
```

6.8.5 集合操作の関数

集合操作の関数

```
adjoin(< 対象 >, < 集合 >)
disjoin(< 対象 >, < 集合 >)
permutation(< 集合 >)
powerset(< 集合 >, < 正整数値 >)
random_permutation(< 集合 >)
```

adjoin 関数は第一引数の対象を第二引数の集合の成分とした集合を返す関数です。

```
(%i15) adjoin(3,{1,4,5});
(%o15)          {1, 3, 4, 5}
(%i16) adjoin({1,2,3},{a,b});
(%o16)          {{1, 2, 3}, a, b}
```

disjoin 関数は第一引数の対象を第二引数の集合から除去した集合を返す関数です。

```
(%i18) disjoin(a,{a,b});
(%o18)          {b}
(%i19) disjoin({a,b},{a,b});
(%o19)          {a, b}
(%i20) disjoin({a,b},{1,2,{a,b}});
(%o20)          {1, 2}
```

permutations 関数は引数の集合の第一層の成分に全ての置換を作用させて得られたリストから構成される集合を返します。

powerset 関数は与えられた集合の全ての部分集合で構成された集合を返します。又、第二引数に正整数値を指定する事で、正整数値で指定された基数を持つ部分集合の集合を返します。

```
(%i78) powerset({1,2,3,4,5});
(%o78) {{}, {1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5},
{1, 2, 3, 5}, {1, 2, 4}, {1, 2, 4, 5}, {1, 2, 5}, {1, 3}, {1, 3, 4},
{1, 3, 4, 5}, {1, 3, 5}, {1, 4}, {1, 4, 5}, {1, 5}, {2}, {2, 3}, {2, 3, 4},
{2, 3, 4, 5}, {2, 3, 5}, {2, 4}, {2, 4, 5}, {2, 5}, {3}, {3, 4}, {3, 4, 5},
{3, 5}, {4}, {4, 5}, {5}}
```

```
(%i79) powerset({1,2,3,4,5},3);
(%o79) {{1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 4}, {1, 3, 5}, {1, 4, 5},
        {2, 3, 4}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5}}
(%i80)
```

random_permutation 関数は乱数を用いて集合やリストの成分の置換を行ったものを返す関数です。置換は集合の第一層に対してのみ実行され、集合の各成分毎に実行しません。

```
(%i31) random_permutation({1,2,3,4,5});
(%o31) [3, 4, 5, 1, 2]
(%i32) random_permutation({1,2,3,4,5});
(%o32) [5, 1, 2, 3, 4]
(%i33) random_permutation({1,2,3,4,5});
(%o33) [4, 5, 1, 2, 3]
(%i34) random_permutation({1,2,3,{4,5}});
(%o34) [1, 2, {4, 5}, 3]
(%i35) random_permutation({1,2,3,{4,5}});
(%o35) [1, 2, 3, {4, 5}]
```

—— 判別に関数を利用する関数 ——

```
external_subset(< 集合 >, < 関数 >, max)
external_subset(< 集合 >, < 関数 >, min)
equiv_classes(< 集合 >, < 2 変数の述語 >)
subset(< 集合 >, < 述語 >)
partition_set(< 集合 >, < 述語 >)
```

external_subset 関数は、第一引数の集合に第二引数の関数を作動させ、第三引数が max の場合は最大値を取る成分、min の場合は最小値を取る成分で各々構成される集合を返します。

```
(%i104) extremal_subset({1,2,3,4,5},lambda([x],abs(x-3)),max);
(%o104) {1, 5}
(%i105) extremal_subset({1,2,3,4,5},sin,min);
(%o105) {5}
```

`equiv_classes` は与えられた集合を判別函数を用いて, 同値なものの集合に分類する函数です. 即ち, 判別函数による同値類の集合を返す函数です.

```
(%i115) equiv_classes({1,2,3,4,5},lambda([x,y],abs(x-3)=abs(y-3)));
(%o115)                {{1, 5}, {2, 4}, {3}}
```

`subset` 函数は与えられた集合から, 第二引数で指定した判別函数を満たす成分で構成される集合を返します.

```
(%i67) subset({1,2,3,4,5},lambda([x],is(x>3)));
(%o67)                {4, 5}
(%i68) subset({1,2,3,4,5},lambda([x],ev(x>3,pred)));
(%o68)                {4, 5}
```

論理式を評価する場合は, `is` 函数, 或いは, `ev` 函数に `pred` を引数に追加した函数も使えます.

`partition_set` は与えられた集合を指定の述語を満たす成分の集合と述語を満たさない成分の集合の二つで構成される集合を返します.

```
(%i70) partition_set({1,2,3,4,5},lambda([x],ev(x>3,pred)));
(%o70)                [{1, 2, 3}, {4, 5}]
(%i71) partition_set({1,2,3,4,5},lambda([x],ev(x<3,pred)));
(%o71)                [{3, 4, 5}, {1, 2}]
(%i72) partition_set({1,2,3,4,5},lambda([x],ev(x<8,pred)));
(%o72)                [{}, {1, 2, 3, 4, 5}]
(%i73) partition_set({1,2,3,4,5},lambda([x],ev(x>8,pred)));
(%o73)                [{1, 2, 3, 4, 5}, {}]
```

この場合, 集合の第一成分が評価函数を満たさない元, 即ち, 返される値が `false` の場合や `unknown` になる集合で, 第二成分が評価函数を満たす元の集合になります.

分割に関連する関数

```

set_partitions(< 集合 >)
set_partitions(< 集合 >, < 正整数値 >)
integer_partitions(< 正整数値 >)
integer_partitions(< 正整数値 >, < 正整数値 >)
num_partitions(< 正整数値 >)
num_partitions(< 正整数値 >, list)
num_distinct_partitions(< 正整数値 >)
num_distinct_partitions(< 正整数値 >, list)

```

set_partition 関数は与えられた集合の分割を返す関数です。第二引数を指定しない場合、全ての分割を返します。

integer_partition は指定した正整数値に対し、和がその整数値になる様なリストを成分とする集合を返します。第二引数を指定した場合、リストの長さが第二引数となるものの集合を返しますが、第二引数の値が第一引数よりも大きな場合、第二引数を指定しない場合の結果で、各成分が第二引数の値の長さになる様に、0 を追加したリストの集合を返します。

```

(%i64) integer_partitions(5);
(%o64) {[1, 1, 1, 1, 1], [2, 1, 1, 1], [2, 2, 1],
        [3, 1, 1], [3, 2], [4, 1], [5]}

(%i65) integer_partitions(5,1);
(%o65) {[5]}

(%i66) integer_partitions(5,3);
(%o66) {[2, 2, 1], [3, 1, 1], [3, 2, 0], [4, 1, 0],
        [5, 0, 0]}

(%i67) integer_partitions(5,5);
(%o67) {[1, 1, 1, 1, 1], [2, 1, 1, 1, 0], [2, 2, 1, 0, 0],
        [3, 1, 1, 0, 0], [3, 2, 0, 0, 0], [4, 1, 0, 0, 0],
        [5, 0, 0, 0, 0]}

```

num_partitions 関数は同一引数の integer_partitions 関数による結果の集合の基数を返す関数です。第二引数に list を指定した場合、1 から第一引数までの整数の分割に対応する分割数を返します。猶、この時に返却されるリストの第一成分は 1 で、実際の分割数は第二成分以降になります。

```
(%i88) is(num_partitions(5)=cardinality(integer_partitions(5)));
(%o88) true
(%i89) num_partitions(7,list);
(%o89) [1, 1, 2, 3, 5, 7, 11, 15]
(%i90) map(lambda([x],cardinality(integer_partitions(x))),[1,2,3,4,5,6,7]);
(%o90) [1, 2, 3, 5, 7, 11, 15]
```

num_distinct_partitions 関数は第一引数で指定された正整数に対応する分割で、各成分が異なる分割の個数を返します。又、第二引数として list を指定すると、1 から第一引数の正整数迄の成分が異なる分割の個数をリストで返します。この時、実際の分割数は第二成分以降になります。

```
(%i92) integer_partitions(5);
(%o92) {[1, 1, 1, 1, 1], [2, 1, 1, 1], [2, 2, 1], [3, 1, 1], [3, 2], [4, 1],
[5]}

(%i93) num_distinct_partitions(5);
(%o93) 3
(%i94) num_distinct_partitions(5,list);
(%o94) [1, 1, 1, 2, 2, 3]
```

6.8.6 集合に関連する真理値関数

—— 集合に関連する真理値関数 ——

```
setp(< 対象 >)
empty(< 集合 >)
elementp(< 要素 >,...,< 集合 >)
subsetp(< 集合1 >,< 集合2 >)
setequalp(< 集合1 >,< 集合2 >)
disjointp(< 集合1 >,...,< 集合2 >)
```

setp 関数は与えられた対象が集合型であるかどうかを判別する関数です

empty 関数は与えられた集合が空集合かどうかを判別する関数です。

elementp は第一引数で与えられた対象が第二引数で与えられた集合に含まれるかどうかを判定する関数です。

setequalp は与えられた二つの集合が同等のものであるかどうかを判定する関数です。

`disjointp` 関数は与えられた二つの集合に共通部分があるかどうかを判定する関数で、共通部分が無い場合に `true` を返します。

6.9 配列

6.9.1 Maxima の配列について

Maxima はリストの他に配列が扱えます。Maxima で配列を生成する場合、幾つかの方法があります。まず、`array` 関数で配列を宣言して具体的な値を割当てて行く方法と、配列を宣言せずに直接値を割当てて行く方法、`make-art-q` 関数や `make_array` 関数を使う方法があります。

```
(%i1) a1[1,2]:10;
(%o1)                                     10
(%i2) a1[0,3]:1;
(%o2)                                     1
(%i3) a2:make_art_q(10);
(%o3)      {Array:  #(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)}
(%i4) array(a3,fixnum,5);
(%o4)                                     a3
(%i5) make_array(hashed,5);
(%o5)      {Array:  #(NIL NIL $HASHED NIL NIL G13202)}
```

最初に示しているのが配列を無宣言で生成する方法で、この場合は配列に直接式や文字列等を入力する事が可能です。又、Maxima の配列は C の配列と同様に 0 から開始します。配列への値の代入は、この例の様に直接指定する事で行います。他に、`fillarray` 関数等を用いて他の配列やリストから生成する事も可能です。

配列を関数を用いて生成する場合、目的に応じて三種類の関数が使えます。まず、`make-art-q` 関数で生成する配列は LISP の `make-array` 関数を直接用いて配列を生成するものです。その為、Maxima の配列の中では最も原始的なものとなります。

`array` 関数で配列を生成する場合、引数に配列名を与える必要があります。そして、最後の `make_array` 関数は `array` 関数の様に配列名を与える必要はありません。これらの関数では共に、配列の型を指定する事が可能です。

Maxima には生成した配列を調べる関数として `listarray` 関数と `arrayinfo` 関数の二つがあります。

—— 配列の情報を表示する関数 ——

```
listarray(< 配列名 >)
arrayinfo(< 配列名 >)
```

最初の listarray 関数は配列データを表示し, arrayinfo 関数は配列の型や大きさを返す関数です.

では, 先程の例から, listarray 関数と arrayinfo 関数で配列の情報を調べてみましょう.

```
(%i6) listarray(a1);
(%o6) [1, 10]
(%i7) arrayinfo(a1);
(%o7) [hashed, 2, [0, 3], [1, 2]]
(%i8) arrayinfo(a2);
(%o8) [declared, 1, [9]]
(%i9) arrayinfo(a3);
(%o9) [complete, 1, [5]]
```

先ず, listarray(a1) で, 配列 a1 に 1 と 10 が設定されている事が判ります. arrayinfo(a1) で返却される hashed が配列 a1 の型になります. その次に配列の次元があり, 最後にデータが設定されている個所が表示されています.

ここで表示されている配列の型を詳しく述べる為に, 配列を生成する関数について詳細を説明しましょう.

配列の生成

```
make_art_q(< 整数 >)
array(< 配列名 >, < 整数1 >, ..., < 整数n >)
array(< 配列名 >, < 型 >, ..., < 整数1 >, < 整数n >)
array([< 配列名1 >, ..., < 配列名2 >], < 整数1 >, ..., < 整数n >)
make_array(< 型 >, < 整数1 >, ..., < 整数n >)
make_array(functional, < 関数 >, < 型 >, < 整数1 >, ..., < 整数n >)
arraymake(< 配列名 >, [< 添字1 >, ..., < 添字n >])
```

make_art_q 関数は LISP の make-array 関数を呼出して配列を生成するだけの関数で, Maxima の配列を生成する関数の中では最も原始的な配列を生成する関数です.

array 関数は引数に配列名と次元を指定し, < 整数 > 次の配列を生成します. ここで < 整数 > は 5 以下の整数でなければなりません. 即ち, array 関数で生成可能な配列は 5 次以下の配列です. 例えば, array(a, 2, 3, 4, 5, 6) は問題ありませんが, array(b, 2, 3, 4, 5, 6, 7) はエラーになります.

この array 関数は大域変数 use_fast_arrays の影響を受けます.

use_fast_arrays

変数名	初期値	概要
use_fast_arrays	false	配列の種類を制限

まず, 大域変数 use_fast_arrays が true の場合, make_array 函数を用いて any 型の配列を生成します. 尚, make_array 函数で any 型の配列を生成する場合, LISP の make-array 函数がそのまま使われて初期値が NIL の LISP の配列を生成します. 初期値が設定される事を除くと, make_array 函数で生成される配列との違いはありません.

次に大域変数 use_fast_arrays が初期値ままの false であれば, array 函数は指定した〈型〉を反映した配列を構成します. ここで設定可能な型には, 数値配列の場合, 浮動小数点型と整数型配列, その他に complete 型があります.

array 函数で指定可能な型

型	引数	概要
flonum	[flonum, float]	浮動小数点データ.array-mode 属性に float を設定
fixnum	[fixnum, integer]	整数データ.array-mode 属性に fixnum を設定
function	function	函数型
complete	complete	その他

〈型〉で flonum 型が指定された場合, 初期値として 0.0 が設定され, 配列の array-mode 属性に float が設定されます.

〈型〉で fixnum 型が指定された場合, 初期値として 0 が設定され, 配列の array-mode 属性に fixnum が設定されます.

〈型〉に function と complete が指定された場合と〈型〉が無指定の場合, Maxima 内部では NIL が設定されます. この NIL を listarray 函数は####で表示します.

```
(%i1) array(a1,flonum,5);
(%o1)                                a1
(%i2) listarray(a1);
(%o2)                                [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
(%i3) array(a2,fixnum,3);
(%o3)                                a2
(%i4) listarray(a2);
(%o4)                                [0, 0, 0, 0]
(%i5) array(a3,3);
(%o5)                                a3
```

```
(%i6) listarray(a3);
(%o6)          [####, ####, ####, ####]
```

この例では最初に配列 `a1` を `flonum` 型のデータ配列として生成し, `listarray` 関数で中身を見ています. 配列の成分が 0.0 で初期化されていますね. 以降, `fixnum` 型と型の指定を行わずに生成し, `listarray` 関数を用いて中身を見ています.

`make_array` 関数は `array` 関数よりも複雑な配列が生成可能です. この `make_array` 関数で指定可能な〈型〉には `any` 型, `flonum` 型, `fixnum` 型, `hashed` 型と `functional` 型があります. 尚, `functional` 型の場合は第一引数のみに設定します.

```
(%i45) make_array('any,5);
(%o45)          {Array:  #(NIL NIL NIL NIL NIL)}
(%i46) make_array('fixnum,5);
(%o46)          {Array:  #(0 0 0 0 0)}
(%i47) make_array('flonum,5);
(%o47)          {Array:  #(0.0 0.0 0.0 0.0 0.0)}
(%i48) make_array('hashed,5);
(%o48)          {Array:  #(NIL NIL HASHED NIL NIL G13873)}
(%i49) make_array(functional,'sin,flonum,3);
(%o49) {Array:  #(NIL NIL $FUNCTIONAL NOTEXIST %SIN #(0.0 0.0 0.0))}
```

`flonum` 型と `fixnum` 型を指定した場合, `make_array` 関数を用いた配列のみを生成します. その為, `flonum` 型の場合は初期値が 0.0, `fixnum` 型の場合, 初期値が 0 の配列となります. 尚, 内部的には LISP の通常の配列となります.

`flonum` 型と `fixnum` 型以外の型を指定した場合, 内部的に構造体 `mgenarray` 型のデータが生成されます. このデータ型の場合, 配列データ本体は `mgenarray` の `content` に設定されます.

先ず, `any` 型の場合, LISP の `make_array` 関数を用いて初期値 `NIL` の配列が生成されます. 尚, `listarray` 関数で表示を行うと `####` で初期値が表示されます.

`hashed` 型と `functional` 型に関しては, `content` に配列本体が設定される仕様の筈ですが, 通常の配列成分の割当を行うと, 配列データが壊れる為, これらの型は事実上使いません.

`array` 関数と `make_array` 関数で生成した配列は大域変数 `arrays` に登録されます. `make_array` 関数で, `hashed` 型と `functional` 型を指定した場合, `gensym` 関数で生成したシンボルがこのリストに登録されます.

—— 配列が登録されるリスト ——

変数名	初期値	概要
arrays	[] 生成した配列名が登録されるリスト	

大域変数 arrays に登録される配列は, array 関数で生成した全ての配列と, make_array 関数で型が hashed で生成した配列です. 尚, この場合は LISP の gensym 関数で生成されたシンボルが表示されます.

```
(%i1) array(a1,fixnum,5)$
(%i2) array(a2,flonum,5)$
(%i3) array(a3,5)$
(%i4) make_array(hashed,5)$
(%i5) arrays;
(%o5) [a1, a2, a3, g13158]
```

但し, listarray 関数や arrayinfo 関数でこのシンボルは使えません. その為, array 関数で生成した配列名のみが登録されたリストと考えた方が実用上問題がありません. arraymake 関数は funmake 関数と同様に実体の無い形式的な配列を生成する関数です.

6.9.2 配列操作に関連する関数

—— リストや配列の値を配列に割当て関数 ——

```
fillarray(<配列>,<リスト>)
fillarray(<配列>,<配列>)
```

第一引数の <配列> に第二引数に指定した <リスト> か <配列> の値を入れます. <配列> が浮動小数点数 (整数) 配列ならば, 第二引数は浮動小数点 (整数) のリストか浮動小数点 (整数) の配列のどちらかでなければなりません.

第一引数の配列には第二引数の内容が先頭から順番に入れられますが, もしも, 第一引数の配列が第二引数よりも大きければ, 第二引数の最後部の元で第一引数の配列の残りの個所を埋めてしまいます.

—— 配列から指定したデータを取出す関数 ——

```
arrayapply(<配列>,[<添字1>,...,<添字k>])
```

arrayapply は第一引数に〈配列〉を取り, その後に配列の添字リストを指定します. 返却値は指定した添字に対応する配列の値です.

—— 配列の大きさを変更する関数 ——

```
rearray(〈配列〉, 〈次元1〉, ..., 〈次元n〉)
```

rearray 関数で配列の大きさの変更を行います. この場合, 新しい配列に古い配列の元は番号順に代入されて行きます. 新しい配列が古い配列よりも大きなものであれば, 残りは 0 か 0.0 の何れかで埋められます.

—— 配列を削除する関数 ——

```
remarray(〈配列1〉, 〈配列2〉, ...)
remarray(all)
```

remarray 関数は関数を除去し, 占領されていた保存領域を解放します. 引数が all であれば全ての配列を除去します.

ここで, 〈式_i〉の内部表現での先頭にある演算子 (主演算子と呼びます) は全て同じもので, map 関数を作用させた結果は〈関数〉を作用させた各成分を主演算子で繋げたものとなります.

6.10 行列

6.10.1 行列について

Maxima では行列を扱う事が出来ます. 但し, MATLAB の様な数値行列処理ソフトの様にリストの一種類として直接設定出来るものではありません.

一般的には, `matrix` 函数を用いて行列を生成します. この `matrix` 函数は Maxima の複数のリストから行列を生成する函数です. 構文は以下の様になっています.

——— 行列の定義を行う `matrix` 函数 ———

```
matrix(< リスト1 >, ..., < リストn >)
```

ここで, $\langle \text{リスト}_i \rangle$ は行列の i 行に対応します. Maxima のリストは `[1,2,3]` の様に, コンマで区切った式の列を大括弧で括ったものです. 行列は Maxima の内部では, 先頭が `matrix` で開始する S 式となっており, `substpart` 等の函数を用いて, `matrix` を `"["` に変更すれば, リストに変換出来ます. 又, 逆にリストから行列に変換する事も可能です, 以下に行列の生成と行列からリスト, リストから行列への変換の例を示します.

```
(%i3) a:matrix([1,2,3],[4,5,6]);
                                [ 1  2  3 ]
(%o3)                          [          ]
                                [ 4  5  6 ]

(%i4) :lisp a;
((MATRIX SIMP) ((MLIST SIMP) 1 2 3) ((MLIST SIMP) 4 5 6))
(%i4) ?car(a);
(%o4)                                (matrix, simp)

(%i5) b:substpart("[" ,a,0);
(%o5)                        [[1, 2, 3], [4, 5, 6]]

(%i6) c:substpart(matrix,b,0);
                                [ 1  2  3 ]
(%o6)                          [          ]
                                [ 4  5  6 ]
```

行列の和と差は演算子 `+` と演算子 `-` を各々用います. 猶, 可換積演算子 `*` と商演算子 `/` は行列の成分同士の積と商になります. 通常の行列積を行いたい場合には, 非可換積演算子を用います.

以下に matrix 関数による行列の生成と、演算子 $+$, $-$, $*$, $/$ の結果を示します.

```
(%i10) a:matrix([1,0,0],[0,2,0],[0,0,3]);
          [ 1  0  0 ]
          [      ]
(%o10)     [ 0  2  0 ]
          [      ]
          [ 0  0  3 ]

(%i11) b:matrix([1,2,3],[1,3,5],[9,7,5]);
          [ 1  2  3 ]
          [      ]
(%o11)     [ 1  3  5 ]
          [      ]
          [ 9  7  5 ]

(%i12) a+b;
          [ 2  2  3 ]
          [      ]
(%o12)     [ 1  5  5 ]
          [      ]
          [ 9  7  8 ]

(%i13) a-b;
          [ 0  -2  -3 ]
          [      ]
(%o13)     [ -1  -1  -5 ]
          [      ]
          [ -9  -7  -2 ]

(%i14) b*b;
          [ 1  4  9 ]
          [      ]
(%o14)     [ 1  9  25 ]
          [      ]
          [ 81 49 25 ]
```



```
(%i15) b/b;
```

```

[ 1  1  1 ]
[      ]
(%o15)
[ 1  1  1 ]
[      ]
[ 1  1  1 ]

```

この様に、演算子 $+$ と $-$ は通常の行列の和と差になりますが、演算子 $*$ は勝手が違います。何故なら、この積演算子 $*$ は可換性を持っている為、可換性を持たない行列の積は表現出来ないからです。その為、通常の行列の積演算子は非可換積の演算子を用います。この演算子は見落とし易いので、ここでは、非可換積、或いは非可換積と表記します。この非可換積による演算は行列 A と B に対して、 $A \cdot B$ と記述します。この非可換積とその他の演算子に関しては大域変数で、その分配律や結合律が制御出来ます。デフォルトでは、分配律と結合律を満す様になっています。次に、非可換積に対応する行列の冪乗は記号 $^{\wedge}$ で記述します。特に、行列 A の逆行列が存在する場合、 $A^{\wedge}(-1)$ で行列 A の逆行列を表現します。猶、非可換積の詳細に関しては、§5.3.6 を参照して下さい。

6.10.2 行列を生成する函数

Maxima には `matrix` 函数の他にも行列を生成する函数を持っています。

—— 行列の生成を行う函数 ——

```

entermatrix(< 整数1 >, < 整数2 >)
ident(< 整数 >)
zeromatrix(< 整数1 >, < 整数2 >)
genmatrix(< 配列 >, < 整数1 >, < 整数2 >, < 整数3 >, < 整数4 >)
ematrix(< 整数1 >, < 整数2 >, < x >, < 整数3 >, < 整数4 >)
diagramatrix(< 整数1 >, < 整数2 >)
coefmatrix([< 方程式1 >, ...], [< 変数1 >, ...])
augcoefmatrix([< 方程式1 >, ...], [< 変数1 >, ...])
echelon(< 行列 >)

```

`entermatrix` を実行すると、Maxima の要求に従って $\langle \text{整数}_1 \rangle \times \langle \text{整数}_2 \rangle$ 個の成分を入力して $\langle \text{整数}_1 \rangle$ 行 $\times \langle \text{整数}_2 \rangle$ 列の行列を生成する函数です。

```
(%i1) entermatrix(3,3);
is the matrix 1. diagonal 2. symmetric 3. antisymmetric
4. general
```

```
answer 1, 2, 3 or 4
```

```
1;
```

```
row 1 column 1: a;
```

```
row 2 column 2: b;
```

```
row 3 column 3: c;
```

```
matrix entered.
```

```
(%o1)      [ a  0  0 ]
            [      ]
            [ 0  b  0 ]
            [      ]
            [ 0  0  c ]
```

この様に対話的に行列を生成する事が出来ますが, 実用上, 用途は限られるかもしれません.

ident 関数は, $\langle \text{整数} \rangle$ 次の単位正方行列, 即ち, 対角成分が全て 1 で, 他が全て 0 となる正方行列を生成します.

```
(%i6) ident(3);

(%o6)      [ 1  0  0 ]
            [      ]
            [ 0  1  0 ]
            [      ]
            [ 0  0  1 ]
```

zeromatrix 関数は, $\langle \text{整数}_1 \rangle$ 行 $\langle \text{整数}_2 \rangle$ 列の零行列を返します.

```
(%i7) zeromatrix(3,2);

(%o7)      [ 0  0 ]
            [      ]
            [ 0  0 ]
            [      ]
            [ 0  0 ]
```

genmatrix 函数を用いると、与えられた二次元配列から行列が生成出来ます。この genmatrixi 函数は二次元配列から行列を抜き出す形になります。抜き出し方は、 $\langle \text{整数}_1 \rangle, \langle \text{整数}_2 \rangle$ で配列の大きさを指定します。それから、 $\langle \text{整数}_3 \rangle, \langle \text{整数}_4 \rangle$ で行列の 1 行, 1 列目の成分を指定します。すると、 $\langle \text{整数}_1 \rangle$ から $\langle \text{整数}_3 \rangle$ 迄の配列の第一成分を行とし、 $\langle \text{整数}_2 \rangle$ から $\langle \text{整数}_4 \rangle$ 迄の第二成分を列とする領域が $\langle \text{整数}_1 \rangle - \langle \text{整数}_3 \rangle$ 行, $\langle \text{整数}_2 \rangle - \langle \text{整数}_4 \rangle$ 列の行列として抜き出されます。ここで、実際の配列が生成する行列よりも小さかった場合、即ち、 i 行 j 列の成分が未定の場合、 $\langle \text{配列} \rangle_{ij}$ の形式で行列の成分が表示されます。又、以下に示す様に配列を函数で与える事も可能です。

```
(%i3) h[i,j]:=1/(i*j+1)
(%i4) genmatrix(h,3,3);
```

[	1	1	]	
[	1	-	-	]
[	2	3	]	
[			]	
[	1	1	1	]
[	-	-	-	]
[	2	3	4	]
[				]
[	1	1	1	]
[	-	-	-	]
[	3	4	5	]

ematrix 函数は $\langle \text{整数}_1 \rangle$ 行 $\langle \text{整数}_2 \rangle$ 列の行列で、 $(\langle \text{整数}_3 \rangle, \langle \text{整数}_4 \rangle)$ 成分のみが $\langle x \rangle$ となり、他が全て零となる行列を生成します。

```
(%i9) ematrix(4,3,1,1,1);
```

[	1	0	0	]
[				]
[	0	0	0	]
[				]
[	0	0	0	]
[				]
[	0	0	0	]

diagmatrix 函数は diagmatrix(n,x) で、 n 行 n 列の行列でその対角成分が x の対角

行列を返します. ここで対角成分は全て〈行列〉のものです. 猶, `diagmatrix(n,1)` は `ident(n)` と同じ n 次元の単位行列を生成します.

```
(%i19) diagmatrix(3,2);
```

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

```
(%o19)
```

`coefficientmatrix` は連立一次方程式の変数リストに対応する係数行列を返します. 猶, 連立一次方程式は, 演算子=の右辺, 或いは左辺の何れかが 0 の場合, 式だけでも構いません.

```
(%i22) coefficientmatrix([2*x-3*y-1=0,3*x+3*y+10=0],[x,y]);
```

$$\begin{bmatrix} 2 & -3 \\ 3 & 3 \end{bmatrix}$$

```
(%o22)
```

```
(%i23) coefficientmatrix([2*x-3*y-z,3*x+3*y+10*z],[x,y]);
```

$$\begin{bmatrix} 2 & -3 \\ 3 & 3 \end{bmatrix}$$

```
(%o23)
```

`augcoefficientmatrix` 関数は, 与えられた方程式と指定した変数のリストから係数行列を生成します. 行列は, 〈方程式₁〉, … から構成される線形方程式系に含まれる 〈変数₁〉, … の係数から構築されます. `coefficientmatrix` 関数との違いは, 生成する係数行列に各方程式の定数項が列として付加される点です.

```
(%i25) augcoefmatrix([2*x-3*y-z,3*x+3*y+10*z],[x,y]);
```

```
(%o25)      [ 2  - 3  - z  ]
             [                ]
             [ 3   3  10 z  ]
```

```
(%i26) augcoefmatrix([2*x-3*y=1,3*x+3*y=10],[x,y]);
```

```
(%o26)      [ 2  - 3  - 1  ]
             [                ]
             [ 3   3  - 10  ]
```

```
(%i27)
```

echelon 関数は、〈行列〉の echelon 形式を生成します。これは初等的な行操作で各々の行の最初の非零元を 1, その元を含む列に対しては、その元を含む行よりも下の成分を全て零となる様に変形するものです (上三角行列)。

```
(%o2)      [2  1 - a  -5 b ]
             [                ]
             [a   b    c  ]
```

```
(%i3) echelon(d2);
```

```
(%o3)      [      a - 1      5 b      ]
             [1  - -----  - ----  ]
             [      2      2      ]
             [                ]
             [      2 c + 5 a b ]
             [0      1  -----]
             [                2      ]
             [      2 b + a  - a]
```

行列処理を指定する大域変数

変数名	初期値	概要
assumescalar	true	引数がスカラー値であると仮定.
detout	false	余因子行列を用いた逆行列計算で, 行列式の処理を制御
mx0simp	true	0 との積を 0 で返すかどうかを制御
ratmx	false	行列成分の CRE 表現の表示を制御
scalarmatrixp	true	1 行 1 列行列のスカラへの自動変換を制御
sparse	false	疎行列を計算するかどうかを指定

assumescalar が true の場合, 値が割当てられていない変数と行列の可換積は, 行列の各成分との可換積で自動的に置換されます.false の場合, 変数は各成分に分配されません.

detout が true であれば, 逆行列を計算した時に行列式の割算がそのまま行列の外に残されます. この大域変数が効力を持つ為には doallmxops と doscmxops が false でなければなりません. この設定をその他の二つが設定される ev で与える事も出来ます.

mx0simp が true であれば, 行列と 0 との積は成分が全て 0 の行列として返却されますが, false であれば, 行列と 0 の積は 0 を返します.

ratmx が false であれば, 行列式や行列の和, 差, 積が行列の表示形式で行われ, 逆行列の結果も一般の表示となります.true であれば, これらの演算は CRE 表現で実行され, 逆行列の結果も CRE 表現となります. これは成分が往々にして望みもしない展開 (ratfac の設定に依存するものの) の原因になります.

scalarmatrixp が true であれば, 二つの行列の非可換積の計算で得られた 1 行 1 列の行列はスカラーに変換されます. もし, all に設定されていれば, この変換は, 1 行 1 列の行列は何時でもスカラに変換されます. 但し, false であれば, この変換は実行されません.

sparse が true で ratmx:true であれば, determinant は疎行列式を計算する為のルーチンを利用します.

do 一家

変数名	初期値	概要
doallmxops	true	行列演算子の評価に関連
domxexpt	true	行列に対する指数関数の処理
domxmxops	true	対行列, 対リスト間の演算を管理
domxnctimes	false	非可換積の実行に関連
doscmxops	false	スカラと行列間の演算を実行
doscmxplus	false	スカラ+行列の処理に関連

doallmxops が true であれば, 全ての行列演算子が評価されます. false であれば, 演算子を支配する個々の dot 大域変数の設定が実行されます.

domxexpt が true の場合, $\%e^{\text{matrix}([1,2],[3,4])}$ は $\text{matrix}([\%e, \%e^2], [\%e^3, \%e^4])$ となります. 一般的に, この変換は $\langle \text{基底} \rangle^{\langle \text{次数} \rangle}$ の形式の変換に影響します. 猶, $\langle \text{基底} \rangle$ はスカラか定数の式であり, $\langle \text{次数} \rangle$ はリストか行列です. この大域変数が false であれば, この変換は実行されません.

domxmxops が true であれば, 行列と行列間の演算子や行列とリストの間の演算子が実行されます. この大域変数が false なら, これらの演算は実行されません. 猶, この大域変数はスカラと行列との間の演算には影響を与えません.

domxnctimes が false であれば行列の非可換積が実行されます.

doscmxops が true であればスカラと行列間の演算子が実行されます.

doscmxplus が true であれば, スカラ+行列が行列値となります. この大域変数は doallmxops と独立した変数です.

行列の括弧を設定する大域変数

変数名	初期値	概要
lmxchar	[	行列の右側の括弧で用いる文字を設定
rmxchar	]	行列の左側の括弧で用いる文字を設定

lmxchar は行列の (左) の括弧として表示する文字を設定します. 右側は rmxchar で指定します.

rmxchar は行列の (右) の括弧として表示する文字を設定します. 左側は lmxchar で指定します.

matrix_element 変数一族

変数名	初期値	概要
matrix_element_add	+ 行列の和の演算子を指定	
matrix_element_mult	*	行列の成分間の積の演算子を指定
matrix_element_transpose	false	転置の際に作用させる関数を設定

matrix_element_add は行列同士の和を計算する際に用いる演算子を設定します。関数名や lambda 式であっても構いません。

matrix_element_mult は行列の成分同士の積を計算する際に用いる演算子を設定します。関数名や lambda 式であっても構いません。

matrix_element_transpose は上記の大域変数と同様に、転置行列を計算する際に、作用させる関数や lambda 式を指定します。

6.10.3 行列に関連する関数

行列の成分に関数作用させる関数

matrixmap(< 関数 >, < 行列 >)

matrixmap 関数は、行列 $\langle m \rangle$ の各成分に $\langle \text{関数} \rangle$ を作用させる関数です。リストに対する map 関数の行列版です。

```
(%i11) A:ident(3)*3;
```

```
[ 3  0  0 ]
```

```
[    ]
```

```
(%o11)
```

```
[ 0  3  0 ]
```

```
[    ]
```

```
[ 0  0  3 ]
```



```
(%i12) matrixmap(lambda([x],cos(x)*exp(-x)),A);
      [  - 3
      [ %e   cos(3)      1      1      ]
      [
(%o12)      [      - 3
      [      1      %e   cos(3)      1      ]
      [
      [      - 3
      [      1      1      %e   cos(3) ]
```

— 行や列の取り出しと追加を行う関数 —

```
col(〈 行列 〉,〈i〉)
row(〈 行列 〉,〈i〉)
addcol(〈 行列 〉,〈 リスト1 〉,〈 リスト2 〉,...,〈 リストn 〉)
addrow(〈 行列 〉,〈 リスト1 〉,〈 リスト2 〉,...,〈 リストn 〉)
copymatrix(〈 行列 〉)
setelm(x,〈i〉,〈j〉,〈 行列 〉)
```

col 関数と row 関数は、〈 行列 〉に対し、〈i〉で指定される列と行を各々行列の形式で返す関数です。

```
(%i17) m1;
      [ x   0   - 3   x   0   0 ]
      [
      [ 0   x   0   - 3   x   0 ]
      [
      [ 0   0   x   0   - 3   x ]
(%o17)      [
      [ y  - 3   0   y   0   0 ]
      [
      [ 0   y  - 3   0   y   0 ]
      [
      [ 0   0   y  - 3   0   y ]

(%i18) row(m1,1);
(%o18)      [ x   0   - 3   x   0   0 ]
```

```
(%i19) col(m1,1);
```

```
[ x ]
[   ]
[ 0 ]
[   ]
[ 0 ]
[   ]
[ y ]
[   ]
[ 0 ]
[   ]
[ 0 ]
```

```
(%o19)
```

addcol 関数は列として, addrow 関数は行として, 複数の複数のリストや行列を〈行列〉に追加します. 猶, 追加するリストや行列は, 行列の大きさに矛盾しないものでなければなりません.

copymatrix 関数は〈行列〉の複製を行います. この関数は〈行列〉を成分毎に再生成する時だけに使います. 猶, copymatrix 関数の実体は, copy 関数を matrixp 関数を用いて行列に制限した関数です.

setelm 関数は〈行列〉の $(\langle i \rangle, \langle j \rangle)$ 成分を $\langle x \rangle$ で置換します.

```
(%i13) A;
```

```
[ 4  0  0  0 ]
[           ]
[ 0  4  0  0 ]
[           ]
[ 0  0  4  0 ]
[           ]
[ 0  0  0  4 ]
```

```
(%o13)
```

```
(%i14) setelmx(4,2,3,A);
```

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ & & & \\ 0 & 4 & 4 & 0 \\ & & & \\ 0 & 0 & 4 & 0 \\ & & & \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

```
(%o14)
```

猶, setelmx 函数を使わなくても, 行列成分を直接指定して置換える事も可能です. この場合, $A[i, j]:x$ で行列 A の (i, j) 成分を x で置換します. 但し, この場合の返却値は x になります.

—— 小行列を生成する函数 ——

```
minor(⟨ 正方行列 ⟩, ⟨ i ⟩, ⟨ j ⟩)
submatrix(⟨ 行1 ⟩, ..., ⟨ 行m ⟩, ⟨ 行列 ⟩, ⟨ 列1 ⟩, ..., ⟨ 列n ⟩)
submatrix(⟨ 行1 ⟩, ..., ⟨ 行m ⟩, ⟨ 行列 ⟩)
submatrix(⟨ 行列 ⟩, ⟨ 列1 ⟩, ..., ⟨ 列n ⟩)
```

minor 函数は与えられた ⟨ 正方行列 ⟩ の ⟨ i ⟩, ⟨ j ⟩ 成分の小行列, つまり, ⟨ 正方行列 ⟩ から ⟨ i ⟩ 行と ⟨ j ⟩ 列を抜いた行列を返します.

submatrix 函数は ⟨ 行 _{i} ⟩ 行と ⟨ 列 _{j} ⟩ 列が削除された新しい行列を生成します. submatrix 函数は minor 函数よりも多くの行と列が削除出来ます.

—— 転置, 上三角, 共役な行列を計算する函数 ——

```
transpose(⟨ 行列 ⟩)
triangularize(⟨ 行列 ⟩)
```

transpose 函数は ⟨ 行列 ⟩ の転置行列を生成します.

```
(%i9) A:matrix([1,2,3],[4,3,1]);
      [ 1  2  3 ]
(%o9)  [          ]
      [ 4  3  1 ]

(%i10) transpose(A);
      [ 1  4 ]
      [      ]
(%o10) [ 2  3 ]
      [      ]
      [ 3  1 ]
```

triangularize 関数は〈行列〉の上三角行列形式を生成します。猶, 行列が正方行列である必要はありません。

```
(%i6) A:matrix([1,2,3,4],[3,4,5,1],[2,3,1,5]);
      [ 1  2  3  4 ]
      [          ]
(%o6)  [ 3  4  5  1 ]
      [          ]
      [ 2  3  1  5 ]

(%i7) triangularize(A);
      [ 1  2  3  4 ]
      [          ]
(%o7)  [ 0 -2 -4 -11 ]
      [          ]
      [ 0  0  6 -5 ]
```

—— 階数と対角和を計算する関数 ——

```
rank(〈行列〉)
mattrace(〈行列〉)
```

rank 関数は〈行列〉の階数を求めます。行列の階数は行列から求めた小行列式で、零にならない小行列式で、最大のものの大きさです。猶,rank は行列成分の値が非常に零に近い場合には誤った答を返す事があります。

mattrace 関数は、〈行列〉が正方行列の場合、対角和、即ち、行列の主対角成分の総和を計算します。この関数は、ncharpoly で利用されています。ncharpoly は Max-

ima の charpoly の代りに使える函数です. 猶, mattrace 函数を利用する為には, 予め `load("nchrpl");` を実行する必要があります.

```
(%i14) load(nchrpl)$
(%i15) A:matrix([1,2,3],[4,3,1],[-2,0,-2]);
          [ 1  2  3 ]
          [      ]
(%o15)     [ 4  3  1 ]
          [      ]
          [ -2  0 -2 ]

(%i16) mattrace(A);
(%o16)                2
```

—— 余因子行列の計算に関連する函数 ——

adjoint(〈 正方行列 〉)
invert(〈 正方行列 〉)

adjoint 函数は 〈 正方行列 〉 の余因子行列を計算します.

invert 函数は逆行列を余因子行列を用いた方法で計算します. これは bfloat 値成分や浮動小数点を係数とする多項式を成分とする行列の逆行列を CRE 形式に変換せずに計算出来ます. determinant 命令は余因子の計算で利用されるので, ratmx が false ならば, その逆行列は成分表現を変更せずに計算されます. 現行の実装は高い次数の行列に対して効率的なものではありません. 猶, 大域変数 detout が true の場合, 行列式の部分は逆行列の外側に出力されます.

invert が返した結果は展開されていません. 最初から多項式成分を持つ行列の場合, `expand(invert(mat)) ,detout;` で生成された出力は見栄えが良くなります.

```
(%i28) A:matrix([t,1,t-2],[1,t,0],[t+1,1,t-1]);
          [  t   1  t - 2 ]
          [      ]
(%o28)     [  1   t   0   ]
          [      ]
          [ t + 1  1  t - 1 ]
```

```
(%i29) adjoint(A);
      [ (t - 1) t      - 1      - (t - 2) t ]
      [
(%o29) [ 1 - t      (t - 1) t - (t - 2) (t + 1)      t - 2 ]
      [
      [
      [ 1 - t (t + 1)      1      t - 1 ]
(%i30) invert(A), expand, detout;
      [ 2      2 ]
      [ t - t      - 1 2 t - t ]
      [
      [ 1 - t      2      t - 2 ]
      [
      [ 2      2 ]
      [ - t - t + 1 1      t - 1 ]
(%o30) -----
      2 t - 1
```

猶, 行列式を行列の中に入れる場合, $\text{adjoint}(\langle \text{行列} \rangle) / \text{determinant}(\langle \text{行列} \rangle)$ を計算した結果を `expand` で展開したり, 有理数係数の多項式が行列の成分に現われる場合は, `ratsimp` 函数を用いると良いでしょう.

```
(%i35) adjoint(A)/determinant(A),expand;
```

```

[      2      2      ]
[      t      t      1      2 t      t      ]
[      ----- - ----- - ----- ]
[      2 t - 1  2 t - 1  2 t - 1  2 t - 1  2 t - 1 ]
[      ]
[      1      t      2      t      2      ]
(%o35) [      ----- - ----- - ----- ]
[      2 t - 1  2 t - 1  2 t - 1  2 t - 1  2 t - 1 ]
[      ]
[      2      2      ]
[      t      t      1      1      t      1      ]
[      - ----- - ----- + ----- - ----- - ----- ]
[      2 t - 1  2 t - 1  2 t - 1  2 t - 1  2 t - 1  2 t - 1 ]

```

```
(%i36) adjoint(A)/determinant(A),ratsimp;
```

```

[      2      2      ]
[      t - t      1      t - 2 t ]
[      ----- - ----- - ----- ]
[      2 t - 1  2 t - 1  2 t - 1 ]
[      ]
[      t - 1      2      t - 2      ]
(%o36) [      ----- - ----- - ----- ]
[      2 t - 1  2 t - 1  2 t - 1 ]
[      ]
[      2      2      ]
[      t + t - 1      1      t - 1      ]
[      - ----- - ----- - ----- ]
[      2 t - 1  2 t - 1  2 t - 1 ]

```

行列式を計算する関数

```
determinant(⟨ 行列 ⟩)
newdet(⟨ 行列 ⟩)
newdet(⟨ 配列 ⟩,⟨ 整数 ⟩)
permanent(⟨ 行列 ⟩,⟨ 整数 ⟩)
```

determinant 関数は, Gauss の消去法と似た方法で ⟨ 行列 ⟩ の行列式を計算します. 計算結果の書式は大域変数 `ratmx` の設定に依存します.

疎行列の行列式を計算する特別な方法もあり, `ratmx:true` と `sparse:true` に設定した場合に使えます.

`newdet` 関数は ⟨ 行列 ⟩ や ⟨ 配列 ⟩ の行列式を計算します. この際に, Johnson-Gentleman tree minor アルゴリズムを用います. 猶 ⟨ 整数 ⟩ を指定した場合, 1 行から ⟨ 整数 ⟩ 行と 1 列から ⟨ 整数 ⟩ 列の正方行列を取出し, その行列式を計算します. この整数値が無指定の場合, ⟨ 行列 ⟩ が正方行列であればそのまま行列式を計算し, ⟨ 行列 ⟩ が正方行列で無ければ, 余分な末尾の行, 或いは列を削除した正方行列の行列式を返します.

```
(%i23) A;
          [ 1  2  3  4 ]
          [          ]
(%o23)    [ 3  4  5  1 ]
          [          ]
          [ 2  3  1  5 ]

(%i24) newdet(A,3);
(%o24)/R/          6
```

`permanent` 関数は, ⟨ 行列 ⟩ の `permanent` を計算します. 猶 ⟨ 整数 ⟩ を指定した場合, 1 行から ⟨ 整数 ⟩ 行と 1 列から ⟨ 整数 ⟩ 列の正方行列を取出し, その行列式を計算します. ここで, `permanent` は行列式に似ていますが, 符号の変化のないものです.

特性多項式の生成に関連する関数

```
charpoly(⟨ 行列 ⟩,⟨ 変数 ⟩)
ncharpoly(⟨ 行列 ⟩,⟨ 変数 ⟩)
```

`charpoly` 関数は ⟨ 行列 ⟩ の特性多項式 $\det(\langle \text{変数} \rangle I - \langle \text{行列} \rangle)$ を計算します.

`determinant` (`(< 行列 > - diagmatrix (length (< 行列 >), < 変数 >))`) と同じ結果を返します.

`ncharpoly` 関数は `< 変数 >` に対する `< 行列 >` の特性多項式を計算します. これは Maxima の `charpoly` とは別物です.

`ncharpoly` では与えられた行列の冪乗の対角和を計算しますが, 対角和は特性多項式の根の冪乗の総和に等しいものです. これらの諸量から根の対称式の計算が可能ですが, それらは特性多項式の係数です. `charpoly` は `var*ident[n]-a` の行列式を計算しており, この点で `ncharpoly` は優れています. 例えば, 整数成分の非常に大きな行列の場合は算術的に多項式の計算を避ける為です. 予め `load("nchrpl");` で読み込む必要があります.

— 行列をリストに変換する関数 —

```
list_matrix_entries(< 行列 >)
```

`list_matrix_entries` 関数は与えられた行列に対して, 各行を繋げた平リストを返す関数です.

```
(%i10) A1:matrix([1,2,3],[4,5,6]);
                                [ 1  2  3 ]
(%o10)                          [          ]
                                [ 4  5  6 ]

(%i11) list_matrix_entries(A1);
(%o11)                        [1, 2, 3, 4, 5, 6]
```

6.10.4 eigen パッケージ

Maxima に標準で附属する `eigen` パッケージには, 固有値計算に関連する関数と, 基底の直交化に関連する関数が含まれています.

ここで説明する関数を利用する為には, `load(eigen);` を予め実行します. この `load(eigen)` を実行する事で, 関数が読み込まれ, 以下の大域変数が定義されます.

eigen パッケージで定義される大域変数

変数名	初期値	概要
hermitianmatrix	false	Hermit 行列かどうかを指定
nondiagonalizable	false	非対角化行列かどうかを指定
knowneigvals	false	固有値を既知として扱うかどうかを指定
knowneigvects	false	固有ベクトルを既知として扱うかどうかを指定
listeigvects	[]	固有ベクトルのリスト
listeigvals	[]	固有値のリスト
rightmatrix	[]	左行列
leftmatrix	[]	右行列

大域変数 hermitianmatrix が true の場合, 与えられた行列が Hermite 行列であると仮定します. その為, 大域変数 leftmatrix は rightmatrix の転置行列と複素共役な行列になります. 又, rightmatrix は $\langle$ 行列 $\rangle$ の正規化した固有ベクトルを列とする行列になります.

大域変数 nondiagonalizable が false の場合, 大域変数 leftmatrix と rightmatrix に行列が設定され, leftmatrix $\cdot$ $\langle$ 行列 $\rangle$ $\cdot$ rightmatrix が対角行列で, $\langle$ 行列 $\rangle$ の固有値がその対角成分に現れるものとなります. 但し, nondiagonalizable が true であれば, これらの行列は生成されません.

knowneigvals が true の場合, 行列の固有値は既知で, 大域変数の listeigvals に保存されていると eigen パッケージの関数は仮定します. その為, knowneigvals が true の場合は listeigvals に eigenvalues 関数の出力と同じリストが設定されていなければなりません.

knowneigvects が true の場合, 行列の固有値に対応する固有ベクトルが既知であり, 大域変数の listeigvects に保存されていると eigen パッケージの関数が仮定します. その為, knowneigvects が true の場合は listeigvects に eigenvects 関数の出力と同じリストが設定されていなければなりません.

内積関数

innerproduct($\langle x \rangle, \langle y \rangle$)
 inprod($\langle x \rangle, \langle y \rangle$) (innerproduct 関数の別名)

innerproduct 関数は内積を表現します. 短縮名は inprod です. リスト $\langle x \rangle$ と $\langle y \rangle$ を引数として取り, $\langle x \rangle$ の複素共役 $\cdot \langle y \rangle$ で定義されています. ここで, 非可換積は通常のベクトルの内積演算子と同じものです.

eigen パッケージには以下の行列操作の関数が含まれています.

行列操作の関数

```
columnvector(< リスト >)
conjugate(< リスト >)
conj(< リスト >)(conjugate 関数の別名)
```

columnvector 関数は与えられたリストから列ベクトルを生成します.

```
(%i6) columnvector([1,2,3]);

          [ 1 ]
          [   ]
(%o6)      [ 2 ]
          [   ]
          [ 3 ]
```

conjugate 関数は引数の複素共役を返します.

```
(%i3) A:matrix([1,2*%i],[1-%i,4]);

          [ 1      2 %i ]
(%o3)      [           ]
          [ 1 - %i    4   ]

(%i4) conjugate(A);

          [ 1      - 2 %i ]
(%o4)      [           ]
          [ %i + 1    4     ]
```

固有値の計算に関連する eigen パッケージ関数

```
eigenvalues(< 行列 >)
eivals(< 行列 >)(eigenvalues 関数の別名)
eigenvectors(< 行列 >)
eivects(< 行列 >)(eigenvectors 関数の別名)
similaritytransform (< 行列 >)
simtran (< 行列 >)(similaritytransform 関数の別名)
```

eigenvalues 函数 (短縮名 eivals) は引数に一つの行列を取り, 固有値と固有ベクトルを含むリストを返します. 返却されるリストの第一成分が固有値リスト, 第二成分が固有値リストに対応する重複度のリストとなります.

eigenvalues 函数では charpoly 函数で特性多項式を計算し, solve 函数を使ってその特性多項式の根を求めています. solve 函数は厳密解を求める函数の為, 高次多項式に対しては, その根を見付け損なう事があります. 更に, 不正確な答を返す事もあります. しかし eigen パッケージに含まれている conjugate 函数, innerproduct 函数, univector 函数, columnvector 函数と gramschmidt 函数を必要としません.

eigenvectors 函数は引数として一つの行列を取り, リストを返します. このリストに含まれる最初の副リストには eigenvalues 函数の出力, 他の副リストには行列の各々の固有値に対応する固有ベクトルが含まれています.

猶, algsys 函数が固有ベクトルの計算で使われています. 猶, 固有値が不正確な場合, algsys が解を生成する事が出来ない事があります. この場合, eigenvalues 函数を使って最初に見つけた固有値の簡易化を行う事を勧めます.

similaritytransform 函数は (行列) を引数とし, uniteigenvectors 函数の出力結果リストを返します.

—— ベクトルの正規化に関連する函数 ——

```

gramschmidt([< リスト1>, ..., < リストn>])
gschmidt([< リスト1>, ..., < リストn>]) (gramschmidt 函数の別名)
unitvector(< リスト >)
uvect(< リスト >)(unitvector 函数の別名)
uniteigenvectors(< 行列 >)
ueivects(< 行列 >)(uniteigenvectors 函数の別名)

```

gramschmidt 函数は Gram-Schmidt による直交ベクトルを求めます. この函数は eigen パッケージに含まれる函数なので, 予め `load(eigen)` を実行して利用します. gramschmidt 函数は引数にリストの列で構成されるリストを取ります. ここで < リスト_i> は全て長さが等しくなければなりません, 各リストが直交している必要はありません.

この gramschmidt 函数は互いに直交したリストで構成されたリストを返します. 猶, 返ってきた結果には因子分解された整数が含まれる事があります. これは Maxima の factor 函数が gram-schmidt の処理の過程で使われた為です. こうする事で式が複雑なものになる事を回避し, 生成される変数の大きさを減らす助けにもなっています.

unitvector 函数は < リスト > の大きさを 1 にしたリストを返します. uniteigenvalues

は与えられた〈行列〉の固有値と固有ベクトルで構成されたリストを返します. 出力リストの第一成分のリストには `eigenvalues` 関数の出力があり, 第二成分の副リストには正規化した固有ベクトルが, 第一成分のリストの固有値に対応する順番で並んでいます.

`uniteigenvectors` 関数は与えられた行列から長さを 1 にした固有ベクトルを返します.

6.11 文字列

6.11.1 Maxima の文字列

この節では,Maxima での文字列操作に関連する函数,主に contrib に含まれる stringproc パッケージに含まれる函数の解説を行います.

Maxima のデフォルトの文字列操作の函数は殆どありません.しかし,LISP 自体は文字列操作に関しては非常に強力な言語です.

この穴を埋めるのが contrib に含まれる stringproc パッケージです. stringproc パッケージは文字列操作を行う為の様々な函数を含んでいます.これらの函数は stringproc.lisp 内部で定義されており,ファイル名からも分る様に LISP で記述されています.この stringproc パッケージは Maxima の文字操作の弱さを補強するものとなっています.この stringproc パッケージで定義される函数は,大きく分けて四種類あります.一つは入出力に関連するもので,ファイルやストリームの処理を行う函数群です.次に,与えられた引数に対し,真偽値を返す函数があります.更に,与えられた数値を文字に変換する函数の様に,与えられた引数を文字や数値,或いは LISP の文字列データ等に変換する函数群,それから最後に,与えられた Maxima の文字列の結合や指定した位置の文字を取出したりする文字列操作の函数群があります.

ここで注意すべき事として,Maxima の版によって,Maxima の文字列データは LISP の文字列型ではない事です.§6.4.1 で解説した様に,旧来の Maxima の文字列データの内部表現は,文字列データの二重引用符を取り除いて,先頭に&を付けたものになっていました.Maxima-5.10.0 で具体的に確認しておきましょう.

```
(%i19) neko1:"x^2+x-1";
(%o19)                                     x^2+x-1
(%i20) neko2:x^2+x-1;
(%o20)                                     2
                                     x  + x - 1
(%i21) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
```

```
MAXIMA> $neko1
&X^2+X-1
MAXIMA> $neko2
((MPLUS SIMP) -1 $X ((MEXPT SIMP) $X 2))
MAXIMA> (stringp $neko1)
```

NIL

MAXIMA> (atom \$neko1)

T

この例では, 変数 `neko1` に " x^2+x-1 " を割り当て, 変数 `neko2` には式 x^2+x-1 を割り当てています. `to_lisp()` で裏で動作している LISP に入り, 変数 `neko1` と `neko2` の内部表現である `$neko1` と `$neko2` の値を表示させています. ここで, `$neko1` に束縛された値が x^2+x-1 と先頭に `&` が付いて, 更に `x` が大文字になっている事に注意して下さい. そして, `$neko1` に束縛させた値は LISP の文字列型ではない事を `stringp` 関数を用いて確認しています. このデータの型は `atom` 型になります.

ところが, Maxima-5.14.0 から Maxima の文字列と LISP の文字列の扱いは同じものとなっています. この点には注意が必要になります. 今度は Maxima-5.14.0 で確認しておきましょう.

```
(%i2)  neko1:"x^2+x-1";
(%o2)                                     x^2+x-1
(%i3)  :lisp $neko1;
x^2+x-1
(%i3)  to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
```

MAXIMA> \$neko1

"x^2+x-1"

この様に返却される値が全く異なっている事が判ります. 従って, 文字列の扱いは 5.13.0 以前と 5.14.0 以降では注意が必要です.

さて, Maxima には文字列型のデータがありますが, LISP との大きな違いは, Maxima には文字型のデータが無い事です. その為, この節を通して長さが 1 の Maxima の文字列の事を Maxima の文字と簡単に呼ぶ事にします.

長く, Maxima と LISP ではデータ型が異なっていた為, `stringproc` パッケージで定義される関数では, Maxima と LISP との間での文字列データの変換関数や, 判別関数が多く含まれています. 又, LISP にある関数を Maxima の関数に移植する傾向がある為, このパッケージで定義される大域変数は少なく, 関数も単機能なものが多いのが特徴です.

猶, 旧来の Maxima にも文字列操作の関数が無い訳ではありません. 例えば, `sconcat` 関数の様に与えられた複数の Maxima の文字列を結合し, 一つの LISP の文字列を生成する関数があります.

Maxima 固有の文字列操作の関数

```
concat(<式1>, ..., <式n>)
sconcat(<Maxima の文字列1>, ..., <Maxima の文字列n>)
string(<対象>)
```

concat 関数は引数として, Maxima の式, 或いは文字列を取ります. ここで concat 関数で利用可能な式は, Maxima の原子, 或いは, 評価を行う事で, 数値や文字列, 或いは原子が返される式です. concat 関数は引数の評価を行い, それから結果を文字列に変換し, これらの文字列を結合した文字列を返します.

```
(%i28) z1:x$
(%i29) concat(z1,12);
(%o29)                                     x12
(%i30) concat("123",z1,12);
(%o30)                                   123x12
```

sconcat 関数は LISP の concatenate 関数を Maxima に実装したもので, macsys.lisp 内部で定義されている関数です. この関数は与えられた複数の文字列を単純に結合するだけですが, 出力が LISP の文字列になります.

```
(%i1) sconcat("これは","テスト","だよ!");
(%o1)                                     これはテストだよ!
(%i2) neko:sconcat("This"," ","is"," ","a"," ","test!");
(%o2)                                   This is a test!
(%i3) ?stringp(neko);
(%o4)                                   true
```

この例では, sconcat 関数の出力が LISP の文字列型である事を確認しています. 猶, concat 関数, sconcat 関数共に, 引数を一つ以上必要とします.

Maxima-5.14.0 より Maxima の文字列は LISP の文字列型に統合されましたが, Maxima-5.13.0 以前で, 結合した文字列も Maxima の文字列であって欲しい場合, LISP の関数を用いて内部データを料理するか, 次に解説する string 関数を用いるか, 或いは stringproc パッケージの sconc 関数を用いなければなりません.

先ず, string 関数は与えられた対象を Maxima の文字列に変換する関数です. 因に, この関数は suprv1.lisp で定義されています.

```
(%i40) tama:string(factor(x^2+2*x+1));
```



```
(%o40)                (x+1)^2
(%i41) stringp(tama);
(%o41)                true
```

この例では,Maxima の式 $x^2+2*x+1$ を string 関数で Maxima の文字列に変換しています.そして,最後に stringproc パッケージの関数である stringp 関数を用いて Maxima の文字列である事を確認しています. 猶,Maxima-5.14.0 以降では Maxima の文字列型は LISP の文字列型と一致する為, `?stringp(tama)` の返却する値も true になります.

それでは stringproc パッケージに含まれる関数の解説を始めましょう. 最初に入力操作に関連する関数について次の小節で解説しましょう.

6.11.2 ストリーム処理に関連する関数

stringproc パッケージには純粋に文字列の操作に関する関数だけではなく,ストリーム操作の関数も含まれています.Maxima は LISP 上で動作している割に,ストリーム処理が全体的に弱かったのですが,stringproc パッケージを用いる事で,Maxima の処理言語だけを用いても,LISP と機能的に大差無い水準に近づけられるのです.

ファイル直接操作の関数

```
openw(< ファイル名 >)
opena(< ファイル名 >)
openr(< ファイル名 >)
close(< ストリーム >)
```

openw 関数は Lisp のストリーム出力をファイルに落す為に用いる関数です. ファイルが存在しない場合には新たに生成され,既存の場合にはファイルを開きますが,ファイルはストリームを閉じた時点で書き換えられてしまいます. この関数は LISP の open 関数を利用しています.

opena 関数は openw 関数に似ていますが,既存のファイルに対して末尾にストリーム出力を追加する点で異なります. この関数も LISP の open 関数を利用しています.

openr 関数は入力ストリームとしてファイルを開く関数です. ファイルが存在しない場合はエラーになります. この関数は LISP の open 関数を利用していますが,他と違い,open 関数のオプションを設定していません.

close 関数はストリームを閉じる関数です. LISP の close 関数をそのまま利用しただけの関数です. 猶,ファイルへの出力を伴うストリームの場合,このストリームが閉じ

られた時点で、ファイルの内容が更新されます。

—— ストリーム処理の関数 ——

```
make_string_input_stream(< 文字列 >)  
make_string_output_stream()  
get_output_stream_string(< ストリーム >)  
fposition(< ストリーム >)  
fposition(< ストリーム >, < 正整数値 >)  
flength(< ストリーム >)  
readline(< ストリーム >)
```

`make_string_input_stream` 関数は LISP の `make-string-input-stream` 関数を利用した関数です。引数の文字列から入力用のストリームを生成します。ここで引数は Maxima の文字列か LISP の文字列型でなければなりません。猶、LISP の `with-input-from-string` は `make_string_input_stream` の様に Maxima には実装されていません。

`make_string_output_stream` 関数は LISP の `make-string-output-stream` 関数を利用した関数で、出力ストリームを開きます。猶、LISP の `with-output-from-string` は `make_string_output_stream` の様に Maxima で実装されていません。

`flength` 関数は指定したストリームの成分数を返す関数です。この関数は LISP の `file-length` 関数をそのまま利用している関数です。その為、ファイルからの出力ストリームに対してのみ利用可能です。

`fposition` 関数は `file-position` 関数を利用した関数で、ストリーム上でポインタを移動させたり、そのポインタの位置を返したりする関数です。先ず、引数がストリームだけの場合は、ストリームの頭からのポインタの位置を返します。この場合、1 がストリームの先頭である事を示します。次に、引数がストリームと正整数値が指定された場合には、正整数値で指定した箇所にポインタを移動させます。勿論、指定可能な正整数値の範囲は、1 から `flength` で求めたストリームの長さに限定されます。

`readline` 関数は指定したストリームから一行ずつ読取を行う関数です。

—— ストリームに対して出力を行う関数 ——

```
freshline()
freshline(< ストリーム >)
newline()
newline(< ストリーム >)
printf(< ストリーム >, < Maxima の文字列 >)
printf(< ストリーム >, < Maxima の文字列 >, < オプション >)
```

freshline 関数は LISP の fresh-line 関数を実装したものです. この関数は指定したストリーム, 引数が無い場合には標準出力に改行を出力します.

newline 関数は LISP の terpri 関数を実装したものです. 引数が無い場合には, 標準出力に, ストリームの指定があれば, 指定されたストリームに対して改行を行います.

printf 関数は Common LISP の format 関数を Maxima で利用する為の関数です. 関数名が C 風なのが面白い事ですが, 書式は C ではなく LISP の format 関数の書式に準じます.

—— 数値の書式 ——

```
~$   お金の表記
~d   十進整数
~b   二進整数
~o   八進整数
~x   十六進整数
~br  b 進整数
~f   浮動小数
~e   科学的表記
~g   ~f か ~e を数値の大きさに切替える
~r   整数を英語で表記
~p   複数形
```

先ず, 数値に対して試してみましょう.

```
(%i11) printf(true, "~d:~%", 128);
128:
(%o11)                                     false
(%i12) printf(true, "~b:~%", 128);
10000000:
```

```

(%o12) false
(%i13) printf(true,"~o:~%",128);
200:
(%o13) false
(%i14) printf(true,"~x:~%",128);
80:
(%o14) false
(%i15) printf(true,"~7r:~%",128);
242:
(%o15) false
(%i16) 2*7^2+4*7+2;
(%o16) 128
(%i17) printf(true,"~r:~%",128);
one hundred and twenty-eight:
(%o17) false
(%i18) printf(true,"~f:~%",128);
128.0:
(%o18) false
(%i19) printf(true,"~e:~%",128);
1.2799999f+2:
(%o19) false
(%i20) printf(true,"~g:~%",128.0);
128.    :
(%o20) false
(%i21) printf(true,"~e:~%",128.0);
1.28E+2:
(%o21) false
(%i22) printf(true,"~g:~%",128.0);
128.    :
(%o22) false

```

~r の場合に 128 が one hundred and twenty-eight となっているのは面白い事でしょう。
又、数値に関しては表示枠の指定と右寄、左寄の指定が行えます。

```

(%i55) printf(true,"~16,8f:~%",128.0);
128.00000000:

```

```
(%o55)                                     false
(%i56) printf(true,"~-16,8f:~%",128.0);
128.000000000:
(%o56)                                     false
```

この例から判る様に,~16,8f で 128.0 を右詰めで表示し, ~-16,8f で左詰めになります.C と異なる点は,16.8f ではなく,16,8f の様に表記する点です. 紛らわしいので注意が必要です.

又, 数値に関する書式で面白いのが,~r と~p です. ~r は整数を英語で置換え,~p は 1 以外に対しては s を返すものです.

```
(%i70) dog:3;printf(true,"~r 柴犬~p are same color.~%",dog,dog);
(%o70)                                     3
three 柴犬 s are same color.
(%o71)                                     false
```

次に文字列に関する書式があります.

文字列出力に関する書式

```
~%    改行 (newline)
~&    改行 (fresh line)
~t    タブ (tab)
~a    Maxima の print 函数と同様に文字列の中身を表示
~s    文字列を二重引用符で括って出力する.
~~~   ~ を表示する.
```

既に,~%は数値の例題に出っていますが,C の \n に相当します. 又,~a と~s の違いは出力が二重引用符で括られるかどうかという点です. 以下の例を見ると明瞭でしょう.

```
(%i66) printf(true,"最初はね~a だけど、次は~s だよ~%", "~a", "~s");
最初はね~a だけど、次は"~s"だよ
```

printf 文の書式には同じ書式を反復して利用する事の様に, 出力を制御する事が可能になっています.

—— 制御に関連する書式 ——

```

~< justification, ~>で閉じる.
~( case conversion, ~)で閉じる.
~[ selection, ~]で閉じる.
~{ リストで与えられた引数に対し, 反復処理を行う~}で閉じる

```

6.11.3 stringproc パッケージの真理関数

ここでは stringproc パッケージに付属する真理関数について解説します.

先ず, 引数が一つだけの真理関数について解説します. これらの関数は与えられた対象が目的の対象 (Maxima の文字列等) に合致する場合に true を返し, それ以外は false を返す関数です.

—— 真理関数 ——

```

lcharp(< 対象 >)
charp(< 対象 >)
stringp(< 対象 >)
lstringp(< 対象 >)
constituent(< 対象 >)
alphanumericp(< 対象 >)
lowercasep(< 対象 >)
uppercasep(< 対象 >)
digitcharp(< 対象 >)

```

lcharp 関数は LISP の characterp 関数に引数をそのまま引渡す関数で, 与えられた対象が LISP 型の文字の場合に true を返す関数です. この関数は内部的に LISP の character 関数を用いています. 猶, Maxima の長さが 1 の文字列を与えても false が返されます.

charp 関数は, 引数が Maxima の文字の場合に true を返す関数です. 内部では引数が Maxima の文字列である事を確認し, その文字列の長さが 1 の場合を Maxima の文字としています.

lstringp 関数は LISP の stringp 関数に引数をそのまま引渡す関数で, 与えられた対象が LISP 型の文字列の場合に true を返す関数です.

stringp 関数は, 引数が Maxima の文字列の場合に true を返す関数です.

consistent 関数は引数が画面上で表示される文字の何れかで空行でない場合に true を返す関数です。猶、この関数の実体は [12] の P.61(原書では P.67) で定義されている constituent 関数そのもので、LISP の graphic-char-p 関数と char=関数を用いています。

alphachar-p 関数は、引数が LISP の文字型であり、且つ、アルファベットの場合に true を返す関数です。この関数は alpha-char-p の Maxima への実装になります。

digit-char-p 関数は数の場合にのみ true を返す関数です。この関数は、内部で char-int 関数を用いて引数の文字を ASCII データの数値に変換し、その数値が 47 よりも大きく、58 よりも小さい場合に true を返します。

alphanumeric-p 関数は名前から判る様に、引数がアルファベットか数の場合に true を返す関数です。この関数は alphanumeric-p 関数の Maxima への実装に他なりません。

lowercase-p 関数は引数が小文字の場合に、uppercase-p 関数は引数が大文字の場合に各々 true を返す関数です。各々 lower-case-p 関数と upper-case-p 関数の Maxima への実装になります。

次に、引数が二つの真理関数を以下に纏めておきます。

—— 引数が二つの真理関数 ——

```
cequal(< 対象1>, < 対象2>)
cequalignore(< 対象1>, < 対象2>)
cless(< 対象1>, < 対象2>)
clessignore(< 対象1>, < 対象2>)
cgreaterp(< 対象1>, < 対象2>)
cgreaterpignore(< 対象1>, < 対象2>)
```

基本的にこれらの関数は二つの引数の比較を行い、ある条件に結果が合致すれば true を返し、それ以外の場合は false を返す関数です。猶、これらの関数には対応する LISP の関数が存在し、実質的に、LISP の関数の Maxima への実装になっています。

先ず、cequal 関数は LISP の char=関数に対応する関数で、二つの Maxima の文字が等しい場合に true を返し、そうでない場合には false を返します。

cequalignore 関数は LISP の char-equal 関数に対応する関数で、二つの Maxima の文字が等しい場合に true を返し、そうでない場合には false を返します。

clessp 関数は LISP の char<関数に対応する関数で、二つの Maxima の文字を ASCII コードで比較する関数です。基本的に第一引数の ASCII コードが第二引数の ASCII コードよりも小さい場合に true を返します。

clesspignore 関数は LISP の char-lessp 関数に対応する関数で、二つの Maxima の文

字を ASCII コードで比較する関数です. 基本的に第一引数の ASCII コードが第二引数の ASCII コードよりも小さい場合に true を返します.

cgreaterp 関数は LISP の char< 関数に対応する Maxima の関数で, 二つの Maxima の文字を ASCII コードで比較する関数です. 基本的に第一引数の ASCII コードが第二引数の ASCII コードよりも大きい場合に true を返します.

cgreaterignore 関数は LISP の char-greaterp 関数に対応する関数で, 二つの Maxima の文字を ASCII コードで比較する関数です. 基本的に第一引数の ASCII コードが第二引数の ASCII コードよりも大きい場合に true を返します.

文字列に対しても同様の関数があります. 但し, これらの関数は cequal 関数と cequalignore 関数の文字列版に対応する関数だけです.

文字列の真理関数

```
sequal(< 文字列1>, < 文字列2>)
sequalignore(< 文字列1>, < 文字列2>)
```

sequal 関数と sequalignore 関数は二つの文字列が等しい場合に true を返す関数です. 各々が LISP の string= 関数と string-equal 関数の Maxima への実装となっています.

6.11.4 文字列変換の関数

以下に LISP と Maxima の間での文字列の変換関数等を纏めておきます.

変換関数

```
cunlisp(< LISP の文字型 >)
sunlisp(< LISP の文字列 >)
lstring(< Maxima の文字列 >)
cint(< Maxima の文字 >)
ascii(< Maxima の整数値 >)
```

cunlisp 関数は LISP の文字型データを Maxima の文字列型のデータに変換する関数です. これに対し, sunlisp 関数は LISP の文字列型データを Maxima の文字列型データに変換する関数になります.

```
(%i43) to_lisp();
```

Type (to-maxima) to restart, (\$quit) to quit Maxima.


```

MAXIMA> (setf $neko1 #\a)
#\a
MAXIMA> (setf $neko2 "aBcD")
"aBcD"
MAXIMA> (to-maxima)
Returning to Maxima
(%o43)                                     true
(%i44) cunlisp(neko1);
(%o44)                                     a
(%i45) sunlisp(neko2);
(%o45)                                     aBcD
(%i46) lstringp(neko1);
(%o46)                                     false
(%i47) lstringp(neko2);
(%o47)                                     true

```

lstring 関数は Maxima の文字列を LISP の文字型に変換する関数です. この関数は単純に Maxima 内部で Maxima の文字列を LISP の文字列に変換する l-string 関数に Maxima の文字列を引渡すだけの関数です.

cint 関数は Maxima の長さが 1 の文字列を ASCII コードの数値に変換する関数です. LISP の char-int 関数を実装したものです. これに対し,ascii 関数は cint 関数の逆操作を行う関数で,与えられた整数値から ASCII コードの数値に対応する Maxima の文字型データを返す関数です.

```

(%i33) map(cint,["a","1"]);
(%o33)                                     [97, 49]
(%i34) map(ascii,[40,87]);
(%o34)                                     [(, w]

```

6.11.5 文字列操作の関数

文字操作の関数

```
scopy(< 文字列 >)
smake(< 整数値 >, < 文字列 >)
charat(< 文字列 >, < 整数値 >)
charlist(< 文字列 >)
slength(< 文字列 >)
parsetoken(< 文字列 >)
sconc(< 文字列1 >, ·, s< 文字列n >)
sposition(< 文字 >, < 文字列 >)
sreverse(< 文字列 >)
strim(< 文字列1 >, < 文字列2 >)
```

scopy 関数は LISP の copy-seq を用いる関数で、与えられた Maxima の文字列を複製する関数です。

smake 関数は LISP の make-string 関数を用いる関数で、Maxima の文字を指定した整数個並べて新しい文字列を生成する関数です。例えば、`smake(3, "a")` から Maxima の文字列 `aaa` が得られます。

charat 関数は与えられた Maxima の文字列から指定した整数で表現される位置にある文字を取り出します。因に、文字列の先頭 (左端) が 1 で、文字列の末尾が `slength` 関数で得られる値、即ち、文字列の文字数になります。例えば、`charat("explode", 2)` を実行すると、与えた文字列 `explode` の先頭から二番目の文字 `x` が返されます。因に、この関数は LISP の `subseq` 関数の Maxima への実装になります。

charlist 関数は与えられた Maxima の文字列を分解してリストデータに変換する関数です。例えば、`charlist("explode")` は `[e, x, p, l, o, d, e]` に変換されます。LISP に依存すると思いますが日本語も可能です。

```
(%i53) charlist("explode");
(%o53)          [e, x, p, l, o, d, e]
(%i54) charlist("爆発");
(%o54)          [爆, 発]
```

slength 関数は与えられた Maxima の文字列の長さを返す関数です。この関数の実体は、引数を LISP の文字型に変換し、`length` 関数を作作用させているだけです。

parsetoken 関数は与えられた文字列が数と空行, コンマ, セミicolon, タブや改行で構成されている場合に, 先頭の数値部分を数値に変換する関数です.

```
(%i106) parsetoken("1;234");
(%o106)                                     1
(%i107) parsetoken("1,234");
(%o107)                                     1
(%i108) parsetoken("1 234");
(%o108)                                     1
(%i109) 1/parsetoken("1234");
(%o109) -----
1234
```

sconc 関数は Maxima-5.14.0 より concat 関数をそのまま利用する関数に改められています. これは, Maxima-5.14.0 より Maxima の文字列の扱いが Maxima 独自のものから, LISP の文字列に統合された為です.

sposition 関数は第一引数で指定した Maxima の文字が第二引数の文字列上で何処で最初に表われているかを返す関数です.

```
(%i67) sposition("c","cCcC");
(%o67)                                     1
```

sreverse 関数は Maxima の文字列の順番を逆にする関数です. LISP の reverse 関数の Maxima への実装です.

```
(%i1) sreverse("うえからよんでも山本山");
(%o1)                                     山本山もでんよらかえう
```

sinsert 関数は指定した位置に文字列を挿入する関数です.

```
(%i18) sinsert(", ","絶景かな絶景かな",5);
(%o18)                                     絶景かな、絶景かな
```

strim 関数, triml 関数と trimr 関数は共に LISP の string-trim, string-left-trim, string-right-trim 関数を Maxima に実装した関数です. 共に, 第二引数〈文字列₂〉から特定の位置にある第一引数〈文字列₁〉を除いた文字列を返す関数です.

これらの関数は〈文字列₂〉に指定した文字列〈文字列₁〉複数含まれている場合、一つだけを削除し、全ての〈文字列₁〉を削除する事はしません。

先ず, `striml` 関数は〈文字列₁〉から開始する文字列〈文字列₂〉に対し、〈文字列₁〉を取り除いた部分を返します。もし、〈文字列₂〉が〈文字列₁〉から開始していない場合には、〈文字列₂〉をそのまま返却します。

`strimr` 関数は逆に〈文字列₁〉を末尾に持つ文字列〈文字列₂〉に対し、〈文字列₁〉を除いた頭の部分を返却します。

`strim` 関数はこれらの `striml` 関数と `strimr` 関数の両方の働きをします。猶、`strim("abc", "123abc123")` を実行すると、真中の `abc` を抜いた文字列を返すのではなく、`"123abc123"` をそのまま返却します。

`substring` 関数は与えられた Maxima の文字列から指定した位置の Maxima の文字列を取り出す関数です。この関数は LISP の `subseq` 関数の Maxima への実装になります。

substring 関数

```
substring(〈文字列〉, 〈正整数〉)
substring(〈文字列〉, 〈正整数1〉, 〈正整数2〉)
```

```
(%i2) substring("うえからよんでも山本山",9);
(%o2)                                     山本山
(%i3) substring("うえからよんでも山本山",3,9);
(%o3)                                     からよんでも
```

split 関数

```
split(〈文字列〉)
split(〈文字列〉, 〈分離文字〉)
split(〈文字列〉, 〈分離文字〉, false)
```

`split` 関数は与えられた文字列を分解したリストを返す関数です。猶、切れ目になる文字はデフォルトでは空行 (space) ですが、別途指定する事が可能です。又、末尾に `false` を指定した場合、区切文字が複数存在する場合に、空行を入れたリストを返します。

```
(%i53) split("first  third fourth", " ", false);
(%o53) [first, , third, fourth]
(%i54) split("first  third fourth", " ");
(%o54) [first, third, fourth]
```

simplode 関数

```
simplode(< リスト >)
simplode(< リスト > < 文字列 >)
```

simplode 関数は explode 関数の逆操作の関数で、与えられたリストから文字列を生成する関数です。

引数が Maxima のリストのみの場合、リストの成分を繋げた文字列を返します。第二引数として Maxima の文字列を指定した場合、リストの成分を繋げる際に、第二引数の文字列を間に挿入します。

```
(%i53) simplode(["12","34","56"]);
(%o53)                      123456
(%i54) simplode(["12","34","56"],"*-*");
(%o54)                      12*-*34*-*56
```

この例では、第二引数に何も指定しない場合と、第二引数を指定した場合の違いを示しています。第二引数を指定しない場合には、単純に文字リストの成分を結合するのですが、第二引数として"*-*"を指定すると、第一引数のリストの各成分にこの文字列を挿入している事が判るかと思います。

supcase 関数は与えられた文字列に対し、指定された範囲のアルファベットを大文字に変換する関数で、後述の sdowncase の逆操作になります。

supcase 関数

```
supcase(< 文字列 >)
supcase(< 文字列 >, < 正整数値 >)
supcase(< 文字列1 >, < 正整数値1 >, < 正整数値2 >)
```

```
(%i44) supcase("this is a pen.",1,1);
(%o44)                      this is a pen.
(%i45) supcase("this is a pen.",1);
(%o45)                      THIS IS A PEN.
(%i46) supcase("this is a pen.");
(%o46)                      THIS IS A PEN.
(%i47) supcase("this is a pen.",5);
(%o47)                      this IS A PEN.
```

```
(%i48) supcase("this is a pen.",1,2);
(%o48)          This is a pen.
```

sdowncase 関数は与えられた文字列に対し、指定された範囲のアルファベットを小文字に変換する関数です。この関数は supcase の逆操作になります。

— sdowncase 関数 —

```
sdowncase(< 文字列 >)
sdowncase(< 文字列 >, < 正整数値 >)
sdowncase(< 文字列1 >, < 正整数値1 >, < 正整数値2 >)
```

```
(%i53) sdowncase("此の猫の名前は MIKE です");
(%o53)          此の猫の名前は mike です
(%i54) sdowncase("此の猫の名前は MIKE です",9);
(%o54)          此の猫の名前は Mike です
(%i55) sdowncase("此の猫の名前は MIKE です",9,11);
(%o55)          此の猫の名前は MikE です
```

sinvertcase 関数は与えられた文字列に対し、指定された範囲でアルファベットの大小を逆に変換する関数です。

— sinvertcase 関数 —

```
sinvertcase(< 文字列 >)
sinvertcase(< 文字列 >, < 正整数値 >)
sinvertcasee(< 文字列1 >, < 正整数値1 >, < 正整数値2 >)
```

```
(%i58) sinvertcase("此の猫の name は MIKE です");
(%o58)          此の猫の NAME は mike です
(%i59) sinvertcase("此の猫の name は MIKE です",5,6);
(%o59)          此の猫の Name は MIKE です
```

6.11.6 真理関数を利用する文字列操作の関数

tokens 関数

```
tokens(< 文字列 >)
tokens(< 文字列 >, < 真理関数 >)
```

tokens 関数は [12] の P.61 にある tokens を取り込んだものです。真理関数は省略可能ですが、この場合は constituent 関数が用いられます。この関数は与えられた文字列を分解して、Maxima のリストに変換する関数です。この分解で真理関数を用います。具体的には真理関数で true になる部分のみが Maxima のリストの成分として返されます。

```
(%i79) tokens("MIKEandTAMAand123");
(%o79) [MIKEandTAMAand123]
(%i80) tokens("MIKEandTAMAand123", constituent);
(%o80) [MIKEandTAMAand123]
(%i81) tokens("MIKEandTAMAand123", lowercase);
(%o81) [and, and]
(%i82) tokens("MIKEandTAMAand123", uppercase);
(%o82) [MIKE, TAMA]
(%i83) tokens("MIKEandTAMAand123", digit);
(%o83) [123]
```

この例で真理関数を指定しない場合と constituent 関数を指定した時の結果は同じものになります。又、lowercase 関数を真理関数として指定した場合、lowercase 関数が true を返す and の部分のみが返却されています。それに対し、uppercase 関数を指定すると大文字の部分のみ、digit を指定した場合には、数値の部分のみが返却されています。

ssubstfirst 関数は文字列に指定した真理関数に適合する部分文字列が存在する場合に、部分文字列の入れ替えを行う関数です。

ssubstfirst 関数

```
ssubstfirst(< 文字列1>, < 文字列2>, < 文字列3>)
ssubstfirst(< 文字列1>, < 文字列2>, < 文字列3>, < 真理関数 >)
ssubstfirst(< 文字列1>, < 文字列2>, < 真理関数 >, < 正整数値 >)
ssubstfirst(< 文字列1>, < 文字列2>, < 真理関数 >, < 正整数値1>, < 正整数値2>)
```

名前の通り, 真理関数に適合する部分が複数存在する場合には, 一番最初の部分が取り換えられます. 又, 真理関数が未指定の場合は `sequal` 関数が用いられます. 末尾の正整数値は真理関数による評価を行う為の領域指定を行う為に用いられます.

```
(%i13) ssubstfirst("三毛","虎","虎猫");
(%o13)                三毛猫
(%i14) ssubstfirst("三毛","虎","虎は虎猫ではない");
(%o14)                三毛は虎猫ではない
(%i15) ssubstfirst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequal,5);
(%o15)                虎は虎猫ではないが、三毛キチでもない
(%i16) ssubstfirst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequal,5,9);
(%o16)                虎は虎猫ではないが、虎キチでもない
```

猶, 領域指定の整数を引数として渡す場合には必ず真理関数を記述しなければなりません.

この `ssubstfirst` 関数と似た関数で, 真理関数に合致する部分を全て入れ替えてしまうのが `ssubst` 関数です.

ssubst 関数

```
ssubst(< 文字列1>, < 文字列2>, < 文字列3>)
ssubst(< 文字列1>, < 文字列2>, < 文字列3>, < 真理関数 >)
ssubst(< 文字列1>, < 文字列2>, < 真理関数 >, < 正整数値 >)
ssubst(< 文字列1>, < 文字列2>, < 真理関数 >, < 正整数値1>, < 正整数値2>)
```

この関数で真理関数を未指定の場合は `equalignore` 関数が用いられます.

```
(%i19) ssubst("三毛","虎","虎猫");
```



```
(%o19)                                三毛猫
(%i20) ssubst("三毛","虎","虎は虎猫ではない");
(%o20)                                三毛は三毛猫ではない
(%i21) ssubst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequalignore)
;
(%o21)                                三毛は三毛猫ではないが、三毛キチでもない
(%i22) ssubst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequalignore,5);
(%o22)                                虎は虎猫ではないが、三毛キチでもない
(%i23) ssubst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequalignore,5,9);
(%o23)                                虎は虎猫ではないが、虎キチでもない
```

ssubst 関数でも領域指定の整数を引数として渡す場合には必ず真理関数を記述しなければなりません。

sremovefirst 関数は文字列に指定した真理関数に適合する部分文字列が存在する場合に、その部分文字列を削除する関数です。

— sremovefirst 関数 —

```
sremovefirst(< 文字列1>,< 文字列2>)
sremovefirst(< 文字列1>,< 文字列2>,< 真理関数 >)
sremovefirst(< 文字列1>,< 文字列2>,< 真理関数 >,< 正整数値 >)
sremovefirst(< 文字列1>,< 文字列2>,< 正整数値1>,< 正整数値2>)
```

名前の通り、真理関数に適合する部分が複数存在する場合、一番最初の部分が削除されます。又、真理関数が未指定の場合は sequal 関数が用いられます。末尾の正整数値は判定を行う為の領域の指定を行う為に用いられます。

```
(%i28) sremovefirst("さようなら","さようなら-ああ、さようなら",sequal,5,7);
(%o28)                                さようなら-ああ、さようなら
(%i29) sremovefirst("さようなら","さようなら-ああ、さようなら");
(%o29)                                -ああ、さようなら
(%i30) sremovefirst("さようなら","さようなら-ああ、さようなら",sequal);
(%o30)                                -ああ、さようなら
```

```
(%i31) sremovefirst("さようなら","さようなら-ああ, さようなら",sequal,3);
(%o31)          さようなら-ああ,
(%i32) sremovefirst("さようなら","さようなら-ああ, さようなら",sequal,5,7);
(%o32)          さようなら-ああ, さようなら
```

猶, 領域指定の整数を引数として渡す場合には必ず真理関数を記述しなければなりません.

sremove 関数は文字列に指定した真理関数に適合する部分文字列が存在する場合に, その部分文字列を全て削除する関数です.

— sremove 関数 —

```
sremove(< 文字列1>, < 文字列2>)
sremove(< 文字列1>, < 文字列2>, < 真理関数 >)
sremove(< 文字列1>, < 文字列2>, < 真理関数 >, < 正整数値 >)
sremove(< 文字列1>, < 文字列2>, < 正整数値1>, < 正整数値2>)
```

名前の通り, 真理関数に適合する部分が複数存在する場合, 全ての部分列が削除されます. 又, 真理関数が未指定の場合は sequalignore 関数が用いられます. 末尾の正整数値は評価を行う為の領域の指定を行う為に用いられます.

```
(%i33) sremove("さようなら","さようなら-ああ, さようなら");
(%o33)          -ああ,
(%i34) sremovefirst("さようなら","さようなら-ああ, さようなら",sequal,3);
(%o34)          さようなら-ああ,
(%i35) sremovefirst("さようなら","さようなら-ああ, さようなら",sequal,3,5);
(%o35)          さようなら-ああ, さようなら
```

猶, 領域指定の整数を引数として渡す場合には必ず真理関数を記述しなければなりません.

ssort 関数は与えられた文字列を指定した真理関数で大小関係を判別して並び換える関数です. 真理関数が未指定の場合は, clessp 関数が用いられます.

— ssort 関数 —

```
ssort(< 文字列 >)
ssort(< 文字列 >, < 真理関数 >)
```

```
(%i63) ssort("bkavbhjsaAA");
(%o63)          AAAabbhjksv
(%i64) ssort("bkavbhjsaAA",cgreaterp);
(%o64)          vskjhbbaaAA
(%i65) ssort("いろはにほへと",cgreaterp);
(%o65)          ろほへはにとい
(%i66) ssort("いろはにほへと",clessp);
(%o66)          いとにはへほろ
```

真理関数を clessp 関数から cgreaterp 関数に変更する事で結果が逆になる事が判ります. 更に, 日本語も扱えます.

smismatch 関数は二つの文字列を指定した真理関数を用いて比較し, 不適合が出る位置を返します. 真理関数が未指定の場合, ssequal 関数が用いられます.

smismatch 関数

```
smismatch(< 文字列1>, < 文字列2>)
smismatch(< 文字列1>, < 文字列2>, < 真理関数 >)
```

```
(%i72) smismatch("桃も李も桃の内", "桃も李も藻も桃も");
(%o72)          5
(%i73) smismatch("1234123", "2345123", clessp);
(%o73)          5
(%i74) smismatch("1234123", "2345123", cgreaterp);
(%o74)          1
```

最初の例では真理関数を未指定の為, ssequal が用いられます. その為, 両者が初めて異なる藻の位置が返されています. 次の例では, 真理関数を clessp にした場合です. この場合 5 番目から大小関係が逆になるので 5 が返されていますが, 最後に cgreaterp を指定すると最初から不適合になる為, 1 が返されています.

ssearch 関数は第一引数の文字列が第二引数の文字列に含まれる場合に, その開始位置を返します. 又, 真理関数を用いて第二引数の文字列の部分列で, 第一引数の文字列との関係を満す箇所があれば, その開始位置を返します. 真理関数を指定する場合に限って, 評価を行う際の第二引数の文字列の開始位置や終了位置を指定する事も可能です. 又, 真理関数が未指定の場合, 内部では sequalignore 関数が用いられています.

ssearch 関数

```

ssearch(< 文字列1>, < 文字列2>)
ssearch(< 文字列1>, < 文字列2>, < 真理関数 >)
ssearch(< 文字列1>, < 文字列2>, < 真理関数 >, < 正整数値 >)
ssearch(< 文字列1>, < 文字列2>, < 真理関数 >, < 正整数値1>, < 正整数値2>)

```

```

(%i1) ssearch("猫","三毛猫と虎猫");
(%o1)                                     3
(%i2) ssearch("A","AAAABCD",clessp);
(%o2)                                     5
(%i3) ssearch("A","AAAABCD",clessp,4);
(%o3)                                     5
(%i4) ssearch("A","AAAABCD",clessp,7,8);
(%o4)                                     7
(%i5) ssearch("A","AAAABCD",clessp,8);
(%o5)                                     false

```

6.11.7 関連する大域変数

stringproc パッケージで定義されている大域変数は実はあまりありません。基本的に、stringprocs で定義されている関数で大域変数を殆ど用いない事が影響していますが、こうなる理由も基本的に LISP にある関数を Maxima 上で実現させる事を目的にした為、関数を制御する大域変数がそんなに必要ではなかった事も挙げられるでしょう。その為、関連する大域変数としては、改行、タブや空行を表現する newline, tab, space 程度しかありません。

関連する大域変数

```

newline  改行
tab       タブ
space     空行

```

第7章 プログラム

Faust

Faust,Faust,nun erfüllt sich dein Augenblick!
Die Zaubermacht in meine Hand gegeben,
die ungeheuren Zeichen mir erschlossen,
heimliche Gewalten mir geknechtet,
und ich kann -ja,ich kann -o,ihr Menschen,die ihr mich gepeinigt, hütet euch vor Faust!

ファウスト

ファウストよ、ファウストよ、ついにお前の望が叶ったぞ!
魔力が我が手に与えられ、
おぞましい象徴が明らさまとなり、秘密の威力が我が物となった、
そして、俺には出来る、そうだ、俺には出来る、
おお、人間共よ、俺を苦しめたお前達よ、ファウストを恐れるがよい!

Buzoni,Doktor Faust[58] より

この章では,Maxima のプログラムに関連した事柄について解説を行います. 更に,Maxima の LISP に関連する事項と, ファイルの入出力についても述べます.

Maxima 上でのプログラムでは,Maxima の PASCAL 風の処理言語を用います. この処理言語を用いる事で, 利用者定義の Maxima の関数を構築する事も可能です.

Maxima 言語で記述された利用者定義関数は, 実行時には LISP で解釈され,LISP で処理されます. その為,LISP による解釈で速度の遅延が生じます. この為, 利用者定義関数を LISP の関数に変換する `translate` 関数や,LISP のコンパイラを利用して高速化を図る `optimize` 関数があります.

7.1 Maxima でプログラム

Maxima には制御文として,if 文, 反復処理に do ループや go といった, 原始的な構文があります. 又,block 文を用いる事で局所変数を利用する事も出来ます.

Maxima の処理言語は C や PASCAL の様な手続型の言語に見えます. しかし,LISP 上で動作する為に, データの内部表現を利用した方が効率的なプログラムが記述し易い側面もあります.

Maxima には compile 函数等で,Maxima の処理言語で記述した函数をコンパイルする手段もあり, それによってある程度の速度向上も見込めます, それでも,Maxima の処理言語で記述したプログラムを LISP で解釈して実行する手間もある為, 直接, LISP で記述する方が速度的には有利です.

7.1.1 block 文

block 文

```
block(<変数リスト>, <式1>, <式2>, ..., <式n>)
```

Maxima の block は FORTRAN の subroutine,PASCAL の procedure に似たものです. block 文は文の集合体ですが,block 内部の文にラベル付けが行え, 局所変数も扱えます. 局所変数は block 文外部にある同名の大域変数との名前の衝突を避ける事に使えます. 同名の大域変数が存在した場合,block 文の実行中, その変数はスタックに保存されるので, その間は参照出来ません.block 文が終了した時点で, スタックに保存されていた値がもとに戻されますが,block 文の局所変数の値は破棄されてしまいます.

猶,局所変数を用いない場合には,局所変数のリストを省略したり, 空のリスト [] で置換えても構いません. 更に, 局所変数に初期値を設定する事も可能です. この場合, [a:1,b:2] の様に局所変数に対してコロン (:) による通常の割当を行います.

ここで,block 内部で用いられているものの,block 文の局所変数リストに含まれていない変数は block 文の外部で用いられている変数と同様に大域変数として扱われます. その為, 値は block 文の終了後も保持されます.

block の値は, 最後の文の値か block から return 函数に渡された引数の値となります. 函数 go は制御を go の引数でラベル付けされた block 内の文に移動する為に使えます. 文のラベル付けは,block 内部で, 原子を文の前に置く事で対処します. 例えば,block(<式₁>, ..., <式_n>, loop,<式>, ...,go(loop),...) の様に用います. go 函数の引数は go 函

数を含む block 内部に現われるラベル名でなければなりません. go を含まない block 文中のラベルに go で移動する事は出来ません.

return 関数は block 文の変数リスト以外の何処にでも置けます.return を置かなかった場合には,block 文の末尾の式の値が返却値となります.

以下に block 文を用いた関数の例を示します. この関数では局所変数として, a,k を用いています.

```
(%i67) a:10;
(%o67)
10
(%i68) neko(x):=block([a:2,k],k:sin(x)*a,return(k));
(%o68) neko(x) := block([a : 2, k], k : sin(x) a, return(k))
(%i69) neko(10);
(%o69)
2 sin(10)
(%i70) a;k;
(%o70)
10
(%i71) k;
(%o71)
k
```

この例で示す様に, 局所変数は block 文内部でのみ利用され, 同名の変数には影響を与えていません.

7.1.2 if 文

if 文は条件分岐で用います. その構文は C 等の言語と違いはありません.

if 文の構文

```
if < 述語 > then < 式1 > else < 式2 >
```

if 文は < 述語 > を評価して,true であれば, < 式₁ > を実行し,false であれば, < 式₂ > を実行します.

< 述語 > はその値が true, 又は false であるかが評価可能な式で, 真偽関数や演算子等で構成されたものです.

if 文で利用可能な演算子

演算子	記号	種類
等しい	=, equal	中置式演算子 (infix)
等しくない	#	中置式演算子 (infix)
大きい	>	中置式演算子 (infix)
小さい	<	中置式演算子 (infix)
以上	>=	中置式演算子 (infix)
以下	<=	中置式演算子 (infix)
論理積	and	中置式演算子 (infix)
論理和	or	中置式演算子 (infix)
否定	not	前置式演算子 (prefix)

猶, 前置式演算子 (prefix) と中置式演算子 (infix) は, 演算子を引数の置き方で区分したものです. 詳細は演算子の節を参照して下さい.

これに対し, $\langle \text{式}_1 \rangle$ と $\langle \text{式}_2 \rangle$ は任意の Maxima の式 (勿論, if 文を含んで入れ子になっても構いません) が利用可能です.

7.1.3 do 文による反復処理

Maxima では, do 文を用いて反復処理を行います.

do 文の基本構造

```
for  $\langle$  制御変数処理  $\rangle$   $\langle$  終了条件  $\rangle$  do  $\langle$  本体  $\rangle$ 
```

do 文には, do 文内部のみで用いられる局所変数があり, これを制御変数と呼びます. この制御変数は do 文の中だけで効力を持ちます. それから, 終了条件の判定を行い, 終了条件を満たさなければ, do 文の本体を実行し, それから制御変数処理で, 制御変数に変更を加え, 再度, 終了条件の判定を行う事を反復します.

最初に $\langle$ 制御変数処理 $\rangle$ の箇所について述べます.

制御変数処理では, 最初に制御変数に初期値を割当て, 次に, 反復処理で再度回って来ると, 制御変数に新しい値を割当てます.

ここで, 最初の初期値の割当てでは, 次の二つの同値な書き方があります.

制御変数の初期値の割当

```

<変数> : <初期値>
<変数> from <初期値>

```

この制御変数の初期値の割当はどちらを用いても構いません。猶、初期値が 1 の場合、`<初期値>` や `from <初期値>` は省略可能です。

次に、制御変数を一定の値で増加させたり、減少させたいければ、`do` 文の内部に、`step <増分>` を追加します。この時、`<増分>` を正実数とすれば、通常の増分となり、負の実数とすれば、減少分になります。猶、`<増分>` が 1 の場合は `step 1` を省略する事が可能です。制御変数に何等かの関数を割当てたければ、`next <制御変数の式>` とすれば、`i` の値に `<制御変数の式>` で計算した値が制御変数に割当てられます。

例えば、以下の二つの反復処理は同値になります。

制御変数の割当方法

```

do i from 1 step 2 ...
do i:1 next i+2 ...

```

これらは、全て制御変数 `i` の初期値を 1、増分を 2 とする反復処理を実行します。最初の二つでは `step` を用いて増分を定めています。最後の `next` を用いたものでは、関数 `i+2` で制御変数の値を定めています。

これらの方法に加えて、リストを用いた制御変数の値の割当て方もあります。

リストを用いた制御変数の割当

```

for i in <リスト> ...

```

この場合、リストの成分には数値以外の関数や式を用いる事も可能です。

```
(%i10) for i in [1,2,3,4,5,6,7,8,9,10] do print(i);
```

```

1
2
3
4
5
6
7
8

```

```

9
10
(%o10)                                done
(%i11)for i in [sin,cos,tan] do print(subst(i,f,f(%pi/4)));
sqrt(2)
-----
      2
sqrt(2)
-----
      2
1
(%o17)                                done

```

この例では、最初にリスト [1,2,3,4,5,6,7,8,9,10] の元を表示し、最後の例では、 $f(\pi/4)$ の f に $\sin, \cos, \tan$ を順番に代入した結果を表示させています。

do 文の終了条件の与え方には、次の三種類があります。

— do 文の終了条件の与え方 —

thru 制御変数の境界値に達した時点で反復処理を終える。
 unless 条件を満たした時点で反復処理を終える。
 while 条件を満たさなくなった時点で反復を終える。

ここで、終了条件を与える式は、MAXIMA の述語、即ち、true か false かが判別可能な式です。unless を用いた do 文は、C 等の repeat-until 文、while を用いた do 文は while 文に相当します。

thru, unless と while を用いた例を以下に示します。猶、三種類ともに全て同じ反復処理、即ち、1 から 10 迄の数を表示して終了、を実行するのです。

— 終了条件の例 —

```

for i:1 thru 10 do print(i);
for i:1 while i <= 10 do print(i);
for i:1 unless i > 10 do print(i);

```

猶、通常の do 文によって返される値は原子の done です。関数 return を用いると、本体の中で do から早目に抜けて必要な値を与える事に使えます。block にある do 文中の return は do 文を出るだけで、block 全体から出る訳ではありません。同様に go 関数

も block 中の do 文から抜ける為に使ってはなりません.

—— プログラムに関連する大域変数 ——

変数名	初期値	概要
backtrace	[]	関数リスト
dispflag	true	block 文中の関数出力を制御
prederror	true	if 文や is 関数のエラーメッセージ出力を制御
errorfun	false	エラー発生時に起動させたい関数名

backtrace は `debugmode:all` の時に, 入力された関数全てのリストを値として持ちます.

dispflag が false ならば, block 文の中で呼ばれた関数の出力表示を禁止します. 記号 \$ のある block 文の末尾では dispflag を false に設定します.

prederror が true であれば, if 文や is 関数で, true か false であるか述語の評価に失敗すると, 何時でもエラーメッセージが表示されます.

false であれば, unknown が代りに返されます.

errorfun に引数を持たない関数名が設定されていれば, エラー発生時に, その関数が実行されます. この設定は batch ファイルで, エラーが生じた場合に Maxima を終了したり, 端末からログアウトしたい時に使えます.

—— block 文内部で利用する関数 ——

```

go(< ラベル >)
return(< 式 >)
break(< 引数 >, ...)
catch(< 式1 >, ..., < 式n >)
throw(< 式 >)

```

go 関数は block 内部で指定した block 文中のラベルに移動する事に用いられます. ラベルとして原子を用い, この原子は文の前に置きます.

```

block([x],
      x:1,
      loop,x+1,
      ...,
      go(loop),
      ...)

```

go の引数は同じ block の中で現われるラベルでなければなりません. go を含む別の block にあるラベルに go を用いて移動する事は出来ません.

return 関数は block 文から引数を伴って抜ける時に用います. 場所は block 文の何処に置いても構いません.

break 関数は、 $\langle$ 引数 $\rangle$ の評価と表示を行い、(maxima-break) にて利用者がその環境を調べ、変更する事が出来る様にします. (maxima-break) からは `exit;` を入力すれば計算が再開されます. 猶、Ctrl+a(^a) で maxima-break に何時も対話的に入る事が出来ます. Ctrl+x は maxima-break 内部で、本体側の処理を終了せずに、局所的に計算を止める事に用いても構いません.

catch 関数は throw と対で用います. これは非局所的回帰 (non-local return) で用いる関数で、最も近い throw に対応する catch に行きます. その為、throw に対応する catch が必ず必要で、そうでなければエラーになります. $\langle$ 式 $_i$ $\rangle$ の評価が何らの throw の評価に至らなかった場合、catch の値は最後の引数 $\langle$ 式 $_n$ $\rangle$ の値となります.

```
(%i51) g(l):=catch(map(lambda([x],
      if x<0 then throw(x) else f(x)),l));
(%o51) g(l) := catch(map(lambda([x], if x < 0 then
      throw(x) else f(x)), l))
(%i52) g([1,2,3,7]);
(%o52) [f(1), f(2), f(3), f(7)]
(%i53) g([1,2,-3,7]);
(%o53) - 3
```

関数 g は、l が非負の数のみであれば l の各要素に対する f のリストを返します. それ以外では、g は l の最初の負の要素を捉えて、それを放します.

throw 関数は $\langle$ 式 $\rangle$ を評価し、近くにある catch に値を投げ返します. throw は catch と一緒に使われます.

エラー処理を行う関数

```
errcatch( $\langle$  式 $_1$  $\rangle$ , ...,  $\langle$  式 $_n$  $\rangle$ )
error( $\langle$  引数 $_1$  $\rangle$ , ...,  $\langle$  引数 $_n$  $\rangle$ )
errormsg()
```

errcatch 関数は引数を一つずつ評価して、エラーが生じなければ最後の値のリストを返します.、任意の引数の評価でエラーが生じた場合、errcatch はエラーを捉えて即座に [] (空のリスト) を返します. この関数はエラーが生じていると疑われる batch ファイル

で, エラーを捉えなければ忽ち batch を終了させる様になると便利です.

error 関数は引数の評価と表示を行い, Maxima のトップレベルか, errcatch にエラーを返します. エラー条件を検知した場合や, Ctrl+^ が入力出来ない場所なら何処でも関数を中断させられるので便利です.

大域変数 error にはエラーを記述するリストが設定されており, 最初のものは文字列で, 残りの対象は問題を起しているものです.

errormsg 関数は最新のエラーメッセージを再表示します. 変数 error にはエラーを記したもののリストが設定されており, 最初は文字列で, 残りは問題の対象です.

ttyintfun:lambda([],errormsg(),print("")) で利用者中断文字 (^u) をメッセージの再表示を行う様に設定します.

7.1.4 構造体

Maxima には構造体もあります. 構造体は defstruct 関数で定義し, 対象の生成は new 関数を用います.

—— 構造体に関連する関数 ——

```
defstruct(< 構造体1 >, ..., < 構造体n >)
new(< 構造体 >)
< 構造体 > @ < 項目 >
```

先ず, Maxima の構造体は, mike(x,y,z..) の様に函数の様な書式になっています. この構造体の宣言では defstruct 関数を用います. この defstruct 関数では複数の構造体の宣言が行えます. この時, 引数として構造体の列を引数として与えます. すると, 大域変数 structures に割当てられたリストに構造体が登録されます.

—— 構造体に関連する大域変数 ——

```
structures  []  定義された構造体が格納されたリスト
```

構造体の生成は new 関数を用います. 引数は defstruct 関数で定義した構造体の名前になります.

ここで構造体の各項目に値を設定したり, 値の参照を行う為に, 演算子@ を用います. さて, 簡単な例題で各函数の動作を確認しておきましょう.

```
(%i1) structures;
```

```
(%o1)
```

```
[]
```

```
(%i2) defstruct(ペット(名前, 歳, 体重, 長さ, 性格), 本(著者, 分野, 出版社));
(%o2)          [ペット(名前, 歳, 体重, 長さ, 性格), 本(著者, 分野, 出版社)]
(%i3) structures;
(%o3)          [ペット(名前, 歳, 体重, 長さ, 性格), 本(著者, 分野, 出版社)]
(%i4) ウチのたま:new(ペット);
(%o4)          ペット(名前, 歳, 体重, 長さ, 性格)
(%i5) ウチのたま@名前:"たま"$
(%i6) ウチのたま@歳:1$
(%i7) ウチのたま@体重:5$
(%i8) ウチのたま@長さ:20$
(%i9) ウチのたま@性格:"温和。寂しがり屋さん"$
(%i10) ウチのたま;
(%o10)          ペット(名前 = たま, 歳 = 1, 体重 = 5, 長さ = 20, 性格 = 温和。寂しがり屋さん)
```

先ず, 最初に大域変数 `structures` の値を確認しています. この大域変数は最初は空リストが割当てられています. 次に, `defstruct` 関数で二つの構造体 (ペットと本) を宣言しています. 大域変数 `structures` に `defstruct` による影響が現われている事に注意して下さい.

構造体を利用する為には, `new` 関数を用います. ここではペットを用いる為, `new(ペット)` で生成した構造体を「ウチのたま」に割当てます.

それから, 各項目に値を設定しますが, ここでは演算子 `@` を用います. 基本的に `<構造体名>@<項目>` で構造体の項目の参照が行え, これを利用して演算子 `:` を用いて値の割当が行えます.

7.2 Maxima で関数の定義

7.2.1 関数定義

Maxima では、予めシステムが持っている関数の他に、利用者定義の関数が扱えます。Maxima の関数定義では演算子”:=”を用いる方法と define 関数を用いる方法があります。

関数の定義方法

```
define ( < 関数名 >, < ( < 引数1 >, ..., < 引数n > ) >, < 式 > )
< 関数名 > ( < 引数1 >, ..., < 引数n > ) := < 関数本体 >
```

基本的に define で定義する関数は block 文、if 文や do 文を含まない、一つの式のみで構成された関数を定義します。

一般的な関数の定義は演算子”:=”を用いた関数定義で関数を定義します。ここで、演算子”:=”を用いる関数定義で、その関数本体は式をコンマで区切ったものになります。関数の定義を行うと、大域変数 functions に定義した関数が追加されます。

大域変数 functions

変数名	初期値	概要
functions	[]	利用者定義関数のリスト

大域変数 functions は利用者が定義した全ての関数名を含むリストです。利用者が関数定義を行うと自動的に functions のリストに関数名が追加されます。functions に追加されるのは演算子:=の左辺となります。

```
(%i1) functions;
(%o1) []
(%i2) f(x):=sin(x);
(%o2) f(x) := sin(x)
(%i3) f(10);
(%o3) sin(10)
(%i4) functions;
(%o4) [f(x)]
```

この例では、関数 f(x) を定義すると、立ち上げ時には空リストであった functions に演算子:=の左辺が登録される為、利用者定義の関数 f(x) が追加されています。

関数定義の演算子:=を用いた関数で、最も簡単なものは、関数本体が式₁, 式₂, ..., 式_nの様に複数の式を並べたものです。この場合、関数の返却値は単純に最後の式の結果だけです。例えば、 $f(x) := (1-x, 2+x, 2*x)$ で関数 f を定義した場合、この関数の結果は最後の式 $2x$ を計算した値になります。

但し、この関数では割当が実行されていない為に、変数の内容の書換えは生じません。では、 $f(x) := (y:x, z:2+y, 2*z)$ と定義した関数を実行した場合はどうなるでしょうか？

```
(%i1) f(x):=(y:x,z:2+y,2*z);
(%o1)          f(x) := (y : x, z : 2 + y, 2 z)
(%i2) f(x);
(%o2)          2 (x + 2)
(%i3) y;
(%o3)          x
(%i4) z;
(%o4)          x + 2
(%i5) f(2);
(%o5)          8
(%i6) x;
(%o6)          x
(%i7) y;
(%o7)          2
(%i8) z;
(%o8)          4
```

この様に、先頭の式から順番に処理されて行き、最後の式の結果だけが返却されています。更に、この関数の場合、内部で用いた変数 y と z の値が書換えられている事に注意して下さい。

この様に、Maxima 内部で用いた変数は、通常は大域変数として扱われます。その為、変数の値の書換えが生じます。この事を避ける為に、Maxima では関数内部のみで有効な局所変数が扱えます。この局所変数は、後述の block 文の中で定義可能です。

Maxima の lambda 式を用いると無名の関数が構築出来ます。この lambda 関数の構文は最初に関数の変数を宣言し、その後に関数本体が続きます。関数本体は基本的に Maxima の式をコンマで区切ったものになります。

— lambda 関数 —

lambda($\langle$ 変数 $_1\rangle, \dots, \langle$ 変数 $_n\rangle, \langle$ 関数本体 $\rangle$)

構文自体は LISP の lambda 式と同様の構文となっています.

次の例では `lambda([i],2*i+1)` で引数 `i` に 1 を加える無名関数を構成し, その関数に `map` 関数でリスト `[1,2,3]` に作用させた結果と, lambda 式を利用した関数 `neko` を示しています.

```
(%i58) map(lambda([i],2*i+1),[1,2,3]);
(%o58) [3, 5, 7]
(%i59) neko(x):=map(lambda([i],sin(2*i+1)),x);
(%o59) neko(x) := map(lambda([i], sin(2 i + 1)), x)
(%i60) neko([1,2,3,4,5]);
(%o60) [sin(3), sin(5), sin(7), sin(9), sin(11)]
(%i61) i;
(%o61) i
```

ここで,lambda 関数内部で用いた疑似変数 `i` が lambda 関数内部のみで値を持つ事に注意して下さい.

Maxima では引数の個数が可変な関数も定義出来ます. この場合,最後の引数を特別な引数リストとして割当てて,関数を定義します.尚,引数が少ない場合,安易な処理を行っているとは少し問題になるかもしれません.

```
(%i41) f([u]):=u;
(%o41) f([u]) := u
(%i42) f(1,2,3,4,5);
(%o42) [1, 2, 3, 4, 5]
(%i43) f(a,b,[u]):=[a,b,u];
(%o43) f(a, b, [u]) := [a, b, u]
(%i44) f(1,2,3,4,5,6);
(%o44) [1, 2, [3, 4, 5, 6]]
(%i45) f(1,2);
(%o45) [1, 2, []]
```

単純に式を並べる方式では,関数の返却値は最後に処理された式の値となります. そうではなく,関数本体に含まれる式の返却値が必要な場合には,block 文と return 文を組合せます.

定義した関数の内容は関数 `dispfun` や `fundef` で参照する事が出来ます.

——— 定義した関数の内容を表示する関数 ———

```
dispfun(< 関数名1>, ..., < 関数名n> )
dispfun(all)
fundef(< 関数名 > )
```

利用者定義の関数 $\langle \text{関数名}_1 \rangle, \dots, \langle \text{関数名}_n \rangle$ の内容を表示します. この関数の表示では, 関数を定義した時点での関数や定数等がそのまま表示されます.

尚, 引数に `all` を設定すると, 大域変数 `functions` と `arrays` で与えられる関数を全て表示します.

`fundef` 関数は $\langle \text{関数名} \rangle$ に対応する関数の定義を返します. `fundef` は `dispfun` に似ていますが, `fundef` では `display` 関数を呼出さない点で異なります.

```
(%i9) neko(x):=sin(x)*exp(x);
(%o9)                      neko(x) := sin(x) exp(x)
(%i10) dispfun(neko);
(%t10)                      neko(x) := sin(x) exp(x)

(%o10)                      done
(%i11) fundef(neko);
(%o11)                      neko(x) := sin(x) exp(x)
```

この例で示す様に, `dispfun` を実行すると結果は `%t` ラベルに表示されていますが, `fundef` の方は通常の `%o` ラベルに表示されています. それ以外で違いはありません.

——— 利用者定義関数を削除する関数 ———

```
remfunction (< 関数名1>, < 関数名2>, ...)
remfunction (all)
```

`remfunction` 関数は利用者定義関数を Maxima から削除します. 利用者定義関数は大域変数 `functions` にその名前が保存されており, `remfunction` は `functions` に含まれている関数名の削除を行います. 尚, 引数として `all` が与えられた場合, 大域変数 `functions` に含まれる全ての利用者定義の関数が削除されます.

7.2.2 マクロの定義

Maxima ではマクロの定義も出来ます. マクロの定義では演算子 `::=` を用います. Maxima のマクロは最終的に LISP の S 式に展開し, それから評価が行われます.

複雑なマクロを生成する為に, block 函数に似た buildq 函数を用います.

———— buildq 函数 ————

buildq(〈変数リスト〉, 〈式〉)

buildq の引数は, 実際の式の代入が実行される迄, Maxima の解釈で勝手に変換される事を防ぐ必要があります. その為に単引用符 `'` を用います.

buildq の応用に漸化式の計算があります. ここで, 幾つかの数列を定義してみましょう.

Fibonacci 数

Fibonacci 数は, 次の漸化式を満たす数です.

———— Fibonacci 数 ————

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+1} &= F_n + F_{n-1} \end{aligned}$$

では, Maxima のマクロを利用してこの数列を定義してみましょう.

```
(%i10) fb:F[n-1]+F[n-2];
(%o10)
          F      + F
          n - 1   n - 2

(%i11) define(F[n],buildq([u:fb],u));
(%o11)
          F      := F      + F
          n      n - 1    n - 2

(%i12) F[0]:0;F[1]:1;F[2]:1;
(%o12)
          0
(%o13)
          1
(%o14)
          1
(%i15) F[10];
(%o15)
          55
(%i16) F[140];
(%o16)
          81055900096023504197206408605
```

漸化式が単純な場合は buildq を使わずに, `define(F[n],F[n-1]+F[n-2])` としても問題はありません.

ルジャンドルの多項式

ルジャンドルの多項式

$$\begin{aligned} P_0(z) &= 1 \\ P_1(z) &= z \\ P_{n+1}(z) &= (2n+1)zP_n - (n+1)P_{n-1} \end{aligned}$$

```
(%i10) pnz:((2*n-1)*z*Pz[n-1]-(n-1)*Pz[n-2])/n;
          (2 n - 1) Pz      z - (n - 1) Pz
              n - 1          n - 2
(%o10) -----
              n
(%i11) define(Pz[n],buildq([v:pnz],expand(v)));
          (2 n - 1) Pz      z - (n - 1) Pz
              n - 1          n - 2
(%o11)  Pz := expand(-----)
              n              n
(%i12) Pz[0]:1;Pz[1]:z;
(%o12)
          1
(%o13)  z
(%i14) Pz[10];
          10      8      6      4      2
    46189 z  109395 z  45045 z  15015 z  3465 z  63
(%o14) ----- - ----- + ----- - ----- + ----- - ---
          256      256      128      128      256      256
```

この例では,expand を用いて式を展開しています. この式の展開を省くと, 結果が複雑になるので, 注意が必要になります. 次の例では同じ計算で,expand を省略したものの計算結果を示しています.

```
(%i17) kill(Pz);
(%o17) done
(%i18) Pz[n]:=((2*n-1)*z*Pz[n-1]-(n-1)*Pz[n-2])/n;
```

```

(2 n - 1) z Pz      - (n - 1) Pz
      n - 1          n - 2
(%o18)      Pz := -----
      n              n
(%i19) Pz[0]:1;Pz[1]:z;
(%o19)      1
(%o20)      z
(%i21) Pz[4];
      2
      5 z (3 z - 1)
7 z (----- - 2 z)      2
      2              3 (3 z - 1)
----- - -----
      3              2
(%o21) -----
      4

```

この例で示す様に、式の簡易化が行われていない為に、 $n = 4$ でも必要以上に複雑な結果となっています。

このマクロの展開に関連する関数に、macroexpand 関数と macroexpand1 関数、大域変数には macroexpansion があります。

—— macroexpand 関数と macroexpand1 関数 ——

```

macroexpand(< 式 >)
macroexpand1(< 式 >)

```

macroexpand 関数は、与式にマクロが含まれている場合、与式の評価を伴わずにマクロの展開のみを行います。

macroexpand1 関数はマクロの展開を行いますが、この時、式の評価は伴いません。

—— 大域変数 macroexpansion ——

変数名	初期値	概要
macroexpansion	false	マクロ展開の制御で利用

大域変数 macroexpansion はマクロ展開を制御する大域変数です。Maxima のマクロは最初に呼出された時点でマクロ展開が実行されます。ここで、展開されたマクロを保持

していれば、次に同じマクロの呼出しがあった場合、展開する手間が不要となるので、その分、効率的な処理が行えます。一方で、展開したマクロを保持する為には、それなりのメモリが必要となります。

- false
マクロが呼出される度にマクロの展開を行います
- expand
マクロ呼出で評価されると、以後、マクロの展開をしなくても良い様に展開内容を記憶します。マクロの呼出では通常 grind と display も呼出します。その分、全ての展開を記憶する為に特別にメモリーを必要とします。
- displace
最初の呼出でマクロ展開が実行されます。expand の場合と比較して、幾らか少ない保存領域を必要とする割に処理速度は同程度ですが、マクロ呼出が記憶されない欠点を持っています。展開は display か grind が呼出されていれば参照出来ます。

7.2.3 apply 関数

Maxima で重要な関数の一つに apply 関数があります。この apply 関数は Maple や Mathematica にもある関数です。

— apply 関数 —

```
apply(〈関数〉, 〈リスト〉)
```

apply 関数は〈関数〉を〈リスト〉に適用した結果を与えます。

例えば、`apply(min,[1,5,-10.2,4,3])` は -10.2 となります。

関数の呼出しで、それらの引数が評価されておらず、それらの評価を希望する場合も apply は便利です。例えば、filespec がリスト [test,case] であれば、`apply(closefile,filespec)`

は `closefile(test,case)` と同値です。

一般的に、apply で評価させる場合、単引用符' を関数の先頭に置いて、関数を名詞型として、apply に評価させます。幾つかの原子の変数が、とある関数と同じ名前を持っていれば、関数としてではなく、その変数値が利用される事になります。何故なら、apply それ自身が第二の引数と同様に第一の引数をも評価するからです。

```
(%i30) neko(x):=2*x;
(%o30)                      neko(x) := 2 x
(%i31) neko:-128;
(%o31)                      - 128
(%i32) neko(10);
(%o32)                      20
(%i33) apply('neko,[20]);
(%o33)                      40
(%i34) appply(neko,[20]);
(%o34)                      appply(- 128, [20])
```

この例では,neko を関数として定義していますが,同時に,変数としての neko には-128 が束縛されています. その為,apply では単引用符' がなければ変数として評価されてしまいます.

7.2.4 関数定義に関連する関数

funmake 関数

funmake(〈関数〉,[〈引数₁〉,⋯,〈引数_n〉])

funmake 関数は〈関数〉で指定した関数を呼出して評価を行う事はしません. 単純に,〈関数〉(〈引数₁〉,⋯,〈引数_n〉)を返すだけです.

```
(%i2) funmake(f,[x,y,z]);
(%o2)                      f(x, y, z)
(%i3) funmake(neko,[x,y,z]);
(%o3)                      neko(x, y, z)
(%i4) funmake(expand,[128,"うちのタマ知りませんか?"]);
(%o4)                      expand(128, うちのタマ知りませんか?)
(%i5) funmake(a,[1,2,3]);
(%o5)                      a(1, 2, 3)
(%i6) a:10;
(%o6)                      10
(%i7) funmake(a,[1,2,3]);
Bad first argument to 'funmake': 10
```

```
-- an error. Quitting. To debug this try debugmode(true);  
(%i8) funmake('a,[1,2,3]);  
(%o8)                               a(1, 2, 3)
```

函数に指定した原子に値が束縛されている場合、その原子は内部で評価される為、エラーになります。この場合は、単引用符'を先頭に付けて名詞型とすれば問題ありません。

—— 局所変数を定義する函数 ——

`local(〈局所変数1〉, ..., 〈局所変数n〉)`

local 函数は、この函数が利用される文中で、〈局所変数₁〉, ..., 〈局所変数_n〉を全ての属性に対して局所的なものにします。local は block 文、函数定義の本体、lambda 式、又は ev 函数でのみ一度だけ使えます。この local 函数は文脈からも独立しています。

7.3 最適化

7.3.1 式の最適化

—— 式の最適化を行う関数 ——

`optimize(〈式〉)`

`optimize` 関数は〈式〉をより効率的に計算出来る Maxima の式を返却します。`optimize` は式の展開を行うものではありませんが、式に含まれる共通部分式を内部変数で置換えて、効率的に計算出来る式に変換します。この際に、`block` 文が用いられます。但し、共通部分式が無い場合は、〈式〉をそのまま返却します。

```
(%i40) optimize((x+1)^3+1/(x+1)^2+exp((x+1)^2));
```

```
(%o40)      block([%1, %2], %1 : x + 1, %2 : %1 , %e2 + %13 + --)
                                                    %2
```

```
(%i41) ans1:solve( x^4+x^3+3*x-1=0,x);
```

```
(%o41) [x = -  $\frac{\sqrt{5}}{4}$  -  $\frac{\sqrt{25 - \sqrt{5}}}{4}$  -  $\frac{1}{4}$  ,
```

```


$$x = - \frac{\sqrt{5}}{4} + \frac{2 \sqrt{2} \sqrt{5}}{\sqrt{25 - \sqrt{5}} - \frac{1}{4}} - \frac{1}{4},$$

```

```


$$x = - \frac{2 \sqrt{2} \sqrt{5}}{\sqrt{\sqrt{5} + 25} \sqrt{i} - \frac{1}{4}} + \frac{\sqrt{5}}{4} - \frac{1}{4},$$

```

```


$$x = \frac{2 \sqrt{2} \sqrt{5}}{\sqrt{\sqrt{5} + 25} \sqrt{i} - \frac{1}{4}} + \frac{\sqrt{5}}{4} - \frac{1}{4}]$$

```

```
(%i42) optimize(ans1);
```

1

1

```
(%o42) block([%1, %2, %3, %4, %5, %6, %7], %1 : -----, %2 : ----,
              sqrt(2)          1/4
                              5
              %3          %3
%3 : sqrt(5), %4 : sqrt(25 - %3), %5 : - --, %6 : --, %7 : sqrt(%3 + 25),
              4          4
              %1 %2 %4    1          %1 %2 %4    1          %1 %2 %7 %i          1
[x = %5 - ----- - -, x = %5 + ----- - -, x = - ----- + %6 - -,
              2          4          2          4          2          4
              %1 %2 %7 %i          1
x = ----- + %6 - -])
              2          4
```

optimize 関数に影響する大域変数

変数名	初期値	概要
otimprefix	%	optimize 関数で生成される記号で利用

大域変数 otimprefix に設定される文字は optimize 関数で生成された記号に使用される前置詞です。

7.3.2 LISP の関数に変換する関数

Maxima は LISP で記述されています。Maxima のプログラムも含めた全てのデータは内部表現として LISP の S 式になっており、LISP がこの内部表現を解釈して処理を実行しています。その為、関数やデータを LISP の関数やデータに変換してしまえば、内部表現の解釈の手間が省ける為、処理速度向上が見込める事になります。

Maxima には利用者定義関数を LISP の関数に変換する関数を幾つか持っています。

LISP 関数に変換する関数

```
translate(〈関数1〉, ..., 〈関数n〉)
translate(functions)
translate(all)
translate_file(〈ファイル〉)
translate_file(〈ファイル〉, 〈LISP ファイル〉)
tr_warnings.get()
```

translate 関数は Maxima の処理言語で記述した利用者定義の関数を LISP の関数に変換する関数です。Maxima 言語で記述された関数は、裏の LISP で解釈されて実行されます。これを LISP の関数にすれば、解釈する手間は省ける分、処理の高速化が望めます。

引数は $\langle \text{関数}_1 \rangle, \dots, \langle \text{関数}_n \rangle$ の様に利用者定義関数を直接指定する方法に加えて、引数に all や functions を指定して、利用者定義関数を一度に変換する事も出来ます。

変換される関数が内部で局所変数を利用する場合、mode_declare 関数を用いて、その局所変数の型を宣言する必要があります。この場合、block 文で局所変数を宣言した直後に mode_declare 関数による局所変数の型の宣言を入れます。この宣言の方法を以下に示しておきましょう。

mode_declare の使い方

```
f( x1,x2,... ) := block([ 局所変数1, 局所変数2, ... ],
                        mode_declare( 局所変数1, 型1, 局所変数2, 型2, ... ),
                         $\langle$  関数本体  $\rangle$ )
```

この mode_declare 関数は §7.3.5 で再度詳細を述べます

次に、関数を translate 関数で変換すると、大域変数 savedef が false の場合、変換された関数の名前は大域変数 functions に割当てられた関数名リストから削除されて、今度は大域変数 props に割当てられたリストに関数名が追加されます。

当然の事ですが、関数は虫取りが完遂されるまで変換すべきではありません。更に、translate 関数は変換する関数内部の式が簡易化されていると仮定しています。そうでなければ最適化されていない LISP 関数が生成されてしまうので、変換する意味が半減するかもしれません。

その為、大域変数の simp を false に設定して変換式の簡易化を禁じる事をしてはいけません。

注意すべき事に、translate 関数を用いて LISP の関数に変換しても、Maxima と LISP の整合性の問題から以前と同じ動作をする保証はありません。

translate_file 関数は Maxima 言語で記述したプログラムを含むファイルを LISP 関数のファイルに変換する関数です。translate_file は Maxima のファイル名、LISP のファイル名と translate_file が評価した引数の情報を含むファイル名を成分とするリストを返します。

最初の引数は Maxima ファイルの名前で、オプションの第二の引数は生成すべき LISP ファイル名です。第二の引数は第一引数に、trisp のデフォルト値の tr_output_file の値を第二ファイル名のデフォルト値として与えます。例えば、`translate_file("test.mc")`

でファイル test.mc を LISP ファイルの test.lisp に変換します。

更に、生成されるものには translate 関数が出力した様々な重要性の度合を持った警告メッセージのファイルがあります。第二ファイル名は常に UNLISP です。このファイルは変数を含み、それには変換されたコードでのバグ追跡の為の情報が含まれています。tr_warnings_get 関数は引数を取らない関数で、translate 関数による変換にて、translate 関数が出力する警告のリスト (内部変数*tr-runtime-warnined* に束縛されたもの) を表示します。

この関数変換に関連する大域変数は他の関数と比較しても多く、その上、名前も長いものが多い為、名前と綴を憶えるのは大変ですが、`apropos(tr_)` を実行すれば、tr_ で開始する Maxima の大域変数等のリストが出力されるので、このリストを出して名前を確認すると良いでしょう。

7.3.3 コンパイルを行う関数

—— コンパイルを行う関数 ——

```
compile ( < 関数1 >, ..., < 関数n > )
compile (functions)
compile (all)
compile_file( < ファイル >, < コンパイルされたファイル >, < LISP のファイル名 > )
compile_file( < ファイル >, < コンパイルされたファイル > )
compile_file( < ファイル > )
compfile( < ファイル >, < 関数1 >, ..., < 関数n > )
```

compile 関数は指定した Maxima の処理言語で記述した関数を LISP の関数に変換し、それを LISP の関数 compile を用いてコンパイルします。猶、compile 関数は関数名リストを返します。猶、引数に functions や all を指定すると利用者定義関数を全てコンパイルします。

これに対し、compile_file 関数は指定したファイルのコンパイルを行います。先ず、指定された < ファイル > には Maxima のプログラムが含まれており、compile_file 関数は、これを LISP 関数に translate 関数で変換し、その結果を compile 関数でコンパイルします。変換とコンパイルに成功すると、今度は結果を Maxima に読み込みます。

compile_file は四個のファイル名のリストを返します。このリストに含まれるファイル名は、元の Maxima プログラムファイル、LISP への変換ファイル、変換に関する註釈

ファイルと compile でコンパイルされたプログラムのファイルです. 猶, コンパイルに失敗すると, 返却されるリストの第四成分は false になります.

compile 関数は $\langle \text{関数}_1 \rangle, \dots, \langle \text{関数}_n \rangle$ を LISP の関数に変換し, $\langle \text{ファイル} \rangle$ に書込みます.

```
(%i28) neko(x):=sin(x);
(%o28)                                neko(x) := sin(x)
(%i29) compile("mike",neko);
```

Translating neko

```
(%o29)                                /home/yokota/mike
(PROGN (DEFPROP NEKO T TRANSLATED) (ADD2LNC 'NEKO PROPS)
(DEFMTRFUN (NEKO ANY MDEFINE NIL NIL) (X) (DECLARE (SPECIAL X))
(SIMPLIFY (LIST '(%SIN) X))))
```

7.3.4 関数の変換やコンパイルに関連する大域変数

translate 関数と compile 関数によるシステムに関連する大域変数を纏めておきましょう.

translate 関数と compile 関数のシステムに関連する大域変数

変数名	初期値	概要
compgrind	false	compile 関数による出力制御
savedef	true	translate 関数による変換後に元の関数を残す
translate	false	translate 関数による自動変換を制御
transrun	true	変換前の関数の実行
undeclaredwarn	compile	未定義変数に対する警告を制御

大域変数 compgrind が true であれば, compile による関数定義の出力が整形表示されます.

大域変数 savedef が true であれば, 利用者関数を translate 関数で変換しても, 元の Maxima のプログラムを残します. その為, 大域変数 functions に割当てられた利用者関数名リストから, 変換した関数名を削除せずに残します. 又, dispfun 関数で関数の定義が表示可能で, 関数の編集も出来ます.

false の場合は, functions に割当てた利用者関数名リストから該当関数名を削除します.

大域変数 `translate` が `true` であれば, 利用者定義関数が自動的に LISP 関数に変換されます. 猶, Maxima と LISP の整合性の問題から, 変換される前と同じ動作をするとは限らない事に注意して下さい.

変数が `mode_declare` された CRE 表現の場合, `rat` 関数を一つ以上の引数で用いたり, `rat-
vars` 関数を使ってはなりません. 又, `prederror:false` は変換しません.

大域変数 `transrun` が `false` であれば, `translate` 関数で変換されたものではなく, 元の Maxima の関数 (それらが存在していれば) が実行されます

大域変数 `undeclaredwarn` には四種類の設定項目があります.

undeclearewarn の設定項目

設定	動作
<code>false</code>	警告を表示しません
<code>compile</code>	<code>compile</code> であれば警告します
<code>translate</code>	<code>translate</code> や <code>translate:true</code> であれば警告します
<code>all</code>	<code>compile</code> や <code>translate</code> であれば警告します

`mode_declare(〈変数〉, any)` を実行して 〈変数〉 が一般の Maxima の変数である事を宣言します. 即ち, `float`, 又は `fixnum` である事に限定されません. `compile` 関数でコンパイルされる利用者定義関数中の変数を宣言する特別な動作は全て無効にしなければなりません.

次に, `translate` 関数に影響を与える大域変数は非常に多くあります. 最初に, 大域変数 `tr_state_vars` に割当てられたリストに含まれる大域変数を以下に纏めておきます.

tr_state_vars に纏められた変数

変数名	初期値	概要
define_variable	[]	
transcompile	false	compile 函数に必要な宣言を自動生成
translate_fast_arrays	true	配列変換を制御
tr_array_as_ref	true	変換函数の配列評価を指定
tr_function_call_default	general	函数変換を制御
tr_numer	false	数値変数の型を制御
tr_semicompile	false	機械コードへの翻訳を制御
tr_warn_bad_function_calls	true	不適切な宣言による函数呼出の警告制御
tr_warn_fexpr	compile	fexpr 型に対する警告制御
tr_warn_meval	compile	meval 型函数に対する警告制御
tr_warn_mode	all	変数型に対する警告を制御
tr_warn_undeclared	compile	未宣言変数に対する警告を制御
tr_warn_undefined_variable	all	未定義変数に対する警告を制御

大域変数 transcompile が true であれば, translate 函数は可能な compile 函数に必要な宣言を生成します. compile 函数は transcompile:true を用います.

大域変数 translate_fast_arrays が true の場合, 配列の変換を行います. 大域変数 tr_array_as_ref は, translate_fast_arrays が false の場合のみに translate_file 函数で変換されたプログラムの配列参照に影響を与えます. 大域変数 tr_array_as_ref が true であれば, 変換された函数は配列を評価しますが, false であれば, 変換されたプログラム中の単なるシンボルとして配列が現れます.

大域変数 tr_function_call_default は apply, expr, general と false を取り, 初期値は general となります.

- false の場合: Maxima 内部函数 meval を呼出して式を評価する事を意味します.
- expr の場合: 引数固定の LISP 函数と仮定します.
- apply の場合: apply 函数を用いて函数を引数に作用させて変換します.
- general の場合: 内部表現が mexprs と mlexprs に対しては良いコードを与えますが, macros に対しては駄目です. general の場合, コンパイルされた函数中で変数の割当てが正確である事を保証します. 例えば, 函数 f(x) を変換する際に, ここ

で f が値を割当てられた変数であったとすると、警告を出して、`apply(f,[x])` の事として関数を変換します。

猶、デフォルト設定で何等の警告メッセージが無ければ、`translate` 関数で変換し、`compile` 関数でコンパイルした利用者定義関数には元の `maxima` 関数と完全な互換性がある事を意味します。

`tr_numer` が `true` であれば、数の属性をそのまま LISP の変数にも継承させます。

`tr_optimize_max_loop` は考えられる形式で `translate` 関数でのマクロ展開と最適化工程ループの最大回数を定めます。これマクロ展開エラーを捉える為で、非中断の最適化属性です。

`tr_semicompile` が `true` であれば、`translate_file` 関数と `compile` 関数の出力形式は拡張されたマクロになりますが、LISP コンパイラで機械コードに翻訳されたものにはなりません。

`tr_warn_fexpr` は、内部形式が `fexpr` 型のものが与えられると警告します。`fexpr` 型は通常変換されたプログラム内の出力であってはず、全ての文法的に正しい特殊なプログラム書式に変換されます。

`tr_warn_meval` は関数 `meval` が呼び出されると警告します。`meval` が呼出されると、変換の問題点を指定します。

`tr_warn_mode` は、変数が指定した型に対して適切でない値が指定されていれば警告します。

`tr_warn_undeclared` が未宣言の変数に関する警告を `tty` に送るべき時を決めます。

`tr_warn_undefined_variable` は未宣言の大域変数が存在する場合に警告します。

以下に、大域変数 `tr_state_vars` にも含まれない `translate` 関数に関連する大域変数を纏めておきます。

LISP 関数への変換に関連する大域変数

変数名	初期値	概要
<code>tr_bound_function_apply</code>	<code>true</code>	変換関数の割当てに対し警告
<code>tr_file_tty_messagesp</code>	<code>false</code>	メッセージ出力制御
<code>tr_float_can_branch_complex</code>	<code>true</code>	逆三角関数の複素数値の制御
<code>tr_optimize_max_loop</code>	100	マクロループの回数
<code>tr_warn_bad_function_calls</code>	<code>true</code>	変換時の警告を制御

`tr_bound_function_apply` が `true` の場合、関数の引数として割当てられた変数が、関数として用いられている場合に警告します。例えば、`g(f,x):=f(x+1)` の様な場合です。

`tr_file_tty_messagesp` は, `translate_file` がファイルの変換を行う間に生成されたメッセージを `tty` に送るかどうかが決めます. `false` であれば, ファイルの `translate` による変換に関するメッセージは UNLISP ファイルのみに挿入されます. `true` であれば, メッセージは `tty` に送られ, UNLISP ファイルにも挿入されます.

`tr_float_can_branch_complex` は逆三角関数が複素数値を返しても良いかどうかを宣言します. 逆三角関数は `sqrt`, `log`, `acos` 等です. `true` の場合, `x` が `float` (浮動小数点型) であったとしても, `acos(x)` は `any` 型となります. `false` にしている時は, `x` が `float` 型で, その時に限って, `acos(x)` は `float` 型となります.

`tr_warn_bad_function_calls` は, 変換時に不適切な宣言が行われた為に, 関数の呼出しが生じた場合に警告します.

7.3.5 変数の型宣言

Maxima の記号は値を割当てれば大域変数となりますが, 一部の alphabetic 属性を持つ記号を除けば, 属性を一切持ちません. 又, alphabetic 属性を持つ記号も, その他の属性は初期状態では持っていません.

これは関数の LISP への変換や最適化では不利になります. その為, 記号に型を指定します. この型を指定する関数としては `mode_declare` 関数と `modeddeclare` 関数がありますが, `modeddeclare` 関数は `mode_declare` 関数と同じ関数です. この変数の宣言に加えて, 初期値を設定する関数に `define_variable` 関数があります.

— 変数に型指定を行う関数 —

```
mode_declare(<変数1>, <型1>, ..., <変数n>, <型n>)
mode_identity(<引数1>, <引数2>)
define_variable(<変数名>, <初期値>, <型>)
```

`mode_declare` 関数は `<変数i>` に `<関数i>` や `<型i>` を設定します. この関数は `translate` 関数で変換する利用者定義の関数で用いる変数の宣言で利用されます. ここで `mode_declare` 関数で指定した型は変数の `mode` 属性に割当てられます.

大域変数の mode 属性

型	mode_declare 関数で利用可能な表記	概要
float	float,real,floatp,flonum,floatnum	浮動小数点
fixnum	fixp,fixnum,integer	整数
rational	rational,rat	有理数
number	number,bignum,big	数 (多倍長整数や多倍長浮動小数点数)
complex	complex	複素数
boolean	boolean,boole	Boolean
list	list,listp	リスト
any	any,none,any_check	任意の型

以下に簡単な例を示します.

```
(%i28) mode_declare(x1,integer);
(%o28)                                     [x1]
(%i29) :lisp (get '$x1 'mode)
$FIXNUM
(%i29) mode_declare(x2,rat);
(%o29)                                     [x2]
(%i30) :lisp (get '$x2 'mode)
$RATIONAL
(%i30) mode_declare(x2,rational);
(%o30)                                     [x2]
(%i31) :lisp (get '$x2 'mode)
$RATIONAL
```

この例では変数 x1 を整数型,x2 と x3 を有理数型として宣言しています.これらの値は各変数の mode 属性に蓄えられています.この属性は LISP の get 関数を用いて取出せます.この様に,mode_declare 関数は基本的に型の指定を行う関数です. mode_declare 関数で指定した変数と型は mode_identity 関数を用いて検証が出来ます.

```
(%i9) mode_identity(integer,x1);
(%o9)                                     128
(%i10) x1:256.988;
(%o10)                                     256.988
(%i11) mode_identity(integer,x1);
```

```
Warning: x1 was declared mode fixnum, has value: 256.988
(%o11)                                256.988
(%i12) mode_identity(float,x1);
(%o12)                                256.988
(%i13) :lisp (get '$x1 'mode);
$FIXNUM
```

この様に,mode_declare で宣言した型以外の値を与える事は可能で, mode_identity も変数に割当てられた型と異なると文句を言うだけです. mode_identity は変数の mode 属性と指定された属性が同じものかどうかを検証するのではなく, 単純に変数に割当てられた型が mode_identity の引数で指定された型と同じものかどうかを検証する関数で, 正しい場合には変数の値を返し, 異なる場合には警告するだけです.

—— 型の検証に関連する大域変数 ——

大域変数	初期値	概要
mode_checkp	true	定数変数の型を検証するかどうか制御
mode_check_errorp	false	型エラー処理の制御
mode_check_warnp	true	型エラーの表示の制御

大域変数 mode_checkp が true の場合,mode_declare 関数で宣言する変数に割当てられた値の型と, 新たに宣言する型が矛盾しないか検証し, 矛盾する場合はエラーを返します.

```
(%i17) x0:1.0$
(%i18) mode_declare(x0,integer);
Warning: x0 was declared mode fixnum, has value: 1.0
(%o18)                                [x0]
(%i19) mode_checkp:false$
(%i20) mode_declare(x0,integer);
(%o20)                                [x0]
```

大域変数 mode_check_errorp が true であれば,mode_declare 関数で値が割当てられた変数に対し, 異なる型で変数の宣言を行う場合にエラーを出力します.

大域変数 mode_check_warnp が true の場合,mode_identity 関数は変数に割当てられた値の型と指定した型が異なる場合に警告を出します.

define_variable 関数は変数の宣言に加え, 初期値と型の設定が行える関数です. 但し, この関数は属性を上手く使う事で, 変数に値を割当てる際に, その変数の利用者が設定

した条件に適合するかどうかを検証する事も行える様に出来ます.
この `define_variable` 関数の処理手順を以下に示しておきます.

- 引数が2個以上ある事を確認します.
- 与えた変数が LISP の記号である事を確認します.
- `mode_declare` 関数を用いて変数型を宣言します.
- `declare` 関数を用いて変数が `special` 属性を持つ事を宣言します.
- 型が `any` でなければ, 属性 `assign` に属性値 `assign-mode-check` を設定します. この属性値が設定されていない変数に対して, Maxima は変数への値の割当の際に, 値の検証を行いません. 値の検証を行う為には大域変数の `value-check` 属性に真理関数を割当てておく必要があります. この設定では `qput` 関数が有効です.
- 変数に値が割当てられていなければ, 指定した初期値を設定します. もしも値が既に割当てられていれば, `define_variable` 関数は与えられた初期値を割当てずに, そのままの値にしておきます.

割当ての際に, 変数値の検証を行う方法は, `qput` 関数を用いて宣言した大域変数の `value-check` 属性に変数値を検証する真理関数名を割当ておきます. 以下に, 動作例を示しておきましょう.

```
(%i1) ptest(y):=if not primep(y) then error(y,"is not prime!!")$
(%i2) define_variable(tama,5,integer)$
(%i3) qput(tama,ptest,value-check)$
(%i4) tama;
(%o4)                                     5
(%i5) tama:15;
15 is not prime!!
#0: ptest(y=15)
-- an error. Quitting. To debug this try debugmode(true);
(%i6) :lisp (get '$tama 'assign)
ASSIGN-MODE-CHECK
(%i6) define_variable(mike,5,any)$
(%i7) properties(mike);
(%o7)                                     [value, special]
```

```
(%i8) qput(mike,ptest,value_check)$
(%i9) properties(mike);
(%o9) [value, [user properties, value_check], special]
(%i10) mike:15;
(%o10) 15
(%i10) :lisp (get '$mike 'assign)
NIL
(%i10) :lisp (put '$mike 'assign-mode-check 'assign)
ASSIGN-MODE-CHECK
(%i10) mike:15;
15 is not prime!!
#0: ptest(y=15)
-- an error. Quitting. To debug this try debugmode(true);
```

この例では、最初に素数でなければエラーを返す真理関数 `ptest` を定義し、それから、`define-variable` 関数で大域変数 `tama` に初期値 5 を与えて整数変数として宣言します。それから、`qput` 関数を用いて `check-value` 属性に `ptest` を与えます。この時、変数 `mike` に 15 を設定しようとするれば、15 は素数ではないのでエラーになっています。

ここまでの動作をもう少し詳しく解説しましょう。先ず、`define-variable` 関数による宣言で、変数の型を `integer` と宣言しています。`define-variable` 関数は、変数の型を `any` 以外に指定していれば、宣言する大域変数の `assign` 属性に内部関数の `assign-mode-check` 関数を割当てます。この内部関数は変数の `value-check` 属性に設定された関数を用いて変数の評価を実行する関数です。次に、この `value-check` 属性の設定で `qput` 関数を用いています。ここでの例では、大域変数 `tama` の `value-check` 属性に `ptest` 関数を割当てていますね。すると、大域変数 `tama` に値を割当てる時に、`assign-mode-check` 関数が、大域変数 `tama` の `check-value` 属性に割当てられた真理関数で割当てようとする値を評価します。この例では、`ptest` 関数に 15 が引渡されます。この例では 15 が素数でない為に `false` となり、変数に 15 が割当てられません。

次の例では、大域変数 `mike` の型を `any` とした場合です。この場合、変数 `mike` の `value-check` 属性に値を設定しても、先程の様に `mike:15` を実行した場合にエラーも何も出ません。この場合、大域変数 `mike` の `assign` 属性に `assign-mode-check` が設定されていない為です。これは LISP で `(get '$mike 'assign)` を実行すれば、NIL が返されるので判ります。

そこで、`:lisp (put '$mike 'assign-mode-check 'assign)` として見ましょう。これによって、属性 `assign` の値として内部関数 `assign-mode-check` が設定されます。す

ると、割当の際に検証が実行される様になります。

7.3.6 declare_translated 関数

declare_translated 関数

```
declare_translated(〈関数1〉, …, 〈関数n〉)
```

declare_translated 関数は、引数の関数が既に変換されている事を宣言する関数です。Maxima のプログラムファイルを LISP に変換する際に、そのファイル中のどの関数が translate 関数で変換された関数、或いは compile 関数でコンパイルされた関数として呼出されるべきか、そして、どれが Maxima の関数で、又、未定義のものであるかを知る事は translate 関数にとって重要な事です。

ファイルの先頭にこの宣言を置くと、ある記号がたとえ LISP 関数の値を持っていなかったとしても、呼出された時にそれを持つ事を教えます。(mfunction-call fn arg₁ arg₂, …) が生成されるのは、〈関数_n〉が LISP 関数に変換されるべきものであるかを translate 関数が知らない時です。

7.4 データ入出力について

7.4.1 データの入出力

この節では Maxima の基本的なデータ入力と出力について述べます。Maxima は Common LISP で記述されている為、入出力では LISP の入出力函数を用いたものになります。但し、Maxima は表示されている形式と内部形式が別物の為、結果や式を保存したり、逆に保存したものを読み込む際には、注意が必要になります。

ファイルに画面と同じ出力を行いたければ、`writefile` 函数を用います。この `writefile` 函数は LISP の `dribble` 函数を用いたもので、入力と出力をそのまま指定したファイルに保存します。但し、`writefile` では指定したファイルを新規に生成するので、単純に既存のファイルに記録したければ、`appendfile` 函数を用います。`writefile` 函数と `appendfile` 函数で開いたファイルを閉じる場合、`closefile()` で開いたファイルを閉じます。

但し、これらのファイルは実質的に記録ファイルであって、Maxima でそのまま再利用は出来ません。再利用可能なファイルを生成するのは、`save` 函数と `stringout` 函数です。ここで、`save` 函数は Maxima の内部表現を保存する函数で、`load` 函数や `loadfile` 函数を用いて Maxima に読み込みます。これに対して、`stringout` 函数と `grind` 函数は内部形式ではなく、Maxima の入力に対応する通常の形式でデータの保存を行います。

Maxima で C の `scan` 命令に似たものに `read` 函数と `readonly` 函数があります。これらの函数は、引数として与えた文字列を全て同一行に表示し、キーボードからの入力を待ちます。利用者は通常の Maxima の入力と同様に式を入力します。ここで、行末にはセミコロン ; か \$ を付けます。すると、`read` 函数の場合は式を Maxima で評価し、`readonly` 函数の場合は、式を評価せずにそのまま受け取ります。

この様に、単純なファイル操作に限定されるとは言え、必要なものが最低限揃っています。とは言え、このままでは不十分で、より柔軟に利用する為には、§6.11 で解説する `stringproc` パッケージに含まれる入出力函数や LISP で補う事になります。

7.4.2 Maxima のファイル検出方法

Maxima には様々なファイルが附属します。そして、各ファイルは所定の場所、即ち、ディレクトリ (フォルダ) に収められています。

`load` 函数、`demo` 函数や `example` 函数を用いる上で、このディレクトリ構成を把握しておく事が重要なので、Maxima のディレクトリ構成を簡単に纏めておきましょう。

Maxima のディレクトリ構成

src	Maxima のソースファイル (LISP のプログラムファイル) が収納されたディレクトリ
share	Maxima の関数, パッケージファイルが収納されたディレクトリ
demo	Maxima の関数等のデモファイルが収納されたディレクトリ
tests	Maxima のテスト用プログラムが収納されたディレクトリ
doc	Maxima の文書が収納されたディレクトリ

そして, これらのディレクトリに対応する Maxima の大域変数があります.

ディレクトリに関連する大域変数

変数名	初期値	概要
file_search_maxima		Maxima のプログラムファイル読込に関連する大域変数
file_search_lisp		lisp のプログラムファイル読込に関連する大域変数
file_search_demo		デモファイル読込に関連する大域変数
file_search_usage		texi ファイル等の読込に関連する大域変数

これらの大域変数は, 全て LISP の文字列型のデータを成分とする Maxima のリストが割当てられています. ここで, LISP の文字列型データの形式は, 基本的に検索経路とファイル型の文字の並びで構成されたものです. そして, これらの大域変数を load 関数, demo 関数や example 関数等のファイルの読込を行う関数が利用する事で, 単純に関数やパッケージの名前だけを記述する事で, ファイルの読込が行える様になっています. 例えば, surfplot という関数を読込みたい場合, `load(surfplot);` と入力したとしましょう. すると, Maxima 内部では, パッケージファイルが収納されているディレクトリ情報が登録されている大域変数 file_search_maxima の成分に surfplot を当て嵌めて, 適合するファイルが存在するかを調べ, 存在する場合にはファイルの読込を行います. この大域変数の成分は具体的には次の様な書式になっています.

ファイル読込に関連する文字列の書式例

```
/usr/local/share/maxima/5.14.0/share/###.{mac,mc}
```

ここで, ### の個所に引数の surfplot が代入されて, 後の, {mac,mc} が surfplot 関数が収納されたファイルの修飾子となるべき修飾子の候補になります. この修飾子は大域変数の種類によって異なります. そして, ### の前のディレクトリが, 読込むべきファイル

が置かれているディレクトリになります。因に、これらの大域変数の値は `src/init-cl.lisp` で設定されています。

このデータ型は LISP の文字列型ですが、Maxima-5.13.0 以前では Maxima の文字列と LISP の文字列はデータ型が異なる為、この大域変数を構成するデータの扱いには注意が必要です。但し、5.14.0 から Maxima の文字列は LISP の文字列型になった為、Maxima の文字列データを LISP の文字列型に変換する等の処理は不要となっています。

さて、ここで、適合するファイルが存在した場合、Maxima はファイルの読込を実行します。もしも、存在しなければエラーを返す仕組みとなっています。

大域変数 `file_search_maxima` は Maxima のプログラムファイルの読込で用いられる大域変数で、ここで指定されるファイルの修飾子は `.mac` と `.mc` になります。

大域変数 `file_search_lisp` は LISP のプログラムファイルの読込で用いられるもので、ここで指定されるファイルの修飾子は、`.fas`、`.lisp`、`.lsp` です。

大域変数 `file_search_demo` は `demo` 関数で用いられるデモファイルの読込で用いられるものです。ここでデモファイルの修飾子は、`.dem`、`.dm1`、`.dm2`、`.dm3`、`.dmt` です。

そして大域変数 `file_search_usage` は `printfile` 関数等で用いられる USAGE ファイルの読込等で用いられる大域変数です。ここでファイルの修飾子は `.texi`、`.usg` です。

——— ファイルの検出等に関連する関数 ———

```
filename_merge(< 文字列1 >, < 文字列2 >)
file_search(< ファイル >, < 検索経路 >)
file_search(< ファイル >)
file_type(< ファイル名 >)
```

`filename_merge` 関数は二つの `< 文字列1 >` と `< 文字列2 >` の結合を行います。内部的では先頭に `#P` を文字列の先頭に付けた対象を生成しますが、Maxima 上では単純に文字列を繋ぎ合せた様にしか見えません。

基本的には Maxima の各種命令でファイルの検索を行う際にパス指定のあるファイル名を生成する際に用いられる関数です。

`file_search` 関数は指定したファイルを大域変数 `file_search_lisp`、大域変数 `file_search_maxima` と大域変数 `file_search_demo` に適合するディレクトリとファイルの修飾子で検索し、ファイルが存在すればファイル名を返し、存在しなければ `false` を返します。猶、上記の検索経路以外で検索する必要がある場合、第二引数に経路を記述する事が可能です。この場合、検索経路の書式は Maxima の検索経路の書式でなければなりません。

```
(%i10) file_search("neko.usg");
```

```
(%o10)                                neko.usg
(%i11) file_search(neko,["/home/yokota/##.{usg,tex}"]);
(%o11)                                /home/yokota/neko.usg
(%i12) file_search(neko,["/home/yokota/##.{usg,tex}"]);
(%o12)                                /home/yokota/neko.usg
(%i13) file_search(neko,["/home/yokota/##.{tex,usg}"]);
(%o13)                                /home/yokota/neko.tex
```

file_type 関数は指定したファイルの属性を返します. 但し, ファイル名の末尾で判断する関数なので, 返却する値も, object, lisp や maxima を返します. ここで, object はコンパイルされた LISP ファイル, lisp は LISP のテキストファイル, maxima は Maxima の処理言語で記述されたファイルになります.

7.4.3 ファイル処理に関連する関数

—— バッチ処理に関連する関数 ——

```
batch(< ファイル名 >)
batchload(< ファイル名 >)
```

batch 関数は指定されたファイルに含まれる Maxima の命令行を逐次評価します. ファイルはパスを含まない場合, 大域変数 file_search_maxima に含まれるディレクトリ上を検索し, 存在した場合には読み込みと実行をします.

ここでバッチファイルの内容は, ほぼ Maxima での入力行と同じものになります. 従って, 行末には; か\$を置きます. 又,%と%th を用いて入力と出力を指定する事も出来ます. 猶, 空行, Tab, 注釈や改行コードは無視されます. そして, 注釈は C の様に /* */ で括ります. この註釈行は複数行に跨っても問題はありません.

batch 処理ファイルは通常のテキストエディタで編集する事も出来ますし, Maxima の stringout 関数で出力したものも使えます.

深刻なエラーが生じた場合, ファイル末端に達した場合にのみ, 利用者に制御が戻されます. 但し, 利用者はどの時点でも Cntrl-g を押せば, この処理を止められます.

batchload 関数は指定されたファイルのバッチ処理を行います. batch との違いは, batchload ではファイルに記述された式の入力や出力表示等を行わない事です.

読込を行う函数

```
load(< ファイル名 >)
loadfile(< ファイル名 >)
aload_mac(< ファイル名 >)
auto_mexpr(< 函数 >, < ファイル名 >)
read(< 文字列1, ... >)
readonly (< 文字列1 >, ...)
```

load 函数は文字列や、リストで表現されたファイル名の読込みを行います。ディレクトリが指定されていないならば、最初にカレントディレクトリ、それから大域変数 file.search_axima, 大域変数 file.search.lisp や大域変数 file.search.demo に指定されているディレクトリとファイルの修飾子に適合するファイルを読込もうとします。

load 函数はファイルが batch 処理に対応している事を見付けると, batchload を用います (これは、黙って端末に出力やラベルを出力せずにファイルの batch 処理を実行する事を意味しています)。

他のファイルの読込を行う Maxima 命令に, loadfile, batch と demo があります。loadfile は save で書込んだファイルに対して動作し, batch と demo は strignout で書込まれたり、テキストエディタで命令のリストとして生成されたファイル向けです。

loadfile 函数は指定されたファイルを読込みます。この函数は以前の Maxima の処理で save 函数で保存した値を Maxima に戻す事に使えます。

ここで、経路の指定はオペレーティングシステムの経路指定に方法に従います。例えば, unix の場合は /home/user ディレクトリにある foo.mc ファイルを読込むのであれば, "/home/user/foo.mc" となります。

猶, save 函数で保存したファイルを loadfile 函数で読込むと, Maxima は初期化されるので注意が必要です。

aload_mac 函数は Maxima のパッケージファイル (修飾子が mc, mac) の読込で利用可能な函数です。修飾子は省略しても構いません。

aut_mexpr 函数はファイルから第一引数で指定した函数の読込を行う函数です。この函数は第一引数で指定した函数が存在する場合には読込を行わずに false を返します。

read 函数は画面上に全ての引数を表示して入力待ちます。利用者が式を入力すると、入力した式は Maxima に引渡されて評価が行なわれます。

```
(%i45) a:read("mikeneko");
mikeneko
diff(x^2+1,x);
```

```
(%o45) 
$$2x$$

```

この例では, mikeneko と表示された後に, `diff(x^2+1,x);` を入力しています. ここでの入力でも通常の入力と同様に行末に ; か \$ が必要です. この例では, 入力した値が Maxima に評価されて, 結局, 変数 `a` に $2x$ が割当てられています.

`readonly` 関数は引数を全て表示し, それから式を読込みます. 基本的には `read` と同様ですが, `read` と違うのは, 読込んだ式を評価しない事です.

```
(%i46) a:readonly("mikeneko");
```

```
mikeneko
```

```
diff(x^2+1,x);
```

```
(%o46) 
$$\text{diff}(x^2 + 1, x)$$

```

stringout 関数

```
stringout(< ファイル名 >, < 式1 >, < 式2 >, ...)
```

```
stringout(< ファイル名 >, [< m >, < n >])
```

```
stringout(< ファイル名 >, input)
```

```
stringout(< ファイル名 >, functions)
```

```
stringout(< ファイル名 >, values)
```

`stringout` 関数は指定したファイルに Maxima が読込める書式で出力します. 直接, < 式₁ >, < 式₂ >, ... と式を並べると, 各式が順番にファイルに書込まれます.

引数に, [< m >, < n >] と指定すると入力行の m 行から n 行がファイルの書込まれます.

これらに対し, `input` を指定すると入力行全てが書込まれます. `functions` を指定すると大域変数 `functions` に記載された利用者定義の関数が全て保存されます.

同様に `values` を指定すると, 大域変数 `values` に記載された利用者定義の変数の値が書込まれます.

猶, この `stringout` 関数は `writefile` を実行中に利用する事も可能です.

大域変数の `grind` が `true` であれば, `stringout` は文字列では無く, `grind` と同じ書式で出力します.

書込を行う函数

```

appendfile(< ファイル名 >)
writefile(< ファイル名 >)
with_stdout(< ストリーム >, < 式1 >, ..., < 式n >)
with_stdout(< ファイル名 >, < 式1 >, ..., < 式n >)
closefile()

```

appendfile 函数は指定したファイルに Maxima の入出力の追加を行います. writefile 函数との違いは, 同名のファイルが存在した場合, writefile 函数は上書きをしますが, appendfile 函数は既存のファイルの末尾に Maxima の入出力を追加する事が違います. 猶, writefile 函数と同様に指定したファイルは closefile() 閉じます.

writefile 函数は書込用のファイルを新規に開きます. writefile 函数を実行すると, それ以降の Maxima への入力と出力処理は全て指定したファイルに記録されます. その為, このファイルをそのまま Maxima に再度読込ませたりする事は出来ません.

ファイル名の指定は文字列で行います. 文字列ではなく, ABCD の様に二重引用符無しで指定すると, Maxima は \$ を頭に付けたファイル名, 即ち, この例では \$ABCD でファイルを生成します. 猶, この writefile の実体は LISP の dribble 函数です.

ファイルを閉じる場合は closefile() を用います.

以下に簡単な例を示します.

```

(%i1) writefile("test1");
(%o1)          #<OUTPUT BUFFERED FILE-STREAM CHARACTER test1>
(%i2) 1+2+3;
(%o2)
6
(%i3) diff(sin(x)*x+2,x);
(%o3)          sin(x) + x cos(x)
(%i4) closefile();
(%o4)          #<CLOSED OUTPUT BUFFERED FILE-STREAM CHARACTER test1>

```

上記の writefile で生成したファイル test1 の内容を以下に示します.

```

;; Dribble of #<IO TERMINAL-STREAM> started 2005-11-17 06:31:16
(%o1)          #<OUTPUT BUFFERED FILE-STREAM CHARACTER test1>
(%i2) 1+2+3;
(%o2)
6
(%i3) diff(sin(x)*x+2,x);

```

```
(%o3)                               sin(x) + x cos(x)
(%i4) closefile();
;; Dribble of #<IO TERMINAL-STREAM> finished 2005-11-17 06:31:40
```

この様に,Maxima の画面入出力そのまゝが保存されています. この writefile 関数を記録ファイルの生成に利用すれば, 下手なフロントエンドも不要になります.

with_stdout 関数は, 指定されたストリームやファイルを開き, $\langle \text{式}_1 \rangle, \dots, \langle \text{式}_n \rangle$ の評価と書込を行います. この時, 各式の評価の標準出力への任意の出力は端末の代りに指定したストリームやファイルに送られ, 端末側には常に false が返されます.

with_stdout 関数は大域変数 file_output_append によってファイルの書込の制御が行われます. この大域変数 file_output_append の初期値は, false ですが, その値が true の場合, 既存のファイルに対して内容の追加を行い, そうでない場合にはファイルの上書を行います.

closefile 関数は closefile() で appendfile や writefile で開かれたファイルを閉じます. closefile 関数は LISP の close 関数を使った関数です. この close 関数は開かれたストリームを閉じる関数です.

save 関数

```
save(⟨ファイル名⟩, ⟨引数1⟩, ⟨引数2⟩, ...)
save(⟨ファイル名⟩, ⟨名称1⟩ = ⟨式1⟩, ⟨名称2⟩ = ⟨式2⟩, ...)
save(⟨ファイル名⟩, [⟨m⟩, ⟨n⟩])
save(⟨ファイル名⟩, values, functions, labels, ...)
save(⟨ファイル名⟩, all)
```

save 関数は指定したファイルに, 指定した式や関数等の値を書込みます. 更に, 保存した値は削除されずに Maxima 本体にも残っています.

$\langle \text{引数}_1 \rangle, \langle \text{引数}_2 \rangle, \dots$ で, 各引数の値を保存します.

$[\langle m \rangle, \langle n \rangle]$ で m 番目の入力行から n 番目の入力行の内容を保存します.

引数に大域変数 values, functions, labels 等を指定する事も可能です. values, functions で利用者が設定した変数値や定義関数を全て保存します. 又, labels を指定すると, 入出力行の内容が全て保存します.

最後に, 引数に all を指定すれば, Maxima の内容をファイルに保存します. この場合は, 入力や計算結果だけではなく, Maxima の様々な設定も一緒に保存されるので, 処理した内容以上にファイルが大きなものとなるので注意が必要になります. save 関数の返却値は保存先のファイル名となります.

```
(%i1) 1+2+3;
```

```

(%o1)                                     6
(%i2) a1:x^2+y^2+1;

                                     2    2
(%o2)      y  + x  + 1
(%i3) resultant(x-t,y-t^2,t);

                                     2
(%o3)      y  - x
(%i4) save("test",all);
(%o4)      test

```

save 関数で保存したファイルは loadfile 関数で Maxima に再び読み込む事が出来ます。但し,loadfile を実行すると,以前の Maxima 自体を初期化し,save 関数を実行した時点にまで戻ってしまう効果があるので,注意が必要になります。

以下の例では最初に loadfile でファイル test を読み込んでいますが,行ラベルは上の save で保存する場合と同じものになっている事と,二度目に loadfile を実行するとラベルが (%i8) から (%i5) に戻っている事に注意して下さい。

```

(%i1) loadfile("test");
(%o4)      test
(%i5) %i2;

                                     2    2
(%o5)      a1 : y  + x  + 1
(%i6) %i1;
(%o6)      6
(%i7) %o3;

                                     2
(%o7)      y  - x
(%i8) loadfile("test");
(%o4)      test
(%i5)

```

猶,save ファイルの内容は,LISP の S 式そのものとなります。要するに,LISP 上で動く Maxima の為のデータファイルになります。ファイルの先頭側に実行内容の内部形式が記述されますが,その後には Maxima の諸設定が保存されます。その為,以下に示す例は $1+2+3$ から 4 行の入力だけでしたが,save 関数で保存したファイル (test) は 256 行に及びます。これは内部形式で記述するとどうしても長くなる側面もありますが,実際は,入出力以外に様々な設定 (大域変数の値等) も保存されている為です。

```

;;; -*- Mode: LISP; package:Maxima; syntax:common-lisp; -*-
(in-package "MAXIMA")
(DSKSETQ %I1 '((MPLUS) 1 2 3))
(ADDLABEL '%I1)
(DSKSETQ %O1 6)
(ADDLABEL '%O1)
(DSKSETQ %I2 '((MSETQ) $A1 ((MPLUS) ((MEXPT) $X 2) ((MEXPT) $Y 2) 1)))
(ADDLABEL '%I2)
(DSKSETQ %O2 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 2)))
(ADDLABEL '%O2)
(DSKSETQ %I3
  '($ (RESULTANT) ((MPLUS) $X ((MMINUS) $T))
    ((MPLUS) $Y ((MMINUS) ((MEXPT) $T 2))) $T))
(ADDLABEL '%I3)
(DSKSETQ %O3
  '((MPLUS SIMP) ((MTIMES SIMP) -1 ((MEXPT SIMP RATSIMP) $X 2)) $Y))
(ADDLABEL '%O3)
(DSKSETQ %I4 '($ (SAVE) &TEST $ALL))
(ADDLABEL '%I4)
(DSKSETQ $A1 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 2)))
(ADD2LNC '$A1 $VALUES)
以下略

```

その為,Maxima 内部でデータがどの様に処理されているかを見る場合には重宝するでしょう. とは言え, 作業を一旦中断し, 中断した個所から再度処理を行う必要がなければ,save 以外の命令, 例えば,stringout や grind を用いた方が総合的な使い勝手自体は良いでしょう.

——— その他の読込に関連する大域変数 ———

loadprint	false	読込に伴うメッセージ表示を制御
packagefile	false	パッケージ作成時の情報を制御

大域変数 loadprint は loadfile 関数や autoloader 関数によるファイル読込に伴うメッセージ表示を制御する大域変数です. 大域変数 loadprint が取る値は, true,loadfile,autoloader,false の四種類で, 各々, 対応が異なります.

- true であれば, メッセージが常に表示されます.
- loadfile の場合は, loadfile 関数が用いられた時のみに表示されます.
- autoload 関数の場合は, ファイルが自動的に読み込まれた時のみに表示されます,
- false の場合, メッセージが決して表示されません.

大域変数 `packagefile` は `save` 関数や `translate` 関数を用いてパッケージ (ファイル) を作成する時に, `packagefile:true` と設定してあれば, ファイルが読み込まれる時点で必要な場所を除いた情報が Maxima の大域変数, 例えば `values`, `functions` に追加される事が避けられます.

この方法でパッケージに含まれる物は, 利用者のデータを付け加えた時点で, 利用者の側では得られません. これは名前の衝突の問題を解決するものではない事に注意して下さい. この大域変数は単にパッケージファイルへの出力に影響を与える事に注意して下さい. 猶, この変数の値を true に設定すると, Maxima の初期化ファイルの生成でも便利です.

—— ファイル編集を行う関数 ——

```
ed(< 文字列 >)
```

`ed` 関数は引数で指定された名称のファイルを編集します. これは LISP の `ed` 関数を単純に用いるものです. 通常, Common LISP 側から `ed` 関数を用いてファイルを編集する場合, スペシャル変数 `*editor*` で指定されたエディタが利用されますが, Maxima から利用する場合は, このスペシャル変数が参照されず, システム側の環境変数 (UNIX の場合は, 環境変数 `EDITOR`) が利用されます. 環境変数に値が未設定の場合, UNIX 環境では `vi`, MS-Windows 環境ではメモ帳がファイル編集のエディタとして利用されます.

—— ファイルの表示を行う関数 ——

```
printfile(< ファイル >)
```

`printfile` 関数は指定されたファイルをそのままフロントエンドに表示する関数です. この関数は単純に `read-char` 関数と `princ` 関数を用いてファイルの EOF (End Of File) が現れる迄, 読み込みと表示を行う関数です. このファイルの検索では `file_search1` 関数を用い, その時に検索するディレクトリ/フォルダとファイルの種類は大域変数 `file_search_usage` で指定します.

この大域変数名からも判る様に `printfile` 関数は Maxima の USAGE ファイル (修飾子が `.usg` の ASCII 形式のファイル) を表示させる為のものです. この USAGE ファイルを表示させる場合は, 二重引用符で括らずに, ファイルの修飾子 (`.usg`) も付けないシンボルで与える事が可能です. 例えば, `printfile(share)` を実行すると, `share.usg` ファイルが表示されます. 但し, 一般のファイルに対しては, その経路も含めて, 二重引用符で括って引き渡さなければなりません.

猶, 日本語の表示も可能なので, 次の様な事が可能です.

—— ファイル `neko.usg` ——

ここで何故, 結び目なのかという理由ですが, 結び目理論はドイツ語では `Knotten Theorie` と呼びます. それにしても, `Knotten!!` 実に, `KNOPPIX` に似て
 い
 ますねえ… . そこで, 結び目愛好家の為に, Maxima で結び目の不変量を計算して,
`KNOPPIX/Math` を `Knotten Pics/Math` と洒落込もうというのが目的です.

このファイル `neko.usg` を `file_search_usage` に登録されたディレクトリに置きます. そして, `prinfile` を実行すると以下の結果を得る事になります.

```
(%i2) printfile(neko);
```

ここで何故, 結び目なのかという理由ですが, 結び目理論はドイツ語では `Knotten Theorie` と呼びます. それにしても, `Knotten!!` 実に, `KNOPPIX` に似て
 いますねえ… . そこで, 結び目愛好家の為に, Maxima で結び目の不変量を計算して,
`KNOPPIX/Math` を `Knotten Pics/Math` と洒落込もうというのが目的です.

```
(%o2) /usr/local/share/maxima/5.14.0/share/neko.usg
```

第8章 Maximaの函数

Sechste Stimme

Faust, ich bin geschwind als wie des Menschen Gedanke.

Faust

Als wie des Menschen Gedanke?

Was will ich mehr?

Konnt' ich so viel erhoffen?

Was will ich mehr denn!

Als daß Erfüllung schreite mit dem Wunsche,

als daß die Tat zugleich ins Leben trete mit der Absicht:

Konnt' ich so viel erhoffen!

Was will ich mehr?

Dein Name?

Sechste Stimme

Mephistopheles.

第六の声

ファウストよ、俺の速さは人間の思考と同じだ。

ファウスト

人間の思考と同じだと？

何をそれ以上？

それ程の事が望めるのか？

何をそれ以上望もうか!

望めば忽ち叶えられ、

意図すれば直ちに行為となる:

それ程の事が期待出来るのか!

何をそれ以上？

お前の名は？

第六の声

メフィストフェレス。

Buzoni, Doktor Faust[58] より

この章では,Maxima の様々な函数について解説します.

但し,Maxima が持つ全ての函数を解説するものではない為, 必要に応じて, 配布のMaxima に附属のマニュアルを参照して下さい.

8.1 数論に関連する関数

8.1.1 階乗

—— 階乗に関連する演算子 ——

! $n!$ n が整数の場合, n の階乗を計算. 一般の場合 $\Gamma(x+1)$
 !! $n!!$ n が奇数 (偶数) ならば, n 以下の奇数 (偶数) の積

演算子!と演算子!!は Maxima の演算子としての属性を持っています. これらの!と!!の演算子としての詳細は §5.3 の後置式演算子の項目を参照して下さい.

先ず, $n!$ は n が整数の場合に n の階乗を計算します. 一般の数値に対しては $\Gamma(n+1)$ の計算を行います.

演算子!は階乗演算子!に似ていますが, 以下の式で定義された演算子です:

$$n!! =_{def} \prod_{i=0}^{\text{entier}(\frac{n}{2})} (n-i)$$

特に, n が奇数の場合, $n!!$ で n 以下の全ての奇数の積を計算し, n が偶数の場合, $n!!$ で n 以下の全ての偶数の積を計算する事になります. 従って, n が整数の場合, $n! = n!!(n-1)!!$ が成立します.

```
(%i6) 10!;
(%o6)                                     3628800
(%i7) 10!!;
(%o7)                                     3840
(%i8) 9!!;
(%o8)                                     945
(%i9) 10!!*9!!;
(%o9)                                     3628800
```

—— genfact 関数 ——

genfact($\langle$ 式₁ $\rangle$, $\langle$ 式₂ $\rangle$, $\langle$ 式₃ $\rangle$)

genfact 関数は一般化された階乗を計算する関数です. この関数は次の式で定義されたものです.

$$\langle \text{式}_1 \rangle \cdot (\langle \text{式}_1 \rangle - \langle \text{式}_3 \rangle) \cdot (\langle \text{式}_1 \rangle - 2\langle \text{式}_3 \rangle) \cdots (\langle \text{式}_1 \rangle - (\langle \text{式}_2 \rangle - 1)\langle \text{式}_3 \rangle)$$

従って, 整数 n に対して $\text{genfact}(n, n, 1)$ が $n!$ に対応し, $\text{genfact}(n, n/2, 2)$ が $n!!$ に対応する事になります.

———— makefact 函数 ————

`makefact(< 式 1 >)`

`makefact` 函数は γ 函数式を階乗に変換する為の函数です. 猶, `gamma` 函数の名詞型を含まない式は, そのままの返す函数です.

```
(%i62) makefact(gamma(x));
(%o62) (x - 1)!
(%i63) makefact(gamma(x)*gamma(y));
(%o63) (x - 1)! (y - 1)!
(%i64) makefact(gamma(x)>gamma(y));
(%o64) (x - 1)! > (y - 1)!
(%i65) makefact(x^2+1);
(%o65) x^2 + 1
```

猶, この `makefact` 函数の逆操作を行う函数が `makegamma` 函数です. この函数は, `makefact` 函数と同様に階乗の名詞型を `gamma` 函数で置換える函数です.

———— multinomial_coeff 函数と minfactorial 函数 ————

`multinomial_coeff(< $a_1$ >, ..., < $a_n$ >)`
`multinomial_coeff()`
`minfactorial(< 式 >)`

`minfactorial` 函数は階乗を含む式の簡易化を行う函数です. 但し, $\frac{m!}{(m-3)!}$ の様な式に有効で, $m(m-1)!$ の簡易化では使えません.

```
(%i17) minfactorial(m!/(m-3)!);
(%o17) (m - 2) (m - 1) m
```

8.1.2 剰余

整数の剰余は `mod` 函数を用います.

mod 関数の構文

```
mod(< 整数1>, < 整数2>)
```

mod 関数は引数を二つ取ります. 例えば, 整数 a, b, c, r に対し, $a = cb + r$ であれば, $\text{mod}(a, b)$ は r を返します.

```
(%i12) mod(129,7);
(%o12)                                     3
(%i13) mod(129,3);
(%o13)                                     0
```

猶, polymod 関数の様に, 大域変数 modulus の影響を mod 関数は一切受けません.

8.1.3 Bell 数

belln 関数

```
belln(< 正整数値 >)
```

belln 関数は Bell 数を返す関数です. ここで Bell 数 B_n は n 個の元で構成される集合のグループ化の総数です. 例えば, 2 成分の集合 $\{a, b\}$ の場合, $\{\{a\}, \{b\}\}$ と $\{\{a, b\}\}$ の二種類となる為, $B_2 = 2$ となります. 猶, B_0 は 1 と定めます.

```
(%i16) makelist(belln(x), x, 1, 10);
(%o16)      [1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
```

8.1.4 Bernoulli 数

Bernoulli 数に関連する関数

```
bern(< 正整数値 >)
bernpoly(< 変数 >, < 正整数値 >)
```

bern 関数は Bernoulli 数 B_n を出力する関数です. 歴史的に Bernoulli 数 B_n は次の級数の和で出て来る数です:

$$\sum_{i=1}^n i^k = \sum_{j=0}^k \binom{k}{j} B_j \frac{n^{k+1-j}}{k+1-j}$$

そして, Bernoulli 数 B_n は次の漸化式で定義される数です:

$$\sum_{j=0}^k \binom{k+1}{j} B_j = k+1$$

猶, この Bernoulli 数を日本の関孝和も Bernoulli とは独立して発見しています¹.

但し, 上述の定義では $B_1 = \frac{1}{2}$ となりますが, 現在は $B_1 = -\frac{1}{2}$ とする流儀が用いられ
ており, Maxima の計算もこの流儀に従ったものです.

この場合, Bernoulli 数は次の漸化式で定義される事になります:

- $B_0 = 1$
- $\sum_{i=0}^n \binom{n+1}{i} B_i = 0 \quad (n \geq 1)$

この Bernoulli 数は漸化式による定義の他に, 形式的冪級数を用いて定義されます.

先ず, 歴史的な Bernoulli 数 ($B_1 = 1/2$) に対応する冪級数 (母函数) による定義を以下
に示しておきます:

$$\frac{se^s}{e^s - 1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} s^n \quad (|s| < 2\pi)$$

次に, Maxima で用いられているとなる Bernoulli 数 ($B_1 = -1/2$) の母函数による定
義を以下に示します:

$$\frac{s}{e^s - 1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} s^n \quad (|s| < 2\pi)$$

この事を makelist 函数と taylor 函数を用いて確認してみましょう.

```
(%o42)                                     true
(%i43) makelist(bern(n)/n!,n,0,10);
          1  1          1          1          1          1
(%o43)    [1, - -, --, 0, - ---, 0, -----, 0, - -----, 0, -----]
```

¹Bernoulli の「Ars Conjectandi」は 1713 年に出版, 関孝和の「括要算法」は正徳 2 年 (1712 年) に出版

```

      2 12      720      30240      1209600      47900160
(%i44) taylor(s/(exp(s)-1),s,0,10);
      2 4 6 8 10
      s s s s s s
(%o44)/T/ 1 - - + -- - --- + ----- - ----- + ----- + . . .
          2 12 720 30240 1209600 47900160

(%i45) taylor(s*exp(s)/(exp(s)-1),s,0,10);
      2 4 6 8 10
      s s s s s s
(%o45)/T/ 1 + - + -- - --- + ----- - ----- + ----- + . . .
          2 12 720 30240 1209600 47900160

```

初期状態で Bernoulli 数の 0 は除外されていません。もし、Bernoulli 数から 0 を除外する必要がある場合には、大域変数 `zerobern` を `false` に設定します。

猶、大域変数 `zerobern` を `false` にした場合、当然ながら、0 の Bernoulli 数が除外されて並び直されるので、大域変数 `zerobern` が `true` の場合と `false` の場合で `bern(n)` の値が異なる事に注意が必要です。

```

(%i38) zerobern:true;
(%o38) true
(%i39) makelist(bern(n)/n!,n,0,6);
      1 1      1      1
(%o39) [1, - -, --, 0, - ---, 0, -----]
          2 12      720      30240

(%i40) zerobern:false;
(%o40) false
(%i41) makelist(bern(n)/n!,n,0,6);
      1 1      1 5      691      7
(%o41) [1, - -, --, - ---, ----, - -----, ----]
          2 12      180 1584      327600 4320

```

この例で判る様に、大域変数 `zerobern` を変更する事で、`bern(n)` の値が異なって `n!` とのズレが発生する為に `bern(n)/n!` の値が異なる事になります。

`bernpoly` 関数は Bernoulli 多項式 $B_n(x)$ を生成する関数です。この Bernoulli 多項式 $B_n(x)$ は Bernoulli 数 B_n を用いて以下の式で定義される多項式です:

$$B_n(x) =_{\text{def}} \sum_{k=0}^n \binom{n}{k} B_{n-k} x^k$$

Maxima の `bernpoly` 函数は引数を二つ取り, 第一引数が多項式の変数, 第二引数が多項式の次数となります.

```
(%i50) makelist(bernpoly(x,n),n,0,4);
```

```
(%o50)      2
           1  2      1  3  3 x  x  4      3  2  1
[1, x - -, x - x + -, x - ---- + -, x - 2 x + x - --]
           2          6          2  2          30
```

更に, Bernoulli 多項式は次の性質を持っています:

- $B_n(0) = B_n$
- $\frac{d}{dx} B_n(x) = n B_{n-1}(x)$

```
(%i8) makelist(bernpoly(0,i),i,0,10)-makelist(bern(i),i,0,10);
```

```
(%o8)      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
(%i9) makelist(diff(bernpoly(x,i),x),i,1,10)-makelist(bernpoly(x,i-1)*i,i,1,10),expand;
```

```
(%o9)      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

この例では, $n = 10$ まで, $B_n(0) - B_n$ の値と $\frac{d}{dx} B_n(x) - n B_{n-1}(x)$ の値を確認するものです. b

Bernoulli 数に関連する大域変数

大域変数名	初期値	概要
<code>zerobern</code>	<code>true</code>	函数 <code>bern</code> の制御に関連

Bernoulli 数に 0 を含めるかどうかを指定する大域変数です. 真理値を設定し, `true` の場合には 0 を含め, `false` の場合には除外します.

8.1.5 beta 関数

— bete 関数 —

beta($\langle$ 式₁ $\rangle$, $\langle$ 式₂ $\rangle$)

beta 関数は次の関係式で定義された関数です:

$$\beta(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

猶,Maxima では内部関数の simpbeta 関数によって式の簡易化が実行されます. この simpbeta 関数の制御で大域変数 beta_args_sum_to_integer が用いられています.simp-beta 関数では makegamma 関数,binomial 関数を用いて式の簡易化が行われます.

8.1.6 二項係数

— binomial 関数 —

binomial($\langle$ 式₁ $\rangle$, $\langle$ 式₂ $\rangle$)
binomial($\langle$ 正整数値₁ $\rangle$, $\langle$ 正整数値₂ $\rangle$)

binomial 関数は二項係数を返す関数です. 二つの引数を評価した結果が共に正整数であれば, 通常の階乗を用いた計算で処理されますが, 双方が数値で, どちらかが実数の場合,gamma 関数を用いた処理になります. 従って,binomial 関数が返却する数値は整数型か浮動小数点数になります.

更に,Maxima の binomial 関数には整数値以外の式を与える事も可能です. この場合,binomial 関数は $\langle$ 式₁ $\rangle$ と $\langle$ 式₂ $\rangle$ の何れもが数値ではなく, $\langle$ 式₁ $\rangle - \langle$ 式₂ $\rangle$ が正整数となる場合のみに有理式を返却し, それ以外は名詞型で返却します.

又, 引数の双方が数値で, その内, どちらか一方が非整数の場合,gamma 関数を用いて計算を行います.

```
(%i14) binomial(17,3);
(%o14)                                     680
(%i15) binomial(17.0,3.0);
(%o15)                                     680.0000000000005
(%i16) gamma(18.0)/(gamma(15.0)*gamma(4.0));
```

```
(%o16)                                     680.0000000000005
(%i17) binomial((x-1)^3+11,(x-1)^3+7);
          3          3          3          3
      ((x - 1)  + 8) ((x - 1)  + 9) ((x - 1)  + 10) ((x - 1)  + 11)
(%o17)  -----
                      24
(%i18) binomial((x-1)^3+11,1.2);
          3
(%o18)  binomial((x - 1)  + 11, 1.2
```

8.1.7 euler 関数

euler 関数は Euler 数 E_n を返す関数です. この Euler 数 E_n は次の式で定義される数です:

$$\frac{1}{\cos x} =_{def} \sum_{n=0}^{\infty} \frac{E_n}{n!} x^n$$

euler 関数

euler(< 正整数値 >)

Euler-Mascheroni 定数

%gamma 0.5772156649015329...

この Euler-Mascheroni 定数の計算方法は Numbers, Constants and Computation [88] の Euler's constant γ で解説されている Bessel 関数を用いた方法を採用しています. この Euler の定数は $\gamma = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} - \log(n) \right)$ で計算される定数で, gamma 関数との関係では, $\Gamma'(1) = -\gamma$ となる性質があります.

8.1.8 Fibonacci 数

———— Fibonacci 数に関連する関数 ————

fib($\langle$ 正整数値 $\rangle$)
 fibtophi($\langle$ 式 $\rangle$)

猶, fib 関数は Fibonacci 数を返す関数です. この Fibonacci 数は $F_0 = 0, F_1 = 1, F_{n+1} = F_n + F_{n-1}$ という漸化式で定義される数です.

猶, fib 関数で一度計算を行うと, 一段前の Fibonacci 数が大域変数 prevfib に保存されています.

fibtophi 関数は fib(n) を $\frac{\pi^n - (1 - \pi)^n}{2\pi - 1}$ で置換える関数ですが, 引数が fib($\langle$ 式 $\rangle$) の書式であり, $\langle$ 式 $\rangle$ が整数以外の式でなければ, 引数をそのまま返す関数です.

```
(%i60) fibtophi(fib(x));
```

```
(%o60)
      x          x
      %phi  - (1 - %phi)
      -----
            2 %phi - 1
```

```
(%i61) fibtophi(fib(1+%i));
```

```
(%o61)
      %i + 1          %i + 1
      %phi  - (1 - %phi)
      -----
            2 %phi - 1
```

```
(%i62) fibtophi(x^3);
```

```
(%o62)
      3
      x
```

```
(%i63) fibtophi(fib(3));
```

```
(%o63)
      2
```

———— 大域変数 prevfib ————

prevfib 関数 fib で計算した Fibonacci 数の一つ前の値を保持

大域変数 prevfib には, fib($\langle n \rangle$) の計算の際に用いた fib($\langle n - 1 \rangle$) の値が保存されています.

```
(%i69) fib(31);
(%o69) 1346269
(%i70) prevfib;
(%o70) 832040
(%i71) fib(30);
(%o71) 832040
```

8.1.9 gamma 函数

gamma 函数に関連する函数

gamma(< 式 >)

gamma 函数は Γ 函数を Maxima に実装した函数です.

ここで, Γ 函数は以下の式で定義される函数です.

$$\Gamma(x) =_{\text{def}} \int_0^{\infty} t^{x-1} e^{-t} dt$$

gamma 函数は引数を評価して, その結果が整数であれば整数を, 浮動小数点数の場合は浮動小数点数を返し, それ以外の式に対しては, 名詞型を返す函数です.

猶, Γ 函数と正整数 n の階乗との間には, $\Gamma(n) = (n-1)!$ となる関係があります. この gamma 函数の簡易化では, 大域変数 `gammalim` が用いられます.

gamma 函数に関連する大域変数

大域変数	初期値	概要
<code>factlim</code>	-1	階乗の展開を制御
<code>gammalim</code>	1000000	gamma 函数の簡易化を制御

大域変数 `factlim` で正整数の階乗や正整数を引数とする gamma 函数の自動展開を制御します. この値を越える正整数の階乗は自動展開されません. 又, gamma 函数は引数の正整数から 1 を引いた値が大域変数 `factlim` の値を越えると自動展開されません. 猶, 大域変数 `factlim` が -1 の場合, 全ての正整数の階乗や正整数を引数とする gamma 函数が自動的に展開されます.

```
(%i25) factlim:10;
(%o25) 10
(%i26) 10!;
```

```
(%o26)                                     3628800
(%i27) 11!;
(%o27)                                     11!
(%i28) gamma(11);
(%o28)                                     3628800
(%i29) gamma(12);
(%o29)                                     11!
```

大域変数 `gammalim` は `gamma` 関数の簡易化で用いられる大域変数です. `gamma` 関数の引数の絶対値が `gammalim` よりも小さな値であれば簡易化を行います. 猶, `gamma` 関数の引数が正整数の場合, 大域変数 `factlim` も `gamma` 関数の簡易化に関わります. 形式的に `gamma` 関数を階乗で置換える関数があります.

————— `makegamma` 関数 —————

`makegamma(< 式 >)`

この `makegamma` 関数は階乗の名詞型を含む式に対し, 階乗の名詞型を形式的に `gamma` 関数で置換える関数です.

```
(%i67) makegamma(n!);
(%o67)                                     gamma(n + 1)
(%i68) makegamma(n!/m!);
(%o68)                                     gamma(n + 1)
                                     -----
                                     gamma(m + 1)
(%i69) makegamma(n!=m!*(m+1));
(%o69)                                     gamma(n + 1) = (m + 1) gamma(m + 1)
```

8.1.10 li 関数

————— `li` 関数 —————

`li(< 正整数 >)(< 正整数 >)`

`li[s](z)` で表現される式は次の関数です.

$$Li_s(z) =_{def} \sum_{k=1}^{\infty} \frac{z^k}{k^s}$$

8.1.11 Möbius の関数 μ

moebius 関数

moebius(< 正整数 >)

moebius 関数は Möbius 関数 μ を Maxima に実装したものです. この moebius 関数は引数として正整数を取りますが, 整数値リスト整数値集合や整数値行列も扱えます.

Möbius の関数 μ は次で定義出来ます:

Möbius の $\mu(n)$ 関数の定義

$$\begin{array}{ll} n = 1 & \rightarrow \mu(1) = 1 \\ n \text{ が素数の平方で割切れる} & \rightarrow \mu(n) = 0 \\ n \text{ が } k \text{ 個の異なる素数の積である} & \rightarrow \mu(n) = (-1)^k \end{array}$$

この Möbius の $\mu(n)$ 関数が満す重要な性質に次のものがあります:

$$\sum_{d|n} \mu(d) = \begin{cases} 1 & n = 1 \text{ の場合} \\ 0 & n > 1 \text{ の場合} \end{cases}$$

更に, Möbius の μ 関数は Riemann の ζ 関数の逆数にも現れる関数です:

$$\frac{1}{\zeta(s)} = \sum_{n=1}^{\infty} \frac{\mu(n)}{n^s}$$

moebius 関数は正整数 n に対して $\mu(n)$ を返す関数ですが, この関数は方程式, リスト, 集合や行列に対して分配されます.

8.1.12 numfactor 関数

numfactor 関数

numfactor (< 式 >)

< 式 > から数値の因数のみを取り出す関数です. ここで < 式 > は単項式でなければなりません. 与式が原子の場合や, 数値因子が無い場合には 1 を返します. 猶, ここでの数値は numberp 関数が true を返す対象の事です. 従って, declare 関数で integer 等と宣言した原子は含まれません.

猶, 和の中の全ての項の係数の gcd が必要ならば, content 関数を用いましょう.

```
(c1) gamma(7/2);
(d1)          15 sqrt(%pi)
          -----
              8

(c2) numfactor(%)
          15
(d2)      --
          8
```

8.1.13 ψ 関数

psi 関数

psi[$\langle$ 正整数 $\rangle$]($\langle$ 式 $\rangle$)

psi[n](x) は $\log(\Gamma(x))$ の第 $n + 1$ 階の微分です.

8.1.14 ζ 関数

zeta 関数に関連する関数

zeta($\langle$ 正整数値 $\rangle$)
 bfzeta($\langle$ 変数 $\rangle$, $\langle$ 正整数値 $\rangle$)
 bfhzeta($\langle$ 変数₁ $\rangle$, $\langle$ 変数₂ $\rangle$, $\langle$ 正整数値 $\rangle$)

zeta は Riemann の ζ 関数です. この Riemann の ζ 関数は次の式で定義されます:

$$\zeta(s) =_{def} \sum_{n=1}^{\infty} \frac{1}{n^s}$$

又, Γ 関数を用いると, 以下の式で定義が可能です:

$$\zeta(s) =_{def} \frac{1}{\Gamma(s)} \int_0^{\infty} \frac{u^{s-1}}{e^u - 1} du$$

更に, 素数との関係では次の式 (Euler 積) が知られています.

$$\zeta(s) = \prod_{p \in \{ \text{素数の集合} \}} \frac{1}{1 - p^{-s}}$$

この ζ 函数で有名な未解決問題が Riemann 予想です.

Riemann 予想

複素平面上の $\zeta(s)$ の非自明な零点の実部は全て $\frac{1}{2}$ である.

猶, 自明な零点は負の偶数 $(-2, -4, \dots)$ で, 非自明な零点は $0 < \operatorname{Re}(s) < 1$ のみである事が知られています.

この zeta 函数は引数が整数, 有理数, 浮動小数点数の以外の型であれば, 名詞型の式を返す函数です.

bfbzeta 函数と bfhzeta 函数は bffac パッケージに含まれる函数です. これらの函数を利用する為には, 予め, `load(bffac)` によるパッケージの読込が必要です. これらの函数は bfloat を返却値とする函数です.

先ず, bfbzeta 函数は引数を二つ取り, 第一引数の第二引数で指定した桁迄の ζ 函数の値を返します. 猶, この函数で第一引数は整数, 有理数, 超越数と代数的数を含む無理数, そして, 第二引数は正整数でなければなりません.

bfhzeta 函数は Hurwitz の zeta 函数で, $\operatorname{bfhzeta}(s, h, n)$ で $\sum_{k=0}^{\infty} (k+h)^{-s}$ の n 桁迄を計算します.

zeta 函数に関連する大域変数

大域変数	初期値	概要
<code>zeta%pi</code>	true	zeta 函数の引数 n が偶数の場合, π^n を表示するかどうかを指定

8.1.15 連分数に関連する函数

連分数に関連する函数

<code>cf(<式>)</code>
<code>cfexpand(<リスト>)</code>
<code>cfdisrep(<リスト>)</code>

cf 函数は与えられた式の連分数による表現をリストの形式で出力する函数です. cf 函数が出力するリストの長さを制御する大域変数が cflength です.

猶,cf 関数の引数として利用可能なのは, 整数, 有理数, 浮動小数点数と代数的数であり, それ以外の超越数や複素数等の値や式に対してはエラーを返します.

cfdisrep 関数は cf 関数で生成した連分数のリスト表現を通常の連分数の表示に変換する関数です. 内部的には与えられたリストを有理数に変換する関数です.

```
(%i31) is(cfdisrep(cf(1.20))-1.2=0),pred;
(%o31)                                     false
(%i32) is(cfdisrep(cf(1.20))-12/10=0),pred;
(%o32)                                     true
```

この例では, 浮動小数点数が cf 関数によって整数リストで置換えられ, 更に,cfdisrep 関数で有理数に変換された為, 本来の浮動小数点数の差を取っても, 型の違いによって 0 にならない事を示しています.

—— 連分数に関連する大域変数 ——

大域変数	初期値	概要
cflength	1	cf 関数の出力を制御

cflength 関数は cf 関数が出力する連分数の段数, 即ち, リスト長を制御する関数です. 初期値は 1 となっており, この場合,cf 関数が出力する段数は 1, リスト長では cflength+1 になります.

```
(%i1) cf(sqrt(2));
(%o1)                                     [1, 2]
(%i2) cfdisrep(%);
                                     1
(%o2)                                     1 + -
                                     2

(%i3) cflength:5;
(%o3)                                     5
```

```
(%i4) cfdisrep(cf(sqrt(2)));
```

```
(%o4)
1
1 + -----
1
2 + -----
1
2 + -----
1
2 + -----
1
2 + -
2
```

8.1.16 二次体に関連する函数

二次体に関連する函数

qunit(< 正整数 >)

qunit 函数は一つの引数を取り, その引数が正整数 $\langle n \rangle$ の場合に, ノルムが 1 となる二次体 $\mathbb{Z}[\sqrt{\langle n \rangle}]$ の元, 即ち, $a^2 - nb^2 = 1$ (Pell の方程式) を満し, $b > 0$ となる元 $a + b\sqrt{\langle n \rangle}$ を返す函数です.

```
(%i72) qunit(23);
```

```
(%o72) 5 sqrt(23) + 24
```

```
(%i73) %*substpart(-1*part(% ,1),%,1);
```

```
(%o73) (24 - 5 sqrt(23)) (5 sqrt(23) + 24)
```

```
(%i74) %,expand;
```

```
(%o74) 1
```

猶, この例では, substpart 函数と part 函数を用いて $5\sqrt{23}+24$ を $-5\sqrt{23}+24$ に変換していますが, これは Maxima の差の内部表現が $a+(-b)$ となっている為, subst("-",%,0) とするとエラーになるので, この様な代入を行っています. 猶, qunit 函数は解の計算を行う為に内部で isqrt 函数を用いています.

8.1.17 ifactor パッケージに含まれる関数

ifactor パッケージに含まれる関数

```
ifactor(< 正整数 >)
primep(< 正整数 >)
next_prime(< 正整数 >)
prev_prime(< 正整数 >)
power_mod(< 正整数1>, < 正整数2>, < 正整数3>)
inv_mod(< 正整数1>, < 正整数2>)
```

ifactor パッケージには内部変数*small-primes*に 9973 以下の素数のリストが束縛されています。又、内部変数*largest-small-primes*に内部変数*small-primes*に束縛された素数リストの最大元 9973 が束縛されています。

ifactors 関数

ifactors 関数は正整数に対し、素数分解を行い、素数と冪のリストを成分とするリストを返す関数です。

即ち、 $n = p_1^{k_1} \cdots p_m^{k_m}$ に対し、ifactor 関数は、素数と冪次数の対のリスト $[[p_1, k_1], \dots, [p_m, k_m]]$ を返します。

大域変数 `ifactor_verbose`²が `true` の場合、ifactor 関数は詳細な報告を返します。

```
(%i13) ifactors(818378923834);
(%o13)          [[2, 1], [97, 1], [42283, 1], [99767, 1]]
(%i14) ifactor_verbose:true;
(%o14)                                     true
(%i15) ifactors(818378923834);
```

```
Starting factorization of n = 818378923834
Factoring out small prime: 2 (degree:1)
Factoring out small prime: 97 (degree:1)
Factoring n = 4218448061
Pollard rho: round #1 of 5 (lim=10000)
Pollard rho: found factor 42283 (5 digits)
```

²`ifactors_verbose` でない事に注意!紛らわし事に、この変数だけ ifactor に s がありません。

```
=====> Prime factor: 42283
```

```
=====> Prime factor: 99767
```

```
(%o15)          [[2, 1], [97, 1], [42283, 1], [99767, 1]]
```

大域変数 `factors_only` が `true` の場合, $n = p_1^{k_1} \cdots p_m^{k_m}$ に対し, `ifactor` 関数は, 素数の冪のリスト $[p_1^{k_1}, \dots, p_m^{k_m}]$ で返します.

```
(%i12) factors_only:true;
```

```
(%o12)          true
```

```
(%i13) ifactors(8218344);
```

```
(%o13)          [2, 3, 17, 20143]
```

```
(%i14) factors_only:false;
```

```
(%o14)          false
```

```
(%i15) ifactors(8218344);
```

```
(%o15)          [[2, 3], [3, 1], [17, 1], [20143, 1]]
```

猶, この素因数分解では楕円曲線を用いた手法 (Elliptic Curve Method(ECM)) を採用しています³.

この関数では, Pollard の Rho 素数判定法と楕円曲線を用いる ECM が用いられています. 最初に, Pollard の Rho 素数判定法が適用され, 次に ECM が利用されます.

ここで, Pollard の Rho 素数判定法を制御する大域変数には, 大域変数 `pollard_rho_limit_step`, 大域変数 `pollard_rho_limit` と大域変数 `pollard_rho_tests` があります. そして, ECM を制御する大域変数は大域変数 `ecm_number_of_curves`, 大域変数 `ecm_limit`, 大域変数 `ecm_max_limit`, 大域変数 `ecm_limit_delta` になります.

primep 関数

`primep` 関数は与えられた正整数に対し, 素数であるかどうかを判定する真理関数です. この真理値集合は $\{\text{true}, \text{false}\}$ です.

`primep` 関数は 1 から 17 迄の素数に対しては直接表を参照し, 341550071728321 よりも小さな正整数に対しては内部関数の `primep-small` を用い, それ以上の正整数に対しては内部関数の `primep-prob` を用います.

³<ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/rpb161.dvi.gz> 参照

これらの関数での素数判定は内部関数の miller-rabin 関数が用いられています。この miller-rabin 関数は文字通りに Miller-Rabin 素数判定法を用いて判定を行う関数です。猶,Miller-Rabin 素数判定法で合成数を素数と判定する可能性が皆無ではありません。先ず,primep-small 関数では 2 以上 17 以下の素数のリストを用いて Miller-Rabin 素数判定法を用いています。

そして,primep-prob 関数では大域変数 primep_number_of_tests で指定した回数程,Miller-Rabin 素数判定を実行します。ここで素数と判定されると、最後に primep-lucas 関数による Lucas 素数判定法を用います。

この primep-lucas 関数内部で、大域変数 save_primes が true の場合、Lucas 素数判定法を通った正整数は内部変数*large-primes*に追加されます。

```
(%i2) :lisp *large-primes*
NIL
(%i2) primep(next_prime(8183789238342905897737));
(%o2)                                     true
(%i3) :lisp *large-primes*
NIL
(%i3) save_primes:true;
(%o3)                                     true
(%i4) primep(next_prime(8183789238342905897737));
(%o4)                                     true
(%i5) :lisp *large-primes*
(8183789238342905897863)
(%i5) primep(next_prime(8183789238342905897737));
(%o5)                                     true
(%i6) :lisp *large-primes*
(8183789238342905897863)
(%i6) primep(next_prime(81837892383429058977887892337));
(%o6)                                     true
(%i7) :lisp *large-primes*
(81837892383429058977887892353 8183789238342905897863)
```

next_prime 関数

与えられた正整数を越える素数で最小の素数を返す関数です。与える正整数は素数でも合成数でも構いません。この関数の内部で、素数判定に `primep` 関数を用いています。従って、341550071728321 よりも大きな素数を求めた場合、大域変数 `save_primes` が `true` であれば、内部変数 `*large-primes*` に、その素数が登録されます。

```
(%i4) save_primes:true;
(%o4)                                     true
(%i5) next_prime(34155007172832);
(%o5)                                     34155007172837
(%i6) :lisp *large-primes*
NIL
(%i6) next_prime(341550071728321);
(%o6)                                     341550071728361
(%i7) :lisp *large-primes*
(341550071728361)
```

prev_prime 関数

与えられた正整数を越えない素数で最大の素数を返す関数です。与える正整数は素数でも合成数でも構いません。この関数では素数判定で `primep` 関数を用いています。従って、341550071728321 を越える素数を求めた場合、大域変数 `save_primes` が `true` であれば、内部変数 `*large-primes*` に、その素数が登録されます。

```
(%i1) save_primes:true;
(%o1)                                     true
(%i2) prev_prime(341550071728321);
(%o2)                                     341550071728289
(%i3) :lisp *large-primes*
NIL
(%i3) prev_prime(3415500717283210);
(%o3)                                     3415500717283163
(%i4) :lisp *large-primes*
(3415500717283163)
```

power_mod 関数

power_mod 関数は 3 個の引数を取る関数で, `power_mod(a,n,m)` は $a^n \bmod m$ を計算します.

inv_mod 関数

inv_mod 関数は二つの正整数を引数に取り, `inv_mod(m,n)` で `mod(m,n)` の逆元を返します. ここで, `m` が環 $\mathbb{Z}[n]$ で正則元 (逆元を持たない元) の場合, 例えば, `n` が `m` の倍数の場合, 或いは, `m` が `n` の倍数の場合に, `false` を返却します.

```
(%i104) makelist(inv_mod(x,5),x,1,4);
(%o104) [1, 3, 2, 4]
(%i105) makelist(mod(mod(x,5)*inv_mod(x,5),5),x,1,4);
(%o105) [1, 1, 1, 1]
(%i106) makelist(inv_mod(x,4),x,1,4);
(%o106) [1, false, 3, false]
```

8.1.18 ifactor パッケージに含まれる大域変数

ifactor パッケージに含まれる大域変数

<code>ifactor_verbose</code>	false	ifactors 関数の動作報告表示を制御
<code>factors_only</code>	false	ifactors 関数の出力形式を制御
<code>save_primes</code>	false	内部変数*large-primes*への素数の登録を制御
<code>primep_number_of_tests</code>	25	Miller-Rabin 素数判定法の適用回
<code>pollard_rho_limit</code>	10000	pollard_rho による判定で利用
<code>pollard_rho_tests</code>	5	pollard_rho による判定で利用
<code>pollard_rho_limit_step</code>	1000	pollard_rho による判定で追加されるステップ数
<code>ecm_number_of_curves</code>	50	ECM による反復回数
<code>ecm_limit</code>	200	ECM による判定条件
<code>ecm_max_limit</code>	51199	ECM による判定条件
<code>ecm_limit_delta</code>	200	ECM で追加されるステップ数

大域変数 `ifactor_verbose` が `true` であれば, `ifactors` 関数は結果リストを表示させるだけでなく, より詳細な報告を表示させます. 猶, この大域変数のみ `ifactors` の様に `s` が付かないので注意して下さい.

大域変数 `factors_only` が `true` の場合, `ifactor` 関数は素数の冪のリストとして出力を行います.

大域変数 `save_primes` が `true` であれば, 内部変数 `*large-primes*` に `primep` 関数で判定した 341550071728321 よりも大きな素数を `cons` で追加します. この大域変数は Maxima の変数に影響を与えない為, その影響が表から見え難い大域変数です.

大域変数 `primep_number_of_tests` は, `primep` 関数による素数判定で, 判定する数が 341550071728321 よりも大きな場合に用いられる内部関数 `primep-prob` での Miller-Rabin 素数判定法を適用する回数を指定します.

先頭に `pollard` が付く大域変数は Pollard の素数判定アルゴリズムで用いられる大域変数です.

大域変数 `pollard_rho_limit` は `ifactor` 関数が呼出す内部関数 `get-one-factor-pollard` 関数で用いられる大域変数です. 内部の判定処理で用いられる大域変数です.

大域変数 `pollard_rho_limit_step` は, `get-one-factor-pollard` 関数内部の反復処理で, 大域変数 `pollard_rho_limit` と組合せて用いられます. この大域変数は `pollard_rho_limit` で不十分な場合に追加される反復処理のステップ数になります.

```
(%i24) ifactors(913284098373894758374598298931);
```

```
Starting factorization of n = 913284098373894758374598298931
```

```
Factoring n = 913284098373894758374598298931
```

```
Pollard rho: round #1 of 20 (lim=10000)
```

```
Pollard rho: round #2 of 20 (lim=11000)
```

```
Pollard rho: round #3 of 20 (lim=12000)
```

```
Pollard rho: round #4 of 20 (lim=13000)
```

```
Pollard rho: round #5 of 20 (lim=14000)
```

```
Pollard rho: found factor 11772548497 (11 digits)
```

```
=====> Prime factor: 11772548497
```

```
=====> Prime factor: 77577433518887355523
```

```
(%o24) [[11772548497, 1], [77577433518887355523, 1]]
```

```
(%i25) pollard_rho_limit:100;
```

```
(%o25)                                     100
(%i26) ifactors(913284098373894758374598298931);

Starting factorization of n = 913284098373894758374598298931
Factoring n = 913284098373894758374598298931
Pollard rho: round #1 of 20 (lim=100)
Pollard rho: round #2 of 20 (lim=1100)
Pollard rho: round #3 of 20 (lim=2100)
Pollard rho: round #4 of 20 (lim=3100)
Pollard rho: round #5 of 20 (lim=4100)
Pollard rho: round #6 of 20 (lim=5100)
Pollard rho: round #7 of 20 (lim=6100)
Pollard rho: round #8 of 20 (lim=7100)
Pollard rho: round #9 of 20 (lim=8100)
Pollard rho: round #10 of 20 (lim=9100)
Pollard rho: round #11 of 20 (lim=10100)
Pollard rho: round #12 of 20 (lim=11100)
Pollard rho: round #13 of 20 (lim=12100)
Pollard rho: round #14 of 20 (lim=13100)
Pollard rho: round #15 of 20 (lim=14100)
Pollard rho: round #16 of 20 (lim=15100)
Pollard rho: round #17 of 20 (lim=16100)
Pollard rho: round #18 of 20 (lim=17100)
Pollard rho: round #19 of 20 (lim=18100)
Pollard rho: round #20 of 20 (lim=19100)
ECM: trying with curve #1 of 50 (lim=200)
ECM: trying with curve #2 of 50 (lim=400)
ECM: trying with curve #3 of 50 (lim=600)
ECM: trying with curve #4 of 50 (lim=800)
ECM: trying with curve #5 of 50 (lim=1000)
ECM: trying with curve #6 of 50 (lim=1200)
ECM: trying with curve #7 of 50 (lim=1400)
ECM: trying with curve #8 of 50 (lim=1600)
ECM: found factor in stage 2: 11772548497 (11 digits)
=====> Prime factor: 11772548497
```

```
=====> Prime factor: 77577433518887355523
```

```
(%o26)          [[11772548497, 1], [77577433518887355523, 1]]
```

この例からも, `pollard_rho_limit` を初期値の 10000 から 100 に変更した為, 自動的に上限 (lim) を増やして処理している事が判ります.

大域変数 `pollard_rho_tests` は `ifactor` 函数が呼出す内部函数の `get-one-factor` 函数で, 大きな正整数を分解する際に, 内部函数 `get-one-factor-pollard` 函数による反復処理の上限を定めます.

最後に頭に `ecm` が付く大域変数は楕円曲線を用いた素数判定法 (Elliptic Curve Method (ECM)) で用いられる大域変数です.

大域変数 `ecm_number_of_curves` は内部函数 `get-one-factor-ecm` 函数の反復処理回数を定める大域変数になります.

大域変数 `ecm_limit`, 大域変数 `ecm_max_limit` は ECM による素数の検出を行う上での判定条件を定める大域変数になります. 大域変数 `ecm_limit_delta` は大域変数 `ecm_max_limit` と組合せて用いられる大域変数で, 必要に応じて計算の反復処理数を増加させる場合に, 用いられます.

8.1.19 numth パッケージ

numth パッケージに含まれる函数

```
divsum(< 整数 >)
divsum(< 整数1>, < 整数2>)
totient(< 整数 >)
jaccobi(< 整数1>, < 整数2>)
gcfactor(< 式 >)
```

`divsum` 函数は引数が一つの場合, 引数の因子の和を返します. 引数が二つの場合, 第一引数の因子を第二引数乗したものの和を返します.

```
(%i24) divsum(128);
(%o24)                                     255
(%i25) makelist(2^i,i,0,7);
(%o25)          [1, 2, 4, 8, 16, 32, 64, 128]
```

```
(%i26) substpart("+",%,0);
(%o26)                                     255
(%i27) divsum(128,4);
(%o27)                                     286331153
(%i28) makelist(2^(4*i),i,0,7);
(%o28)      [1, 16, 256, 4096, 65536, 1048576, 16777216, 268435456]
(%i29) substpart("+",%,0);
(%o29)                                     286331153
```

関数 divsum は大域変数 intfaclim の影響を受けます.

totient 関数は与えた正整数以下の整数で, 与えた正整数と互いに素となる整数の数を返します.

```
(%i6) totient(10);
(%o6)                                     4
(%i7) totient(11);
(%o7)                                     10
```

この例では, 10 と互いに素な整数は 3, 5, 7, 9 の 4 個, 11 は素数の為, 11 を除く 10 個の数と互いに素となります. この関数の内部では factor 関数が用いられており, 大域変数 intfaclim で制御されます.

jacobi 関数は Jacobi 記号を計算する関数です. Jacobi 記号は Legendre 記号 $(a|b)$, $\displaystyle\left(\frac{a}{b}\right)$

を拡張したものです. 先ず, Legendre 記号は整数 $n = \prod_{i=1}^k p_i^{e_i}$ が与えられた時,

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)_1^e \left(\frac{a}{p_2}\right)_2^e \cdots \left(\frac{a}{p_k}\right)_k^e$$

を満たします. 猶, 任意の整数 a に対し, $\left(\frac{a}{1}\right) = 1$ となります.

Jacobi の記号は次の性質を満たすものです:

Jacobi の記号の性質

- $\left(\frac{a}{n}\right) = 0$ $\gcd(a, n) > 1$
- $\left(\frac{-1}{n}\right) = 1$ $n = 1 \pmod{4}$
- $\left(\frac{-1}{n}\right) = -1$ $n = 3 \pmod{4}$
- $\left(\frac{a}{n}\right) \left(\frac{b}{n}\right) = \left(\frac{ab}{n}\right)$
- $\left(\frac{a}{m}\right) \left(\frac{a}{n}\right) = \left(\frac{a}{mn}\right)$
- $\left(\frac{a}{n}\right) = \left(\frac{b}{n}\right)$ $a = b \pmod{n}$

gcfactor 函数は Gauß 整数上で多項式の因子分解を行います.

```
(%i23) gfactor(x^7+1);
```

```
(%o23)          6      5      4      3      2
              (x + 1) (x  - x  + x  - x  + x  - x + 1)
```

```
(%i24) gfactor(x^6+1);
```

```
(%o24)          2              2
              (x - %i) (x + %i) (x  - %i x - 1) (x  + %i x - 1)
```

猶,gcfactor 函数内部では jacobi 函数が用いられています.

8.1.20 Kronecker の δ と Stirling 数

nset パッケージに含まれる数論関連の函数

```
kron(< 式1>,< 式2>)
stirling1(< 正整数値1>,< 正整数値2>)
stirling2(< 正整数値1>,< 正整数値2>)
```

kron_delta は引数を二つ取る整数値函数です. 基本的に Kronecker の δ を表現する函数です. 従って, 二つの引数が等しい場合には 1 を返し, そうでない場合には, 0 を返します. 但し, 等しいかどうか判別出来ない場合には, 名詞型で返します.

```
(%i112) kron_delta(1,0);
```

```
(%o112)          0
```

```
(%i113) kron_delta("a","a");
(%o113) 1
(%i114) kron_delta(x^2+2*x+1,(x+1)^2);
(%o114) 1
(%i115) kron_delta(integrate(x*2,x),x^2);
(%o115) 1
(%i116) kron_delta(x,y);
(%o116) kron_delta(x, y)
(%i117) kron_delta(x,y),x=y;
(%o117) 1
```

stirling1 関数と stirling2 関数は夫々第一種と第二種の Stirling 数を返す関数です。猶、Maxima では、これらの Stirling 数には以下の同値関係が設定してあります。

—— stirling1 関数と stirling2 関数の同値関係 ——

(1 _a)	stirling1 (0, n)	~	kron_delta(0,n)
(2 _a)	stirling1 (n, n)	~	1
(3 _a)	stirling1 (n, n - 1)	~	binomial(n,2)
(4 _a)	stirling1 (n + 1, 0)	~	0
(5 _a)	stirling1 (n + 1, 1)	~	n!
(6 _a)	stirling1 (n + 1, 2)	~	2 ⁿ - 1
(1 _b)	stirling2 (0, n)	~	kron_delta(0,n)
(2 _b)	stirling2 (n, n)	~	1
(3 _b)	stirling2 (n, n - 1)	~	binomial(n,2)
(4 _b)	stirling2 (n + 1, 0)	~	0
(5 _b)	stirling2 (n + 1, 1)	~	1
(6 _b)	stirling2 (n + 1, 2)	~	2 ⁿ - 1
(7 _b)	stirling2 (n,0)	~	kron_delta(n,0)
(8 _b)	stirling2 (n,m)	~	0 (m > n の場合)

8.2 三角函数

8.2.1 三角函数一覧

Maxima は沢山の三角函数を持っています. 三角函数の恒等式, 即ち, $\cos^2(x) + \sin^2(x) = 1$ や $\cos(2x) = 2\cos^2(x) - 1$ の様なものは予め Maxima に組込まれていますが, 多くの恒等式を規則として利用者が付加する事が出来ます.

Maxima で予め定義された三角函数は下記のものがあります.

三角函数と双曲函数

cos	余弦函数
cosh	双曲線余弦函数
cot	余接函数
coth	双曲線余接函数
csc	余割函数
csch	双曲線余割函数
sec	正割函数
sech	双曲線正割函数
sin	正弦函数
sinh	双曲線正弦函数
tan	正接函数
tanh	双曲線正接函数

逆三角関数と逆双曲関数

関数名	概要
acos	逆余弦関数
acosh	逆双曲線余弦関数
acot	逆余接関数
acoth	逆双曲線余接関数
acsc	逆余割関数
acsch	逆双曲線余割関数
asec	逆正割関数
asech	逆双曲線正割関数
asin	逆正弦関数
asinh	逆双曲線正弦関数
atan	逆正接関数
atan2	逆正接関数
atanh	逆双曲線正接関数

猶, 逆正接関数には atan 関数 と atan2 関数の二種類がありますが, atan2 関数は, atan2(y,x) の様に引数を二つ必要とする関数で, 区間 $(-\pi, \pi)$ の間で atan(y/x) を計算します.

三角関数は, 倍角公式や角の和の公式等の様々な公式を持っています. これらを自動的に入力した式に適応する事も可能です. この場合, 大域変数 trigexpand を true にした状態で, 角の整数倍には, 大域変数 trigexpandtimes, 角の和に対しては大域変数 trigexpandplus を各々 true に設定すると, 角の和と積の公式を用いて, 与式の自動展開を行います.

```
(%i43) x+sin(5*x)/cos(x),trigexpand=true,expand;
          5
      sin (x)
----- - 10 cos(x) sin (x) + 5 cos (x) sin(x) + x
      cos(x)

(%o43)
(%i44) trigexpand(cos(3*x+2*y));
(%o44)      cos(3 x) cos(2 y) - sin(3 x) sin(2 y)
(%i45) trigexpand:true;
(%o45)
          true
(%i46) trigexpandtimes:true;
```



```
(%o46)                                     true
(%i47) trigexpandplus:true;
(%o47)                                     true
(%i48) cos(3*x+2*y);
          3          2          2          2
(%o48) (cos (x) - 3 cos(x) sin (x)) (cos (y) - sin (y))
          2          3
        - 2 (3 cos (x) sin(x) - sin (x)) cos(y) sin(y)
```

この例で、最初の `x+sin(5*x)/cos(x),trigexpand=true,expand` は `ev` 関数による評価の書式の一つで、与式 $x + \frac{\sin(5x)}{\cos(x)}$ を大域変数 `trigexpand` を `true` にした状態で展開を行う事を意味します。この表記は Maxima のトップレベルだけで利用可能です。猶、`ev` 関数の詳細に関しては、§5.8.2 を参照して下さい。

三角関数の半角公式に関しては、大域変数 `halfangles` を `true` にすると式の自動展開を実行します。この変数は大域変数 `trigexpand` の影響は受けません。

これらの変数とは別に、三角関数の引数の `declare` 関数による宣言に従って、三角関数を含む式の自動簡易化も行えます。

```
(%i2) declare(i,integer,a,even,b,odd);
(%o2)                                     done
(%i3) sin(x+(a+1/2)*%pi);
(%o3)                                     cos(x)
(%i4) sin(x+(b+1/2)*%pi);
(%o4)                                     - cos(x)
(%i5) cos(x+b*2*i*%pi);
(%o5)                                     cos(x)
```

この例では、最初に変数 `i` を整数、`a` を偶数、`b` を奇数として宣言しています。すると、Maxima は三角関数の引数を評価し、式を自動的に変換しています。

三角関数に付随する関数は、大域変数の `trigexpand`, `trigreduce` と `trigsign` を参照して下さい。二つの `share` パッケージは Maxima に組み込みの簡易化の規則の `trig` と `atrig` を拡張します。

三角関数に関連する大域変数

変数名	初期値	概要
<code>%piargs</code>	true	<code>%pi</code> と有理数の積を引数とする三角関数の簡易化を制御
<code>%iargs</code>	true	<code>%i</code> と有理数の積を引数とする三角関数の簡易化を制御
<code>halfangles</code>	false	半角公式の自動適用を制御
<code>trigexpandplus</code>	true	和公式の自動適用を制御
<code>trigexpandtimes</code>	true	角度の積による展開を制御
<code>triginverses</code>	all	逆関数との合成による簡易化を制御
<code>trigsign</code>	true	負の引数の簡易化を制御

大域変数 `%piargs` が true の場合, $\sin(\pi/2)$ の様な式を実数に変換します. 大域変数 `%iargs` が true の場合, $\sin(i \cdot x)$ の様に, 純虚数を引数に持つ三角関数を双曲関数に変換します.

大域変数 `halfangles` が true の場合, 半角 $\frac{\theta}{2}$ に対して簡易化が実行されます. この変数は大域変数 `trigexpand` の影響は受けません.

大域変数 `trigexpandplus` は和の規則を制御する大域変数です. つまり, 大域変数 `trigexpand` に true が設定されている時に, 引数の和を含む $\sin(x + y)$ の様な三角関数の自動展開が, 大域変数 `trigexpandplus` が true の場合に限り実行されます.

大域変数 `trigexpandtimes` は `trigexpand` 関数の積規則を制御します. 大域変数 `trigexpand` が true に設定されている時に, 三角関数の引数が整数, 或いは有理数倍の式に対し, 三角関数の自動展開が実行されます.

`triginverses` は三角関数, 双曲関数とその逆関数との合成の簡易化を制御します.

- all
両方, 例えば, $\operatorname{atan}(\tan(x))$ と $\tan(\operatorname{atan}(x))$ の両方が x に簡易化されます.
- true
 $\operatorname{arcfunction}(\operatorname{function}(x))$ の簡易化が切り捨てられます.
- false
 $\operatorname{arcfunc}(\operatorname{func})$ と $\operatorname{fun}(\operatorname{arcfun}(x))$ の簡易化が切り捨てられます.

`trigsign` が true であれば三角関数に対し負の引数の自動簡易化を行います. 例えば, `trigsign` が true の時に限り, $\sin(-x)$ は $-\sin(x)$ に変換されます.

8.2.2 三角函数に関連する函数

—— 三角函数の展開と簡易化に関連する函数 ——

```
trigexpand(<式>)
trigreduce(<式>,<変数>)
trigsimp(<式>)
trigrat(<三角函数を含む式>)
```

trigexpand 函数は <式> に含まれる三角函数や双曲函数に対し、倍角公式等を適用して式の展開を実行します。最良の結果を得る為に、予め expand 函数等で <式> を展開しておきましょう。

簡易化の利用者制御を拡張する為、この函数は一度に一つのレベルのみの角の和と角の積の展開を行います。sin と cos の全体の自動展開が必要ならば、大域変数 trigexpand を true に設定しておきます。

trigreduce 函数は <変数> の積を持つ三角函数と双曲 sin 函数と cos 函数の積と冪乗を結合します。分母で現われたこれらの函数を消去する事も試みます。猶、<変数> が省略されると、<式> の全ての変数が利用されます。

```
(%i1) trigreduce(-sin(x)^2+3*cos(x)^2+x);
```

```
(%o1)      cos(2 x)      cos(2 x)      1      1
----- + 3 (----- + -) + x - -
          2          2          2          2
```

trigsimp 函数は tan, sec 等を含む <式> の簡易化の為に、恒等式 $\sin^2(x) + \cos^2(x) = 1$ と $\cosh^2(x) - \sinh^2(x) = 1$ を使って、sin, cos, sinh, cosh へと変換します。trigreduce と組合せる事で、より良い簡易化が得られます。

```
(%i6) trigreduce(trigsimp(-sin(x)^2+3*cos(x)^2+x));
```

```
(%o6)      cos(2 x)      1
4 (----- + -) + x - 1
          2          2
```

```
(%i7) trigsimp(trigreduce(-sin(x)^2+3*cos(x)^2+x));
```

```
(%o7)      2 cos(2 x) + x + 1
```

trigrat 函数は sin, cos, tan 等の三角函数による有理式の正規簡易化を与えます。与式に含まれるこれらの三角函数の引数は、変数、有理数と %pi による線形結合とします。trigrat の計算結果は簡易化された sin と cos を含む有理式となります。

```
(%i10) trigexpand((cos(2*x+%pi*4/3*y+%pi/2)+sin(2*y+x))/(cos(x)+sin(y)));
              4 %pi y              4 %pi y
(%o10) (- cos(2 x) sin(-----) - sin(2 x) cos(-----) + cos(x) sin(2 y)
              3                  3
              + sin(x) cos(2 y))/(sin(y) + cos(x))
```

8.2.3 atrig1 パッケージ

atrig1 パッケージには逆三角関数に対する幾つかの追加の簡易化規則が含まれています。Maxima で既知の規則と共に、次の角が実装されています。

0, %pi/6, %pi/4, %pi/3, %pi/2

他の3つの象限に於ける角度でも利用可能です。猶、このパッケージを利用する為には予め `load(atrig1);` を実行する必要があります。

8.3 指数関数と対数関数

8.3.1 指数関数と対数関数の概要

Maxima には指数関数 `exp` と対数関数 `log` と `plog` 関数があります.

—— 指数関数と対数関数 ——

```
exp(< 式 >)
log(< 式 >)
plog(< 式 >)
```

指数関数 `exp` は Maxima 内部では, 自然底 e の冪として表現されています.

`log` 関数と `plog` 関数は違いが判り難い関数です. 勿論, 内部書式が異っていますが, `plog` 関数は `log` 関数の機能を内包する関数であり, 引数が複素数の場合の処理で大きく異なります.

```
(%i24) log(%i);
(%o24)                                     log(%i)

(%i25) plog(%i);
(%o25)                                     %i %pi
                                           -----
                                           2

(%i26) log(-(abs(x+1))^2);
(%o26)                                     2
                                           log(- (x + 1) )

(%i27) plog(-(abs(x+1))^2);
Is x + 1 zero or nonzero?

pos;
(%o27)                                     2 log(x + 1) + %i %pi
```

これは処理で用いる内部関数の特性の違いによるもので, `log` 関数は `simpln` 関数, `plog` 関数は `simplplog` 関数と `simpln` 関数の双方を用います. ここで, `log` 関数で入力される一般の式は正であると仮定した面があり, それに対し, `plog` 関数では正值性を `asksign` 関数を用いて確認します.

```
(%i38) log((x+1)^2);
1. Trace: (SIMPLN '((%LOG) ((MEXPT SIMP) ((MPLUS SIMP) 1 $X) 2)) '1 'NIL)
2. Trace: (SIMPLN '((%LOG) ((MPLUS SIMP) 1 $X)) '1 'T)
2. Trace: SIMPLN ==> ((%LOG SIMP) ((MPLUS SIMP) 1 $X))
1. Trace: SIMPLN ==> ((MTIMES SIMP) 2 ((%LOG SIMP) ((MPLUS SIMP) 1 $X)))
(%o38)
2 log(x + 1)
(%i39) plog((x+1)^2);
1. Trace: (SIMPPLOG '((%PLOG) ((MEXPT SIMP) ((MPLUS SIMP) 1 $X) 2)) '1 'NIL)
Is x + 1 zero or nonzero?
```

```
pos;
2. Trace: (SIMPLN '((%LOG) ((MEXPT SIMP) ((MPLUS SIMP) 1 $X) 2)) '1 'T)
3. Trace: (SIMPLN '((%LOG) ((MPLUS SIMP) 1 $X)) '1 'T)
3. Trace: SIMPLN ==> ((%LOG SIMP) ((MPLUS SIMP) 1 $X))
2. Trace: SIMPLN ==> ((MTIMES SIMP) 2 ((%LOG SIMP) ((MPLUS SIMP) 1 $X)))
1. Trace: SIMPPLOG ==> ((MTIMES SIMP) 2 ((%LOG SIMP) ((MPLUS SIMP) 1 $X)))
(%o39)
2 log(x + 1)
```

この例では、内部関数 `simpln` と `simpplog` に LISP の `trace` 関数を用いて処理の違いを見たものです。`plog` 関数では与式に純虚数の積が含まれている場合、単純に $i\pi/2$ を `add2*` 関数を用いて加え、与式のノルムに対して `log` 関数を用いる関数と言えます。

```
(%i17) plog(%i*x^2);
1. Trace: (SIMPPLOG '((%PLOG) ((MTIMES SIMP) $I ((MEXPT SIMP) $X 2))) '1 'NIL)
Is x zero or nonzero?

pos;
2. Trace: (ADD2* '((MTIMES SIMP) 2 ((%LOG SIMP) $X)) '((MTIMES) 1 ((RAT) 1 2) $I $PI))
2. Trace: ADD2* ==>
((MPLUS SIMP) ((MTIMES SIMP) ((RAT SIMP) 1 2) $I $PI)
((MTIMES SIMP) 2 ((%LOG SIMP) $X)))
1. Trace: SIMPPLOG ==>
((MPLUS SIMP) ((MTIMES SIMP) ((RAT SIMP) 1 2) $I $PI)
((MTIMES SIMP) 2 ((%LOG SIMP) $X)))
```

```

                                %i %pi
(%o17)          2 log(x) + -----
                                2

(%i18) plog(-%i*x^2);
1. Trace: (SIMPLOG '((%PLOG) ((MTIMES SIMP) -1 $%I ((MEXPT SIMP) $X 2))) '1 'NIL)
Is x zero or nonzero?

pos;
2. Trace: (ADD2* '((MTIMES SIMP) 2 ((%LOG SIMP) $X)) '((MTIMES) -1 ((RAT) 1 2) $%I $%PI))
2. Trace: ADD2* ==>
((MPLUS SIMP) ((MTIMES SIMP) ((RAT SIMP) -1 2) $%I $%PI)
 ((MTIMES SIMP) 2 ((%LOG SIMP) $X)))
1. Trace: SIMPLOG ==>
((MPLUS SIMP) ((MTIMES SIMP) ((RAT SIMP) -1 2) $%I $%PI)
 ((MTIMES SIMP) 2 ((%LOG SIMP) $X)))
                                %i %pi
(%o18)          2 log(x) - -----
                                2

```

この例で示す様に, `plog` 関数では $x+1$ の正值性を確認していますが, ここで `neg;` を入力しても結果は同じ $2\log(x+1)$ になります. 但し, `zero;` を入力すると, `plog(0)` is undefined と返されます. 従って, 0 かどうかをチェックする機能と考えた方が良いでしょう.

`exp` 関数, `log` 関数と `plog` 関数は次の大域変数の設定によって, Maxima 上で自動的に簡易化が実行されます.

8.3.2 対数関数に関連する関数

—— 逆三角関数や逆双曲関数を対数関数に変換する関数 ——

`logarc(< 式 >)`

関数 `logarc` は同名の大域変数 `logarc` の設定とは無関係に, 逆三角関数 (`acos`, `asin`, `atan`, `atan2`, `asec`, `acsc`, `acot`) と逆三角関数 (`asinh`, `acosh`, `atanh`, `asech`, `acsch`, `acoth`) を対数関数による有理式に変換し, `ev` 関数を用いた式の再評価を行う関数です.

```
(%i14) logarc(atan(x));
              %i (log(%i x + 1) - log(1 - %i x))
(%o14)      - ----
              2

(%i15) logarc:true;
(%o15)      true
(%i16) atan(x);
              %i (log(%i x + 1) - log(1 - %i x))
(%o16)      - ----
              2
```

—— 対数関数を潰す関数 ——

logcontract(〈式〉)

logcontract 関数は log を含む式を簡易化で潰す関数です。即ち, log 関数の係数を引数に取込み, その結果によって, log から sqrt 等の関数に変換します。具体的には, 〈式〉を再帰的に調べて, $a_1 \log(b_1) + a_2 \log(b_2) + c$ の形の部分式を $\log(\text{ratsimp}(b_1^{a_1} * b_2^{a_2})) + c$ に変換します。

```
(%i8) 2*log(x)+4*log(y)+8;
(%o8)      4 log(y) + 2 log(x) + 8
(%i9) logcontract(%);
              2  4
(%o9)      log(x y ) + 8
(%i10) declare(n,integer);
(%o10)      done
(%i11) logcontract(2*log(x)+4*n*log(y)+8);
              2  4 n
(%o11)      log(x y ) + 8

(%i12) logcontract(2*log(x)+4*m*log(y)+8);
              4      2
(%o12)      m log(y ) + log(x ) + 8
```

この logcontract 関数は log 関数の整数係数に対して影響を与える関数です。例えば, `declare(n,integer);` を実行して, `logcontract(2*log(x)+4*n*log(y)+8);`

を実行すれば、第二式の $4+n+\log(y)$ の係数 $4*n$ は `featurep(coeff,integer)` を満す為、 $\log(x^2y^{4n})+8$ に簡易化されています。ところが、`logcontract(2*log(x)+4*m*log(y)+8);` の場合、変数 `m` は integer として未宣言の為、簡易化が途中で停止しています。

猶、この `logcontract` 関数は大域変数 `superlogcon` の影響を受けます。この大域変数 `superlogcon` を初期値の `true` から `false` に変更する事で、内部関数 `lgcsqrt` の処理を外す事になります。その為、与式の主演算子が和の場合と積の場合で処理する以上の事が出来なくなります。

```
(%i54) superlogcon:true$
(%i55) neko: 2*(a*log(x) + 2*a*log(y))$
(%i56) :lisp (trace lgcsort)
WARNING: TRACE: redefining function LGCSORT in top-level, was defined in
        /usr/local/maxima-5.14.0/src/binary-clisp/comm2.fas
;; Tracing function LGCSORT.
(LGCSORT)
(%i56) logcontract(neko);
1. Trace:
(LGCSORT
  '((MTIMES SIMP) 2
    ((MPLUS SIMP) ((MTIMES SIMP) $A ((%LOG SIMP) $X))
      ((MTIMES SIMP) 2 $A ((%LOG SIMP) $Y))))))
1. Trace: LGCSORT ==>
((MTIMES SIMP) $A
  ((MPLUS SIMP) ((MTIMES SIMP RATSIMP) 2 ((%LOG SIMP) $X))
    ((MTIMES SIMP RATSIMP) 4 ((%LOG SIMP) $Y))))
                                2 4
(%o56)                a log(x y )
(%i57) superlogcon:false$
(%i58) logcontract(neko);
                                2
(%o58)                2 (a log(y ) + a log(x))
(%i59) logcontract(expand(neko));
                                4      2
(%o59)                a log(y ) + a log(x )
```

但し、この例の様に、式を展開して与えれば、`log` 関数毎に式を纏める作用があり、この

様な出力が必要であれば, 大域変数 `superlogcon` を `false` にします.

極座標形式に変換する関数

`polarform(<式>)`

`polarform` 関数は与えられた `<式>` を $r * e^{(i * \theta)}$ の形に変換します. 猶, 多項式が実数係数多項式の場合は入力そのまま返され, 関数を含む場合には, その関数が負であれば $e^{(i * \pi)}$ をかけた式が返されます.

```
(%i31) polarform((1+%i)^3);
```

$$e^{3 i \pi}$$

4

```
(%o31) 2 sqrt(2) %e
```

```
(%i32) polarform(x^2+1);
```

ppppp

2

```
(%o32) x^2 + 1
```

```
(%i33) polarform((x+1)^2);
```

Is $x + 1$ zero or nonzero?

pos;

2

```
(%o33) x^2 + 2 x + 1
```

```
(%i34) polarform(sin(x+1));
```

Is $\sin(x + 1)$ positive or negative?

neg;

$$e^{i \pi}$$

```
(%o34) - %e sin(x + 1)
```

対数関数に関連する大域変数

変数名	初期値	概要
%e_to_numlog	false	指数に対数を持つ冪乗の簡易化
logabs	false	log を含む不定積分の結果を制御
logarc	false	逆三角関数や逆双曲関数を対数関数で表現
logconcoeffp	false	logcontract で潰される係数を制御
logexpand	true	対数関数の積や冪の自動変換
lognegint	false	対数関数の引数が負の場合の処理を制御
lognumer	false	対数関数の浮動小数点引数の制御
logsimp	true	log を含む指数関数の冪乗の自動化を制御
superlogcon	true	logcontract 関数による簡易化を制御

先ず,%e_to_numlog が true とした時, 与式 $e^{r \log x}$ の r が numberp 関数や内部関数の maxima-integerp 関数に対して true を返す場合, 与式は x^r に簡易化されます. 猶,radcan 関数もこの変換を行います.

```
(%i1) %e^(a1*log(b1));
                                a1 log(b1)
(%o1)                                %e
(%i2) %e_to_numlog:true;
(%o2)                                true
(%i3) declare(a1,integer);
(%o3)                                done
(%i4) %e^(a1*log(b1));
                                a1
(%o4)                                b1
```

この例では, 変数 a1 に属性として integer を指定した為, numberp 関数は false になっても maxima-integerp 関数で true になる為, 自動変換が実行されています.

大域変数 logabs が true の場合, integrate(1/x,x) の様に計算結果に log 関数が結果に含まれる場合, log 関数を log(abs(...)) で置換されます. 但し, logabs が false であれば log(...) の項を持つものになります. 猶, 定積分では logabs:true として対数関数が処理されます. これは, 不定積分の両端点での評価が必要となる事が多い為です.

大域変数 logarc が true であれば, 自動的に逆三角関数, 逆双曲関数を対数関数の書式に変換します.

大域変数 `logconcoeffp` は `logcontract` 関数による式の変形で、式に含まれた `log` 関数を含む項の係数に対し、この大域変数で指定した真理関数が真となる場合に、その係数を `log` 関数の内部に取り込む操作を行います。ここで、大域変数 `logconcoeffp` に指定する真理関数は名詞型、即ち、`logconcoeffp: 'numberp;` の様に真理関数名を名詞型にして割当てます。猶、ここで指定する真理関数は一つの引数のみを持つ関数でなければなりません。

```
(%i9) logconcoeffp: 'numberp;
(%o9)                                     numberp
(%i10) logcontract(1/2*log(x));
(%o10)                                     log(sqrt(x))
(%i11) logcontract(5/2*log(x));
(%o11)                                     5/2
                                             log(x)
```

この例では、最初の `logcontract(1/2*log(x))` で、`logcontract` 関数は大域変数 `logconcoeffp` に設定された真理関数 `numberp` に `log` 関数が含まれている項の係数 $\frac{1}{2}$ を引渡します。ここで、真理関数の返却値が真理関数の結果が `true` であれば、この係数を `log` 関数の中に取込みますが、 $1/2$ に対し、`numberp` 関数は `true` を返す為に、`log(sqrt(x))` を返します。猶、真理関数の返却値が `false` の場合、与式をそのままを返します。

大域変数 `logexpand` が `true` の場合、自動的に $\log(a^b)$ を $b \log(a)$ に変換します。ここで、`all` の場合、自動的に $\log(ab)$ は $\log(a) + \log(b)$ に変換されます。`super` の場合、 $a = 1$ でない有理数 a/b に対し、 $\log(a/b)$ は $\log(a) - \log(b)$ に変換されます。猶、整数 b に対し、 $\log(1/b)$ は `logexpand` とは無関係に、常に簡易化されます。`false` の場合はこれらの簡易化は全て実行されません。

大域変数 `lognegint` が `true` の場合、正整数 n に対し、 $\log(-n)$ を $\log(n) + i\pi$ で置換える規則が内部的に設定されます。

大域変数 `lognumber` が `true` の場合、`log` の負の浮動小数引数は `log` に渡される前に、常にその絶対値に変換されます。

大域変数 `logsimp` が `false` の場合、`log` を含む `%e` の冪乗の自動簡易化は実行されません。大域変数 `superlogcon` が `false` の場合、`logcontract` 関数内部で、内部関数 `lgcsort` が省略され、式中の `log` 関数を纏めるのではなく、`log` 関数項毎の簡易化になります。

8.4 代数方程式

8.4.1 Maxima の方程式

Maxima の方程式は演算子= の両側に演算子=を持たない式を配置した式です. 即ち, $x^2+2x+1=0$ の様な形式になります. 但し, 右辺が 0 の場合は, x^2+2x+1 の様に, 演算子=と 0 を省略しても構いません.

ここで演算子=は infix 型の演算子の為, この演算子の左右の式は lhs 関数と rhs 関数を使って取り出す事が出来ます.

```
(%i17) eq1:x^2+2*x+1=y^2;
```

```
(%o17)          2          2
              x  + 2 x + 1 = y
```

```
(%i18) lhs(eq1);
```

```
(%o18)          2
              x  + 2 x + 1
```

```
(%i19) rhs(eq1);
```

```
(%o19)          2
              y
```

この例では方程式として $x^2 + 2 * x + 1 = y^2$ を eq1 に割当てており, `lhs(eq1)` で方程式の左側の x^2+2x+1 , `rhs(eq1)` で方程式右側の y^2 を各々取出しています. この lhs 関数と rhs 関数は infix 型の内挿演算子に対して利用可能な関数です.

Maxima で扱える方程式としては, 1 変数多項式で構成された方程式, 多変数多項式で構成された連立方程式, cos や log 等の初等関数を含むより一般的な方程式があります. 他に, 微分 'diff' を含む方程式や積分 'integrate' を含む方程式もあります. 猶, 微分を含む方程式は, 常微分方程式の節で詳細を述べる為, ここでは, 微分と積分を含まない代数方程式を中心に述べます.

更に, Maxima では連立方程式を扱う事も可能です. この場合, $[eq_1, \dots, eq_n]$ の様に, 方程式を成分とするリストで連立方程式を表現します.

```
(%i25) eq2: [2*x^2-5*y=1,x+y*x+y^2=4];
                2          2
(%o25)          [2 x  - 5 y = 1, y  + x y + x = 4]
(%i26) eq2[1];eq2[2];
                2
(%o26)          2 x  - 5 y = 1
(%i27)
                2
(%o27)          y  + x y + x = 4
```

この例では、二つの方程式 $2x^2-5y=1$ と $x+yx+y^2=4$ で構成される方程式のリストを eq2 に割当てています。連立方程式をリストで表現する為、一つの方程式を取り出す場合は、リストの成分の取り出しと同じ方式で行えます。

Maxima には与えられた方程式の近似解を数值的に解く関数に allroots 関数と realroots 関数があります。又、厳密解を求める関数として, linsolve 関数と solve 関数、厳密解が計算出来る場合には厳密解を計算し、厳密解の計算に失敗した場合に近似解を計算する algsys 関数があります。

これらの関数は、与えられた方程式が1変数の多項式で構成される場合、線形連立方程式の場合、多変数多項式で構成される方程式系の場合、そして、より一般的な初等関数を含む方程式の場合に区分出来ます。

先ず、方程式系が一つの1変数多項式で構成される場合、近似解を計算する allroots 関数と realroots 関数が使えます。線形連立方程式の場合は, linsolve 関数を使って厳密解を計算出来ます。そして、多変数多項式で構成された方程式系に対しては、algsys 関数を用いて可能であれば厳密解、近似解が計算可能な場合は、近似解を計算出来ます。最後に、より一般的な方程式に対しては solve 関数を用いて厳密解の計算が行えます。

Maxima で計算した結果を自動的に変数に代入したり、重複リストを生成させる為には、以下の大域変数を調整する必要があります。

— 方程式に関連する大域変数 —

変数名	初期値	概要
backsubst	true	三角関数化した方程式の代入を抑制
globalsolve	false	解の値の自動代入を制御
multiplicities	not_set_yet	重複度リスト
programmode	true	allroots,linsolve,solve 等の出力を制御

大域変数 backsubst は三角関数化した方程式に対して代入の制御を行います。backsubst

が `false` の場合, 方程式を三角関数化した後で, 代入を防ぎます. これは, 後代入でとてつもなく大きな式が生成される様な問題で必要となります.

大域変数 `globalsolve` は Maxima で方程式を解いた時に, 方程式の変数に求めた解を自動代入するかどうかを制御する変数です. 大域変数 `globalsolve` が `true` の場合, 解かれた変数に解が実際に割当てられます.

```
(c101) globalsolve:true;
(d101)                                     true
(c102) solve([xx*2+yy*3-1=0,xx+yy=10],[xx,yy]);
(d102)                                     [[xx : 29, yy : - 19]]
(c103) xx;
(d103)                                     29
(c104) yy;
(d104)                                     - 19
(c105) globalsolve:false;
(d105)                                     false
(c106) solve([mm*2+nn*3-1=0,mm+nn=10],[mm,nn]);
(d106)                                     [[mm = 29, nn = - 19]]
(c107) mm;nn;
(d107)                                     mm
(c107)
(d107)                                     nn
```

`globalsolve:true` とした状態で, ある方程式を解いた後に同じ変数の方程式を解こうとすると次のエラーが出るので注意します. 例えば, 上記の例の (c106) 行の方程式以下の行で置き換えた場合には次の様になります.

```
(c106) solve([xx*2+yy*3-1=0,xx+yy=10],[xx,yy]);
a number was found where a variable was expected -solve
-- an error. quitting. to debug this try debugmode(true);)
(c107)
```

猶, 重複度が 2 以上の変数が存在する場合, 自動代入は実行されない事に注意して下さい.

大域変数 `multiplicities` は, `solve` や `realroots` で返される個々の解に対応する重複度のリストが設定されます.

```

(%i2) multiplicities;
(%o2) not_set_yet
(%i3) solve(x^2-4*x+4,x);
(%o3) [x = 2]
(%i4) multiplicities;
(%o4) [2]
(%i5) realroots(x^4+2*x^3-3*x^2-4*x+4);
(%o5) [x = - 2, x = 1]
(%i6) multiplicities;
(%o6) [2, 2]
(%i7) solve(x^5+x^4-2*x^3-2*x^2+x+1,x);
(%o7) [x = 1, x = - 1]
(%i8) multiplicities;
(%o8) [2, 3]
(%i9) factor(x^5+x^4-2*x^3-2*x^2+x+1);
(%o9) (x - 1)^2 (x + 1)^3

```

この例では、最初に方程式 $x^2 - 4x + 4 = 0$ を solve 関数を用いて解き、次に realroots 関数を使って、方程式 $x^4 + 2x^3 - 3x^2 - 4x + 4 = 0$ を解いています。そして最後には、 $x^5 + x^4 - 2x^3 - 2x^2 + x + 1 = 0$ を解いています。最初の例では、重複度が2の為、multiplicities にはリスト [2] が割当てられています。次の例では、重複度が各々2である事が判ります。最後の例では、 $x = 1$ の重複度が2、 $x = -1$ の重複度が3である事が判ります。実際、与式を factor 関数で因子分解すれば確認出来ます。

大域変数 programmode は、返却値に中間行ラベルを付けて出力するかどうかを制御します。programmode が false の場合、solve, realroots, allroots と linsolve 関数は %t ラベル (中間行ラベル) に答をラベル付けして出力します。

true の場合、これらの関数はリストの要素として答えを返します (programmode:false も使われます。但し、backsubst が false に設定された時を除きます)。

```

(%i4) programmode:false;
(%o4) false
(%i5) solve(x^2+1,x);
Solution:
(%t5) x = - %i

```



```
(%t6)                                x = %i
(%o6)                                [%t5, %t6]
(%i6) programmode:true;
(%o6)                                true
(%i7) solve(x^2+1,x);
(%o7)                                [x = - %i, x = %i]
```

8.4.2 1 変数多項式方程式の場合

最初に, 方程式が多項式で構成された場合について述べます. 方程式系が一つの 1 変数多項式のみで構成されている場合, その近似解を `allroots` 関数と `realroots` 関数を用いて計算出来ます.

— 数値解を求める関数 —

```
allroots(< 方程式 >)
realroots(< 多項式 >, < 許容範囲 >)
realroots(< 多項式 >)
```

最初の `allroots` 関数は単変数の実数係数多項式の実数解と複素解全てを計算します. `allroots` 関数は, 多項式が実係数で, 大域変数 `polyfactor` が `true` の場合に実数上で因子分解を行います, 係数に `%i` が含まれていれば複素数上で因子分解を行います.

```
(%i14) allroots(%i*x^2+1=0);
(%o14) [x = .7071067811865475 %i + .7071067811865475,
        x = - .7071067811865475 %i - .7071067811865475]
(%i15) polyfactor:true;
(%o15)                                true
(%i16) allroots(x^2+1=0);
(%o16)                                2
                                x  + 1.0
(%i17) allroots(%i*x^2+1=0);
(%o17) %i (x - .7071067811865475 %i - .7071067811865475)
        (x + .7071067811865475 %i + .7071067811865475)
```

`allroots` は重複解を持つ時に不正確な結果を返す事があります. この場合は与式に `%i` をかけたものを計算すれば解決するかもしれません.

allroots は多項式方程式以外には使えません.rat 命令を実行した後に, 方程式の分子が多項式で, 分母が高々複素数でなければなりません. polyfactor が true であれば, allroots の結果として常に同値な式 (但し, 因子分解されたもの) が返されます.

realroots 関数は与えられた実単変数多項式 (多項式) の全ての実根を (許容範囲) で指定する許容範囲内で求めます. 猶, (許容範囲) が 1 よりも小さければ, 全ての整数根を厳密に求めます. (許容範囲) は必要であれば, 任意の小さな数を設定しても構いません. (許容範囲) を省略した場合は, 大域変数 rootsepsilon の値が使われます.

```
(%i34) realroots(x^2-2=0,1.0e-5);
                                370727      370727
(%o34)      [x = - ----, x = ----]
                                262144      262144
(%i35) float(sqrt(2)-rhs(%o34[2]));
(%o35)      2.289179735770474E-6
```

この例では方程式 $x^2 - 2 = 0$ の解を精度 10^{-5} 以内で求めています. 解は浮動小数点ではなく, 有理数で返されます.

realroots は解の大域変数 multiplicities に重複度の情報をリストの形式で追加します. 大域変数 multiplicities に解の重複度リストを設定する関数は, 他に solve 関数があります. ここで, 重複度リストは, 求めた解に対応する形で, 整数のリストとして表現されています.

allroots 関数に影響を与える大域変数

変数名	初期値	概要
polyfactor	false	因子分解の有無
rootsepsilon	1.0E-7	根を含む区間

polyfactor は allroots で利用される大域変数です.polyfactor が true であれば, polyfactor に与えられた多項式が実係数多項式なら実数上で因子分解し, 係数に純虚数%iが含まれていれば複素数上で因子分解を行った結果を返します.

大域変数 rootsepsilon は realroots 関数によって見つけられた根を含む確実な区間を設定する際に使う実数です.

8.4.3 区間内の根の個数

Maxima では指定した半开区間に存在する根の個数を計算する関数 nroots 関数があります. この nroots 関数は 1 変数多項式に対して利用可能です.

———— 区間内の根の個数を返す関数 ————

`nroots(<多項式>, <下限>, <上限>)`

`nroots` 関数は `<上限>` と `<下限>` で指定された半開区間 `(<下限>, <上限>]` 内部に、幾つの 1 変数多項式 `<多項式>` の根があるかを返します。

ここで、区間の終点は負の無限大と正の無限大に各々対応する `minf, inf` でも構いません。このアルゴリズムには Sturm 級数による手法が適用されています。

```
(%i18) nroots(x^2+2,-2,2);
(%o18)                                0
(%i19) nroots(x^2-2,-2,2);
(%o19)                                2
(%i20) nroots(x^2-5,-2,2);
(%o20)                                0
(%i21) nroots(x^2-1,-2,2);
(%o21)                                2
```

8.4.4 多項式方程式の場合

`allroots` 関数と `realroots` 関数は、1 変数多項式の近似解を計算する関数です。これに対し、方程式の厳密解を計算出来る関数として、`linsolve` 関数、`algsys` 関数と `solve` 関数があります。

ここで、多変数の線形多項式で構成された方程式系に対しては `linsolve` 関数が使えます。

———— 線形連立方程式を解く関数 ————

`linsolve([<方程式1>, <方程式2>, …], [<変数1>, <変数2>, …])`

`linsolve` 関数は与えられた線形連立方程式を変数リストに対して解きます。各方程式は各々与えられた変数リストの多項式でなければなりません。

```
(%i68) linsolve([x+y-2=0,y-x+1=0],[x,y]);
              3      1
(%o68)      [x = -, y = -]
              2      2
```

— linsolve に影響を与える大域変数 —

変数名	初期値	概要
<code>linsolve_params</code>	<code>true</code>	解に助変数を導入
<code>linsolvewarn</code>	<code>true</code>	<code>linsolve</code> の警告を抑制

`linsolve_params` が `true` であれば, `linsolve` はまた記号 `%ri` を生成し, `algsys` に記載された任意の助変数を表現する為に用いられます.

`false` であれば, 以前の様に `linsolve` が動作します. 即ち, 不定方程式型に対し, 他の項の幾つかの引数に対して解きます.

`linsolvewarn` が `false` であれば, `dependent equations eliminated`(従属方程式が消去された) というメッセージ出力が抑制されます.

多変数多項式で構成された連立方程式は, `solve` 関数や `algsys` 関数で解く事が出来ます. 但し, `solve` 関数がより一般的な方程式の厳密解の計算が可能ですが, `algsys` 関数は, 多変数多項式で構成された方程式系の厳密解の計算に失敗すると, 今度は近似解を計算する点で, 多項式方程式の計算では使い易いでしょう.

— algsys 関数 —

```
algsys([< 方程式1> , < 方程式2> , ...], [< 変数1> , < 変数2> , ...])
```

`algsys` 関数では, 方程式と変数はリストで与えます. 返却される解に `%r1` や `%r2` といった記号が含まれる事がありますが, これらは助変数を表示する為に用いられるもので, 助変数は大域変数 `%rnum_list` に蓄えられています.

```
(%i1) algsys([2*x+3*y=1], [x,y]);
```

```
(%o1)                2 %r1 - 1
      [[x = %r1, y = - ----]]
                        3
```

```
(%i2) %rnum_list;
```

```
(%o2)                [%r1]
```

この例の様に, (連立) 方程式の階数が足りない場合には, 助変数を補って与えられた方程式を解きます. 猶, `%rnum_list` には `algsys` で使われた助変数が逐次追加されて行きます.

`algsys` は以下の手順で方程式を解き, 必要であれば再帰的に処理を行います.

1. 最初に方程式を `factor` で因子分解し, 各因子から構成される部分系 `systemi`, 即ち, 方程式の集合を構築します.

2. 部分系 system_i から, 方程式 eq_n を取出し, それから変数 var を選択します. この変数の選択は方程式 eq_n に含まれる変数の中から最小次数のものを選出します. それから方程式 eq_n と system_i の部分系 $\text{system}_i \setminus \{\text{eq}_n\}$ に含まれる方程式 eq_j を変数 var を主変数とする多項式と看做して終結式を計算します. この操作によって, 新しい部分系 system_{i+1} は system_i よりも少ない変数で生成されます. それから 1 の処理に戻ります.
3. 一つの方程式で構成される部分系が最終的に得られると, その方程式が多変数で, 係数が浮動小数点数でなければ, 厳密解を求める為に `solve` 関数を呼出します. 更に, 方程式が単変数で線型, 二次, 或いは四次の多項式であれば, `solve` を再び呼出します.

係数が浮動小数点数で近似されている場合, 方程式が単変数で線型, 二次又は四次の何れでもなく, 大域変数 `realonly` が `true` の場合, 実数値解を見付ける為に函数 `realroots` を呼出します. `realonly` が `false` の場合, 解を求める為に函数 `allroots` を呼出します.

猶, `algsys` が要求以下の精度解を生成した場合, 大域変数 `algepsilon` の値をより小さな値に変更しても構いません. 大域変数 `algexact` が `true` であれば, `solve` 関数を呼出します.

4. 3 の段階で得られた解を以前の段階に代入して, 解の計算過程の 1 に戻ります. 猶, 浮動小数を近似した多変数方程式に対しては, 次のメッセージを表示します:

"algsys cannot solve - system too complicated." (意味: "algsys では解けません - 系があまりにも複雑です.")

`radcan` 函数を使えば, 大きくて複雑な式が出来ます. この場合, `pickapart` や `reveal` を解の計算に用います.

ここで終結式は二つの一変数多項式に対して定義されるもので, 例えば, 多項式 f と g の根を α_i, β_j とすると, $\text{res}(f, g, x) = a_m^n b_n^m \prod_{0 \leq i \leq m, 0 \leq j \leq n} (\alpha_i - \beta_j)$ となる事が知られています.

この事は f と g に共通の零点が存在する場合には終結式が零になる事を意味します. 従って, f と g が多変数の場合, f と g を f と g の共通の変数 x_1 の多項式と看做してその終結式を計算すると, f と g の共通の根が存在する場合, 終結式は零でなければなりません. こうする事で, 変数 x を含まない新しい多項式 $\text{res}(f, g, x)$ が得られま

す. この操作を方程式系に対して行う事で, 1 変数の多項式が得られると, その多項式を解いて, 一段前の方程式系に代入し, 解を求めて行く方式となっています.

話を簡単にする為に, 一次の連立方程式で簡単に説明しましょう.

最初に次の線形方程式が与えられたとします.

$$f: ax + by + p = 0 \quad g: cx + dy + q = 0$$

この連立方程式を構成する多項式 f と g の終結式は次の行列式を計算すると得られます.

$$\text{res}(f, g, x) = \det \begin{pmatrix} a & by + p \\ c & dy + q \end{pmatrix}$$

この行列式は $(ad - bc)y - ap + cp$ です. ここで, f と g の終結式は f と g が共通の解を持つ場合に 0 となります. このことから, 方程式 $(ad - bc)y - ap + cp = 0$ が得られます. この終結式では, 前の方程式系から変数 x と方程式が減り, 変数 y の方程式となります. そこで, 終結式から得られた方程式を解けば $y = \frac{ap - cp}{ad - bc}$ が得られます. それから, 一段前の方程式系に戻って変数 y に値を代入すれば, 変数 x の方程式が得られ, その方程式を解けば最終的に $x = \frac{bq - dp}{ad - bc}$ が得られます.

algsys 関数は, 方程式系を構成する各多項式を因子分解し, 次数を落した各因子で構成される方程式の集合に対して終結式を計算し, 新しい方程式系から変数を一つ消去します. この処理を繰り返す事で, 最終的に一変数の多項式方程式が得られると, そこから一つの変数の解を定められます. この計算で, 4 次以下の方程式であれば, 公式を用いた根の計算が可能ですが, それ以外の場合で厳密解が計算出来なければ, allroots を用いて近似数値解を求め, それから, 処理を逆に遡る事で, 全ての解を求める事が出来ます. algsys はこのようなアルゴリズムを採用しているのです.

— algsys 関数に影響を与える大域変数 —

変数名	初期値	概要
%rnum_list	[]	algsys 関数で解に導入された変数リスト
algexact	false	algsys 関数が solve 関数を呼出すかどうかを制御
algepsilon	10^8	algsys で用いられる変数
algedelta	10^{-5}	algsys で用いられる変数
realonly	false	true の場合, algsys 関数は実数解のみを返却

%rnum_list は algsys 関数で方程式の解を求めた時に導入された変数 %r が生成順で追加されるリストです. これは後に解に代入する時に便利です.

大域変数 `algexact` は `algsys` 関数に影響を与える大域変数の一つです。true であれば, `algsys` は `solve` を呼出し, `realroots` を常に利用します。false であれば, 終結式が単変数でない場合と quadratic か biquadratic な場合のみ `solve` の呼出しを行います。`algexact:true` とすると, 厳密解のみを保証するものではなく, `algsys` が最初に厳密解を計算しようと試み, 結局, all が失敗した時に近似解のみを生成します。

大域変数 `algepsilon` は `algsys` 関数で利用される定数で, 解の精度を制御する大域変数です。

大域変数 `algdelta` も `algsys` 関数で利用される定数ですが, こちらは計算した近似解を方程式に代入して零点からのズレを見る際に, 代入した式に複素数が現われた時に誤差の判定で用いられる大域変数です。初期値は $1.0e-5$ が設定されています。

大域変数 `realonly` が true の時, `algsys` は実数解, 即ち `%i` を持たない解のみを返します。

8.4.5 一般的な方程式

一般的な方程式に対しては `solve` 関数や `find_root` 関数が利用出来ます。`solve` 関数は, 与えられた方程式系の厳密解を返す関数です。それに対し, `find_root` 関数は与えられた区間内で近似解を求める関数です。

まず, `solve` 関数で扱える方程式としては, `sin` 等の三角関数, 指数関数や対数関数を含んだ方程式が扱えます。但し, `algsys` 関数と違い, 厳密解の計算に失敗した場合に数値近似解を計算する様な事は行いません。

— solve 関数 —

```
solve(<式>)
solve(<式>, <変数>)
solve([<方程式>, ..., <方程式n>])
solve([<方程式>, ..., <方程式n>], [<変数1>, ..., <方程式n>])
```

`solve` 関数は代数方程式 `<式>` を `<変数>` に対して解き, 解のリストを返します。

`<式>` が方程式でなければ, `<式>` が零に等しいと設定されていると仮定します。即ち, 式 $x^2+2*x+1$ が `<式>` であれば, `solve` は方程式 $x^2+2*x+1=0$ が与えられたと `solve` 関数は解釈します。

`<変数>` は和や積を除く関数の様な原子でない式でも構いません。猶, `<式>` が関数 $f(x)$ の多項式であれば, 最初に $f(x)$ に対して解き, その結果が `c` であれば, 方程式 $f(x) = c$ を解く事で対処出来ます。

具体的には以下の処理を行います。

```
(%i26) solve(log(x)^2-2*log(x)+1,log(x));
(%o26) [log(x) = 1]
(%i27) solve(%o25[1],x);
(%o27) [x = %e]
```

〈式〉が1変数のみの場合は〈変数〉を省略出来ます。更に、〈式〉は有理式でも良く、その上、三角関数、指数関数等を含んでも構いません。

solveは与えられた方程式が単変数の場合は次の手順で解の計算を行います。

- 方程式が変数 var の線形結合であれば、 var に対して自明に解けます。
- 方程式が $a \cdot var^n + b$ の形式ならば、解は $(-b/a)^{1/n}$ に1の n 乗根を掛けたもので得られます。
- 方程式が変数 var の線形結合ではなく、方程式に含まれる変数 var の各次数の $\text{gcd}(n \text{ とします})$ が次数を割切る場合、大域変数 `multiplicities` に n が追加されます。そして、solveは var^n で方程式を割った結果に対して再び呼出されます。
- 方程式が因子分解されている場合、各因子に対して solve が呼出されます。
- 方程式二次、三次、又は四次の多項式方程式の場合、解の公式を必要があれば用います。

`solve([〈方程式1〉, ..., 〈方程式n〉], [〈変数1〉, ..., 〈変数n〉])` の場合、多項式の方程式系を `linsolve` 関数、或いは `algsys` 関数を用いて解き、その変数で解のリストを返します。ここで、`linsolve` 関数を用いる場合、第一引数のリスト `[〈方程式i〉, i=1, ..., n]` は解くべき方程式を表現し、第二の引数リストは求めるべき未知変数のリストになりますが、方程式中の変数の総数が方程式数と等しい場合、第二の引数リストは省略しても構いません。

与えられた方程式が十分でない場合、`inconsistent` と云うメッセージを表示します。これは大域変数 `solve_inconsistent_error` で制御出来ます。単一解が存在しない場合は `singular` と表示されます。

— solve 関数の挙動に影響する大域変数 —

変数名	初期値	概要
<code>solvedecomposes</code>	<code>true</code>	<code>polydecomp</code> を用いるかどうかを制御
<code>solveexplicit</code>	<code>false</code>	陰的な解を許可するかどうかを制御
<code>solvefactors</code>	<code>true</code>	因子分解の実行の有無
<code>solvenullwarn</code>	<code>true</code>	空リストの警告の有無
<code>solveradcan</code>	<code>false</code>	<code>radcan</code> を用いるかどうかを指定
<code>solvetrigrwarn</code>	<code>true</code>	方程式を解く際に逆三角関数を利用するかを指定
<code>solve_inconsistent_error</code>	<code>true</code>	階数が不十分な連立方程式に対するエラー表示の有無
<code>breakup</code>	<code>true</code>	解の表示を制御

`solvedecomposes` が `true` であれば、多項式を解く際に、`solve` に `polydecomp` を導入します。

`solveexplicit` が `true` であれば `solve` に陰的な解、即ち、 $f(x) = 0$ の形式で返す事を禁止します。

`solvefactors` が `false` であれば、`solve` は式の因子分解を実行しません。

`solvenullwarn` が `true` であれば、空の方程式リストや空の変数リストで `solve` を呼んだ場合に警告が出ます。例えば、`solve([],[])` と入力すると警告と一緒に空リスト `[]` が返されます。

`solveradcan` が `true` であれば `solve` は `radcan` を用います。`radcan` を使うと `solve` は遅くなりますが、指数や対数関数を含む問題に対処出来ます。

`solvetrigrwarn` が `false` であれば、`solve` は方程式を解く為に逆三角関数を利用し、それによって解が失われる事を警告しません。

`solve_inconsistent_error` が `true` であれば、`solve` と `linsolve` は、 $[a+b=1, a+b=2]$ の様に階数が不十分な線形連立方程式に遭遇すればエラーを表示します。猶、`false` であれば、空リスト `[]` を返します。

`breakup` が `false` であれば、`solve` はデフォルト値の幾つかの共通部分式で構成されたものではなく、一つの式として三次又は二次の方程式の解の表示を行います。但し、`breakup` が `true` となるのは `programmode` が `false` の時だけです。

— find_root 関数 —

```
find_root(< 方程式 >, < 変数 >, < x0 >, < x1 >)
find_root(< 式 >, < x0 >, < x1 >)
```

find_root 関数は閉区間 $[x_0, x_1]$ 内部で与えられた方程式や式の根の近似解を求めます。

find_root 関数で与えられる式は、未知変数が一つの方程式や通常の式で、plot2d 関数で描画可能な式であれば十分です。

まず、引数が4個の場合、第一引数に対して内部で lhs 関数と rhs 関数が用いられている為、infix 型の内挿式演算子で、lhs 関数や rhs 関数で左右の式が取り出せる様な束縛力を持った演算子の一つ含む式が与えられます。但し、内部では lhs 関数で取出した式と rhs 関数で取出した式の差を計算しているので、演算子は=として処理が行われるのと何等の違いありません。従って、この場合には方程式を与える事になります。

これに対して引数が3個の場合は、第一引数の式に対してこの様な前処理を行いません。それから、引数が4個の場合は演算子の左右の式の差、引数が3個の場合は第一引数を plot2d 関数や plot3d 関数で用いられる内部関数 coerce-float-fun に引渡して数値データを生成し、この数値データを基に近似解の検出を二分法、与式の関数が十分に滑らかな場合には、線形近似を適用して近似解を求めます。

猶、find_root 関数で根を求める閉区間は、その両端の点の符号が一致するものである事を前提にしています。その為、开区間の両端の点の符号が異なる場合や両端の点が一致する場合にはエラーメッセージが出力されます。

— find_root 関数を制御する大域変数 —

大域変数	初期値	概要
find_root_error	true	区間が0点を包含する場合のエラー処理を制御
find_root_abs	0.0	近似解の精度に関連
find_root_rel	0.0	近似解の精度に関連

find_root 関数の近似解の精度は大域変数 find_root_abs と大域変数 find_root_rel で制御されます。

猶、find_root 関数は以前、interpolate 関数という名前の関数であった為、interpolate(< 式 >)を実行すると、find_root 関数を使う事を指示するメッセージが出力されます。

8.4.6 漸化式の場合

Maxima では, nusum パッケージを用いる事で, 漸化式を扱う事が可能です. 但し, 機能的にはまだ不十分です.

—— 漸化式を解く関数 ——

```
funcsolve(〈方程式〉, 〈g(t)〉)
```

funcsolve 関数は 〈方程式〉 を満たす有理関数 $\langle g(t) \rangle$ が存在すれば有理関数のリストを返し, 存在しない場合は空リスト `[]` を返します.

但し, 方程式は $\langle g(t) \rangle$ と $\langle g(t+1) \rangle$ の一次線形多項式でなければなりません. 即ち, funcsolve は一次線形結合の漸化式に対して利用可能です.

```
(%i28) funcsolve((n+1)*foo(n)-(n+3)*foo(n+1)/(n+1) =
(n-1)/(n+2),foo(n));
```

dependent equations eliminated: (4 3)

```
(%o28)
                                n
foo(n) = -----
              (n + 1) (n + 2)
```

8.5 極限

8.5.1 極限について

Maxima では `limit` が使えます. 一見すると `limit` は代入と似たものに見えるかもしれませんが, 実際は全く異った操作です.

例えば, $\frac{\sin(x)}{x}$ の原点での値は何になるのでしょうか? 安易な代入では $\frac{0}{0}$ となってし

まって判りませんね. 因に, $\frac{0}{0}$ や $\frac{\infty}{\infty}$ は不定形と呼ばれるものです. 字面は同じでも, 安易に割ってしまつては意味がありません.

さて, $\frac{\sin(x)}{x}$ に話を戻しましょう. この場合は $\sin(x)$ の原点周りの級数展開を考えると判り易くなります.

$\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$ となり, これを x で割ってしまうと, $\frac{\sin(x)}{x} = 1 + x \cdot (x \text{ の冪級数})$

となります. 以上から, x を 0 に近づけると 1 になる事が判ります.

Maxima で試してみましょう. Maxima には `limit` 関数があり, `limit(< 関数 >, < 変数 >, < 値 >)` で極限の計算が行えます.

```
(%i39) limit(sin(x)/x,x,0);
```

```
(%o39) 1
```

```
(%i40) plot2d(sin(x)/x,[x,-50,50]);
```

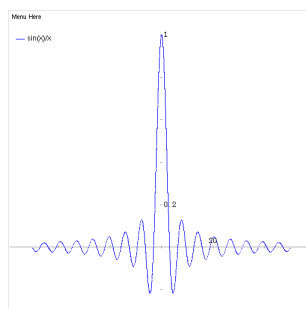
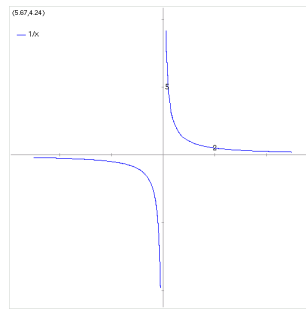


図 8.1: $\sin(x)/x$ のグラフ

極限はこの様な計算を行いますが, この近づけるという操作には方向を考えなければなりません. 例えば, $\frac{1}{x}$ はどうでしょうか. この関数は $x > 0$ なら正, $x < 0$ で負, そして, $x = 0$ が不連続点になっています.

図 8.2: $1/x$ のグラフ

```
(%i73) limit(1/x,x,0);
(%o73)                                     und
(%i74) limit(1/x,x,0,plus);
(%o74)                                     inf
(%i75) limit(1/x,x,0,minus);
(%o75)                                     minf
```

この様に右側から近付けた場合には正の無限大, 左側から近付けた場合には負の無限大となっていますね. この様に左右の極限が異なるので, $\text{limit}(1/x, x, 0)$ の結果は不定値 `und` となっています.

`limit` では正の無限大は `inf`, 負の無限大は `minf`, 複素数での無限大は `infinity`, 左右の極限が異なる場合には `und`, 未定でも有界なものには `ind` といった表記を返します.

```
(%i89) limit(1/(x^2-1),x,1,plus);
(%o89)                                     inf
(%i90) limit(1/(x^2-1),x,1,minus);
(%o90)                                     minf
(%i91) limit(1/(x^2-1),x,1);
(%o91)                                     und
(%i92) limit(sin(1/x),x,0);
(%o92)                                     ind
(%i93) limit(1/(x^2+1),x,%i);
(%o93)                                     infinity
```

—— 極限に関連する大域変数 ——

変数名	初期値	概要
lhospitallim	4	limit 関数で用いられる l'Hospital 則の適用階数の上限
limsubst	false	limit の代入を制御
tlimswitch	false	taylor 展開を利用するかどうか

lhospitallim は limit で用いられる l'Hospital 則の適用回数の最大値, これは $\text{limit}(\cot(x)/\csc(x), x, 0)$ の様な場合に無限ループに陥いる事を防ぐ為のものです.

limsubst は limit が未知の形式に代入を行う事を防ぎます. これは $\text{limit}(f(n)/f(n+1), n, \text{inf})$ の様な式が 1 となる問題点を避ける為です. limsubst が true であれば, この様な代入が許容されます.

tlimswitch が true であれば, 極限パッケージは可能な時に taylor 展開を利用します.

—— limit 関数 ——

```
limit(<式>, <変数>, <値>, <方向>)
limit(<式>, <変数>, <値>)
limit(<式>)
```

limit 関数は与えられた <式> の極限を計算します. この際, 変数が近づく方向を指定する事も可能です. この場合, <変数> が <値> に <方向> で指定したから接近する場合の <式> の極限を計算します. ここで, 方向は右極限なら plus, 左極限なら minus とします. 猶, 方向を省略した場合は両側極限が計算されます.

ここで, 原点の極限計算であれば特別に zeroa や zerob も使えます. この場合は zeroa が原点の左側 (-側), zerob が原点の右側 (+側) から近付ける事を意味します. この場合は, minus や plus の様な方向を指定する必要はありません.

計算手法は, Wang, p. の "Evaluation of definite integrals by symbolic manipulation" - ph.d. thesis - Mac tr-92 October 1971. を参照して下さい.

limit は特別な記号として inf(正の無限大) と minf(負の無限大) を用います. 出力では und(未定義), ind(不定だが有界) と infinity(複素無限大) が使われる場合があります. 猶, -inf と minf, -minf と inf は Maxima では意味が異なるので注意して下さい. この様に極限を含む式の計算で limit 関数が利用可能です即ち, 上記の -inf や -minf に加え, inf-1 様な定数式に対しては, limit 関数は式のみで評価を行います.

```
(%i51) inf - 1;
(%o51)                                     inf - 1
(%i52) limit(%);
(%o52)                                     inf
(%i53) limit(-inf);
(%o53)                                     minf
(%i54) limit(x^2+inf*x);
Is x positive, negative, or zero?

pos;
(%o54)                                     inf
```

この例で示す様に, $\text{inf}-1$ の様な式は Maxima では自動的に評価されませんが, `limit` 関数を用いて値を評価する事が出来ます. `limit(x^2+inf*x)` の様な式も同様です.

—— tlimit 関数 ——

```
tlimit(〈式〉, 〈変数〉, 〈値〉, 〈方向〉)
tlimit(〈式〉, 〈変数〉, 〈値〉)
tlimit(〈式〉)
```

`tlimswitch` を true にした `limit` 関数です. この `tlimit` 関数は 〈式〉 の Taylor 展開を行います. その展開式に対して極限計算を行います.

8.6 微分

8.6.1 微分に関する関数

Maxima の式の微分では diff 関数を用います.

diff 関数

```
diff(⟨式⟩,⟨変数1⟩,⟨階数1⟩, ...,⟨変数n⟩,⟨階数n⟩)
diff(⟨式⟩,⟨変数⟩)
diff(⟨式⟩)
del(⟨式⟩)
```

1 階微分の場合のみ, $\text{diff}(\langle \text{式} \rangle, \langle \text{変数} \rangle)$ の様に階数を省略しても構いませんが, 通常は $\langle \text{変数}_i \rangle$ と $\langle \text{階数}_i \rangle$ の一組を指定して $\langle \text{式} \rangle$ の微分を行います.

$\text{diff}(\langle \text{式} \rangle)$ は全微分を与えます. 即ち, $\langle \text{式} \rangle$ の各変数に対する微分と, 各変数の関数 del との積の和になります.

```
(%i41) diff(f(x*y));
```

```

      d                      d
(%o41)  (--- (f(x y))) del(y) + (--- (f(x y))) del(x)
      dy                      dx
```

```
(%i42) diff(g(x+y+z));
```

```

      d                      d
(%o42)  (--- (g(z + y + x))) del(z) + (--- (g(z + y + x))) del(y)
      dz                      dy
                        d
                        + (--- (g(z + y + x))) del(x)
                        dx
```

微分方程式を記述する場合, 関数の名詞型 'diff' を用います. ここで微分の名詞型の表示はデフォルトで二次元的書式になりますが, 大域変数 `display2d` を `false` にすれば入力と同等の式を一行で返します.

この大域変数 `display2d` は微分の名詞型に限らず, Maxima の計算結果の表示で数学式の表示を制御する大域変数ですが, 名詞型の微分の表示のみを制御する大域変数として大域変数 `derivabbrev` があります.

—— 微分の表示を制御する大域変数 ——

変数名	デフォルト値	概要
derivabbrev	false	微分の表示を制御

この大域変数 derivabbrev が true の場合, 名詞型の微分は添字で表示されます.

```
(%i30) 'diff(f(x),x);
                                     d
(%o30)                               -- (f(x))
                                     dx

(%i31) derivabbrev:true$
(%i32) 'diff(f(x),x);
(%o32)                               f(x)
                                     x

(%i33) display2d:false$
(%i34) 'diff(f(x),x);
(%o34) 'diff(f(x),x,1)
```

更に, 大域変数 derivsubst で名詞型の微分項の代入制御が行えます.

—— 名詞型の微分の代入を制御する大域変数 ——

変数名	デフォルト値	概要
derivsubst	false	名詞型の微分を含む項の代入を制御

大域変数 derivsubst は微分を含む項の代入の制御を行います. 例えば, $\frac{d^2y}{dt^2}$ は y の t による二階微分ですが, $\frac{dy}{dt}$ を x と置くと, $\frac{d^2y}{dt^2}$ は $\frac{dx}{dt}$ で置換えられます. 大域変数 derivsubst は名詞形の微分を含む式に対し, この様な置換を行うかどうかを制御するものです.

大域変数 derivsubst が false の場合, subst 関数による微分項の置換が出来ませんが, true の場合は置換が行えます.

```
(%i33) derivsubst;
(%o33)                               false
```

```
(%i34) subst(x,'diff(y,t),'diff(y,t,2));
                2
                d y
(%o34)          ---
                2
                dt

(%i35) derivsubst:true;
(%o35)          true
(%i36) subst(x,'diff(y,t),'diff(y,t,2));
                dx
(%o36)          --
                dt

(%i37) subst(x,'diff(y,t),2*t+t^2*'diff(y,t,2));
                2 dx
(%o37)          t  -- + 2 t
                dt
```

次に、微分の名詞型を含む式に対し、その式に含まれる 名詞型微分の最大の階数を返す関数があります。

———— derivdegree 関数 ————

derivdegree(〈式〉,〈従属変数〉,〈独立変数〉)

derivdegree 関数は 名詞型の微分を含む式で、〈独立変数〉に対する〈従属変数〉の微分で最も高い階数を見付けます。この関数は多項式の次数を返す関数 hipow と似ています。

```
(%i16) derivdegree('diff(y,x,3)*x^4+'diff(y,x,2)*'diff(y,x),y,x);
(%o16)          3
(%i17) 'diff('diff(y,x,2),x,3)+'diff(y,x,2);
                5      2
                d y    d y
(%o18)          --- + ---
                5      2
                dx     dx

(%i19) derivdegree(%,y,x);
```

```
(%o19)                                     5
```

但し, derivdegree 関数は与式の展開等を行わないので, 場合によっては間違った答を返す事もあるので注意が必要です.

```
(%i26) 'diff('diff(y,x,2),x,3)*(x^2-1)+'diff(y,x,2)
      -'diff(y,x,5)*(x-1)*(x+1);
```

$$\begin{array}{c}
 \begin{array}{ccccc}
 & & 5 & & 5 & 2 \\
 & 2 & & d\,y & & d\,y & d\,y \\
 (x^2-1) & \frac{\quad}{5} & - & (x-1)(x+1) & \frac{\quad}{5} & + & \frac{\quad}{2} \\
 & dx & & & dx & & dx
 \end{array}
 \end{array}$$

```
(%i27) derivdegree(%,y,x);
```

```
(%o27)                                     5
```

```
(%i28) expand('diff('diff(y,x,2),x,3)*(x^2-1)+'diff(y,x,2)
      -'diff(y,x,5)*(x-1)*(x+1));
```

$$\begin{array}{c}
 \begin{array}{c}
 2 \\
 d\,y \\
 \frac{\quad}{\quad} \\
 2 \\
 dx
 \end{array}
 \end{array}$$

```
(%i29) derivdegree(%,y,x);
```

```
(%o29)                                     2
```

この例では, 与式の 5 階の微分は式を展開する事で消去されるものですが, derivdegree 関数は与式を展開する事を行わずに, 安易に式に含まれる y の x による微分で階数の最も高い項を求め, その階数を返却しています.

8.6.2 vect パッケージ

vect パッケージはデフォルトで Maxima に含まれているパッケージで, grad, div, curl や laplace 等の微分演算子や, それらの式を簡易化する関数が含まれています. このパッケージを利用する為には予め `load(vect)` を実行しておく必要があります.

vect パッケージでは非可換積. を内積演算子として再定義します. その為, 非可換積が可換化されてしまうので注意が必要です. 更に, 非可換積の結合律と対応する冪への簡易化を勝手に実行しない様にする為, 関連する大域変数 dotsassoc と dotexptsimp が

false に設定されます. その一方で, スカラーに対して, 大域変数 `dotrules` を true にする事で, スカラーとの非可換積が可換に設定されます.

`vect` パッケージには以下の演算子の宣言とそれに付随する関数が収録されています.

— vect パッケージに含まれる主な関数 —

```
express(<expression>)
potential(<grad>)
scalefactors(<座標変換>)
vectorsimp(<ベクトル式>)
```

— vect パッケージに含まれる演算子 —

演算子	演算子の属性	左束縛力	右束縛力
~	内挿式演算子	134	133
grad	前置式演算子		142
div	前置式演算子		142
curl	前置式演算子		142
laplacian	前置式演算子	142	

`vect.mac` に含まれているこれらの演算子は宣言された演算子であり, 演算子の実体は, `share/vector/vector.mac` に含まれています.

これらの演算子の簡易化は `expandflags` 変数に割り当てられたリストに含まれる変数で制御されます. 但し, これらの変数を変更しても, 直ちに上記の演算子が自動的に簡易化を行うものではありません. `vect` パッケージの関数 `vectorsimp` 関数を用いて簡易化を行います. 但し, `vectorsimp` 関数自体は内部で演算子の属性を大域変数 `expandflag` のリストに登録された大域変数から設定し, `vsimp` 関数で実際の処理を実行させています.

expandflags		
大域変数名	初期値	概要
expandall	false	全ての演算子を展開
expandplus	false	被演算子に含まれる和を展開
expanddot	false	和と内積. を展開
expanddotplus	false	和と内積. を展開
expandgrad	false	grad 演算子を展開
expandgradplus	false	被演算子の和で展開
expanddiv	false	div 演算子を展開
expanddivplus	false	被演算子の和で展開
expandcurl	false	curl 演算子を展開
expandcurlplus	false	被演算子の和で展開
expandlaplacian	false	laplace 演算子を展開
expandlaplacianplus	false	被演算子の和で展開
expandprod	false	積に対して展開
expandgradprod	false	grad 演算子にて積を展開
expanddivprod	false	div 演算子にて積を展開
expandlaplacianprod	false	laplace 演算子にて積を展開
expandcurlcurl	false	curl curl を処理
expandlaplaciantodivgrad	false	laplace 演算子を div grad に展開
expandcross	false	外積を展開
expandcrosscross	false	外積を外積に対して展開
expandcrossplus	false	外積を和に対して展開
firstcrossscalar	false	外積とスカラーの処理

全てのこれらの大域変数はデフォルト値として false を持ちます. 変数名の後に plus の付く大域変数は加法性と被演算子の分配性に関連します. 同様に, 後に prod の付く大域変数は可換積や内積 (通常は非可換積) 等の積演算に対する被演算子への分配性に関連するものです.

expandcrosscross は $\mathbf{p} \cdot (\mathbf{q} \times \mathbf{r})$ を $(\mathbf{p}, \mathbf{r}) \cdot \mathbf{q} - (\mathbf{p}, \mathbf{q}) \cdot \mathbf{r}$ で置換するかどうかを決めます.

expandcurlcurl は $\text{curl curl } \mathbf{p}$ を $\text{grad div } \mathbf{p} + \text{div grad } \mathbf{p}$ で置換するかどうかを決定します.

expandcross が true の場合, expandcrossplus と expandcrosscross が true と同じ効果があります.

二つの大域変数 `expandplus` と `expandprod` は似た名前の大域変数に `true` に設定した場合と同効果です. これらが, `true` であれば, 他の大域変数, `expandlaplaciantodivgrad` は `laplace` 演算子を `div grad` で置換えます.

猶, 簡便の為にこれら全ての大域変数は, `load(vect)` で `vect` パッケージの読込を行った時点で `evflag` として宣言されます.

```
(%i1) load(vect)$
(%i2) expandall:true$
(%i3) laplacian(a*V+b*W);
(%o3)                                laplacian (b W + a V)
(%i4) vectorsimp(laplacian(a*V+b*W));
(%o4)                                laplacian (b W) + laplacian (a V)
(%i5) vectorsimp(grad(a*V+b*W));
(%o5)                                grad (b W) + grad (a V)
(%i6) vectorsimp(grad(a*b));
(%o6)                                grad (grad a . b) + grad (a . grad b)
(%i7) (V1+V2).(W1+W2);
(%o7)                                (V2 + V1) . (W2 + W1)
(%i8) vectorsimp((V1+V2).(W1+W2));
(%o8)                                V2 . W2 + V2 . W1 + V1 . W2 + V1 . W1
```

`evflag` 属性を持たない大域変数で重要な大域変数として, 大域変数 `vector_cross` があります.

大域変数 `vector_cross`

大域変数名	初期値	概要
<code>vector_cross</code>	<code>false</code>	<code>diff</code> 関数による外積の展開を制御

この大域変数 `vector_cross` は `diff` 関数による外積の微分の展開を制御する関数です. 初期値の `false` の場合, `diff` 関数による外積の展開を行いませんが, `true` の場合, `diff` 関数による外積の展開を行います.

```
(%i4) vect_cross;
(%o4)
false
(%i5) diff(f(x)~g(x),x);
d
(%o5)
-- (f(x) ~ g(x))
dx
(%i6) vect_cross:true;
(%o6)
true
(%i7) diff(f(x)~g(x),x);
d d
(%o7)
(- g(x)) ~ (-- (f(x))) + f(x) ~ (-- (g(x)))
dx dx
```

express 函数

express(< 式 >)

express 函数は名詞型の微分を展開します。express 函数は vect パッケージで定義される grad, div, curl, laplacian と外積~を認識します。但し, express 函数は微分を名詞型で返す為, 実際の微分の計算は ev 函数に 'diff オプションを付けて行います。

```
(%i1) load(vect);
(%o1)
/usr/local/share/maxima/5.9.2/share/vector/vect.mac
(%i2) e1:laplacian(x^2*y^2*z^2);
2 2 2
(%o2)
laplacian (x y z )
(%i3) express(e1);
2 2 2
d 2 2 2 d 2 2 2 d 2 2 2
(%o3)
--- (x y z ) + --- (x y z ) + --- (x y z )
2 2 2
dz dy dx
(%i4) ev(%, 'diff);
2 2 2 2 2 2
(%o4)
2 y z + 2 x z + 2 x y
(%i5) v1:[x1,x2,x3]~[y1,y2,y3];
```

```
(%o5) [x1, x2, x3] ~ [y1, y2, y3]
```

```
(%i6) express(v1);
```

```
(%o6) [x2 y3 - x3 y2, x3 y1 - x1 y3, x1 y2 - x2 y1]
```


8.7 積分

8.7.1 記号積分について

Maxima は記号積分, 数値積分の両方が行えます. 積分に関しては, Risch の積分も不完全ながら実装されています. その為, 単純に公式を当て嵌めるだけで積分の計算を行う様なシステムよりも一段と優れた処理が行えます.

猶, Maxima は初等函数 (有理式, 三角函数, 対数, 指数) と多少の拡張 (error 函数, dilogarithm) で可積分なものに限定しているので, $g(x)$ や $h(x)$ の様な未知函数の積分は, 形式的な扱い以上の事は行いません.

—— 記号積分を行う函数 ——

```
integrate(〈式〉, 〈変数〉)
integrate(〈方程式〉, 〈変数〉)
integrate([〈式1〉, …, 〈式n〉], 〈変数〉)
integrate([〈方程式1〉, …, 〈方程式n〉], 〈変数〉)
integrate(〈行列〉, 〈変数〉, 〈下限〉, 〈上限〉)
risch(〈式〉, 〈変数〉)
```

Maxima で記号積分を行う函数に, integrate 函数と risch 函数の二種類があります. integrate 函数から, risch 函数の本体を呼出す事も, ev 函数を利用する事で出来るので, 記号積分に関しては integrate 函数だけで済ませられます.

積分の計算で式が非常に繁雑になる場合や, 積分記号の文字による表示が不要な場合, 大域変数 display2d を false にすると, 一行で結果が表示されます. 複雑な式の計算を行う場合には非常に便利です.

integrate 函数は, 通常の式, 方程式 (演算子=を含む式) や, これらのリストや行列の積分も行えます. それに対し, risch 函数は比較の演算子 (=, >, <, >=, <=) を含まない通常の式の積分に限定されます.

ここで, integrate 函数を使って方程式の不定積分を行うと, 積分定数が結果の式に導入されます. この積分定数は順番が付いており, この順番が大域変数 integration_constant_counter に記録されています.

—— 積分定数を定める大域変数 ——

変数名	初期値	概要
integration_constant_counter	0	積分定数のカウンター

実際に, その動作を見てみましょう.

```
(%i1) integration_constant_counter;
(%o1) 0
(%i2) integrate(x^2,x);
3
x
--
3
(%o2)

(%i3) integrate(x^2=0,x);
3 /
x [
(%o3) -- = integrationconstant1 + I 0 dx
3 ]
/

(%i4) integration_constant_counter;
(%o4) 1
(%i5) integrate(x^3=0,x);
4 /
x [
(%o5) -- = integrationconstant2 + I 0 dx
4 ]
/

(%i6) integration_constant_counter;
(%o6) 2
```

この様に, 演算子=を含まない式の積分では, 結果に積分定数 integrationconstant が含まれませんが, 演算子=を含む式の積分では integrationconstant が出現します. これは, integration 関数が呼出す sinint 関数内部でその様に設定されている為で, この設定は, 他では行われていません.

integrate 関数から risch 関数を用いて Risch 積分を実行させる事が可能です. この場合は ev 関数を用います.

```
(%i21) ev(integrate(3^log(x),x),'risch);
              log(3) log(x)
              x %e
(%o21) -----
              log(3) + 1
(%i22) trigsimp(%);
              log(x)
              x 3
(%o22) -----
              log(3) + 1
(%i23)
```

猶, integrate 関数は depends 関数で設定される大域変数 dependencies の影響を受けません.

integrate 関数は定積分の計算も出来ます. この場合は defint 関数と同じ引数を取ります. 猶, 実際に定積分を実行するのは defint 関数です. 猶, defint 関数による定積分は, 数値計算を主体としたものではなく, 記号積分を主体としたものです. 数値積分が必要な場合, Maxima には romberg 関数や quanc8 関数 を用います.

広義の定積分では, 正の無限大には inf, 負の無限大には minf, 複素無限大には infinity が使えます. この inf と minf に関しては, -inf や -minf が各々 minf や inf と同値なものではない事に注意して下さい. Maxima の広義の積分や, その他の代入操作で inf や minf は普通に使えますが, -inf や -minf を用いると全く無意味な結果を得る事があるので注意が必要です. その為, -inf や -minf が現れる式は limit 関数で一度評価しましょう. 積分形式 (例えば, 幾つかの助変数に関してある数値を代入するまで計算出来ない積分) が必要であれば, 名詞型の 'integrate を利用します.

risch 関数は Risch の積分アルゴリズムから, Transcendental case を用いて式の積分を行います. 猶, Maxima では Risch アルゴリズムで代数的な場合は実装されていません. risch 関数は, integrate の主要部が処理出来ない入れ子状態の指数関数と対数関数の場合の処理が行えます. integrate は, これらの場合, 自動的に risch 関数を適用します.

```
(%i24) risch(x^2*erf(x),x);
```

```
(%o24)

$$\frac{\pi x^3 \operatorname{erf}(x) + (\sqrt{\pi} x^2 + \sqrt{\pi}) e^{-x^2}}{3 \pi}$$

```

```
(%i25) diff(%,x),ratsimp;
```

```
(%o25)

$$x^2 \operatorname{erf}(x)$$

```

risch 関数の動作を制御する大域変数に erf flag があります.

大域変数 erf flag

変数名	デフォルト値	概要
erf flag	true	risch 関数による erf 関数の挿入を制御

この大域変数 erf flag が false であれば, erf 関数が被積分関数の中に含まれていない場合, risch 関数が答に erf 関数を入れる事を抑制します.

ここで, erf 関数は次の error 関数の事です.

erf 関数

erf(x)

この関数の微分 $\frac{d}{dx}(\operatorname{erf}(x))$ は $\frac{2e^{-x^2}}{\sqrt{\pi}}$ となります.

8.7.2 変数の変換について

Maxima では被積分関数の変数を新しい変数で置換えて積分する事が可能です. この為に, changevar 関数を用います.

変数変換を行う関数

changevar(〈式〉, 〈f(x,y)〉, 〈y〉, 〈x〉)

changevar 関数は 〈式〉 に現われる全ての 〈x〉 に対する積分で 〈f(x,y)〉 = 0 を満す 〈y〉 を新しい変数とする変数変換を行います.

```
(%i17) assume(z>0)$
```

```
(%i18) I1:=integrate(%e^sqrt(1-y),y,0,1);
              1
              /
              [      sqrt(1 - y)
(%o18)      I  %e      dy
              ]
              /
              0
(%i19) changevar(I1,y+z^2-1,z,y);
              0
              /
              [      z
(%o19)      - 2 I  z %e  dz
              ]
              /
              - 1
(%i20) ev(%,risch);
              - 1
(%o20)      - 2 (2 %e  - 1)
```

changevar は総和 ($\sum$) や積 ($\prod$) の添字の変更にも使えます. この場合, 添字の変更は単純なシフトだけで, 次数の高い関数には出来ません.

```
(%i3) sum(a[i]*exp(i-5),i,0,inf);
              inf
              ====
              \      i - 5
(%o3)      >  %e      a
              /      i
              ====
              i = 0
```

```
(%i4) changevar(%,i-5-n,n,i);
      inf
      ====
      \      n
      >      %e  a
      /      n + 5
      ====
      n = - 5
```

8.7.3 有理式の記号積分

多項式 $f(x), g(x)$ の有理式 $\frac{f(x)}{g(x)}$ の積分は比較的容易に行えます. 但し, $\frac{1}{x^3 - x^2 - x + 4}$ の様に式の因数分解が容易に出来ない式の積分は, そのままでは上手く計算出来ません.

```
(%i3) integrate(1/(x^3-x^2-x+4),x);
      /
      [      1
      I ----- dx
      ] 3    2
      / x  - x  - x + 4
```

この様な有理式の記号積分に対しては, 大域変数 `integrate_use_rootsof` を `true` に設定する事で, 計算を行う事が可能です.

—— 代数的数の利用を制御する大域変数 ——

変数名	デフォルト値	概要
<code>integrate_use_rootsof</code>	<code>false</code>	代数的数を用いた <code>integrate</code> 関数による積分を実行

先程の例に対し, 大域変数 `integrate_use_rootsof` を `true` にして, `integrate` 関数を使って再度積分を行ってみましょう.

```
(%i4) integrate_use_rootsof:true;
(%o4)                                     true
(%i5) integrate(1/(x^3-x^2-x+4),x);
```

```

=====
\      log(x - %r1)
(%o5)  >  -----
/      2
=====  3 %r1  - 2 %r1 - 1
          3      2
          %r1 in rootsof(x  - x  - x + 4)

```

今度は積分の計算が出来ました。大域変数 `integrate.use_rootsof` が `true` の場合、`integrate` 関数は有理式の積分で、分母の多項式から定義される代数的数を導入して積分を実行します。この意味を上例で説明しましょう。まず、`integrate.use_rootsof` が `true` に設定されると、Maxima の `integrate` 関数は与式の分母 $x^3 - x^2 - x + 4$ から、方程式 $x^3 - x^2 - x + 4 = 0$ の根を決めます。Maxima では単純に `%r1` 等の `%r` で開始する変数を導入します。しかし、このままでは判り難いので、この例ではもう少し細かく説明します。この例の場合、三次方程式なので根は3個あり、それらを α, β, γ とします。すると、与式はこれらの根を用いると、 $\frac{1}{(x-\alpha)(x-\beta)(x-\gamma)}$ に変換されます。この様に分母が一次式の有理式の積になってしまえば、後は簡単に計算する事が可能です。実際の処理では、各根に記号を割当てて必要は無く、方程式 `xxx` の解、`%r2` といった括り方で十分です。

但し、この方法の気持ちの悪さは方程式の根 `%r1` が不明瞭ながら式に存在する事です。又、返却される式は名詞型の為に、安易に微分が出来ません。何故なら、Maxima によるこの処理は、単純に公式に当て嵌めているだけだからです。

8.7.4 記号積分の結果検証

記号積分の計算は、記号微分と比べると格段に難しい問題の為、比較的単純な式の計算でも思わぬ結果を得る事があります。その為、記号積分の結果は何等かの形で検算を行う事を強く薦めます。最も簡単な方法は積分した結果を微分して同じ結果が得られるかどうかを確認する事です。

Maxima の `integrate` 関数や `risch` 関数を使っても上手く計算出来ない簡単な式の例として、 $\sqrt{x + \frac{1}{x}} - 2$ を挙げておきます。この式は $\sqrt{\frac{(x+1)^2}{x}}$ に変形出来ませんが、この式の形の違いで結果が大きく異なります。

```

(%i44) integrate(sqrt(x+1/x-2),x);
          3/2

```

```

(%o44)

$$\frac{2x^3 - 6\sqrt{x}}{3}$$

(%i45) integrate(sqrt(factor(x+1/x-2)),x);
/
[ abs(x - 1)
(%o45)  I ----- dx
  ] sqrt(x)
/
(%i46) assume(x<1 and x>0);
(%o46) [x < 1, x > 0]
(%i47) integrate(sqrt(x+1/x-2),x);
3/2
2 x  - 6 sqrt(x)
(%o47) -----
3
(%i48) integrate(sqrt(factor(x+1/x-2)),x);
3/2
2 x  - 6 sqrt(x)
(%o48) - -----
3
(%i49) diff(%,x);
3
3 sqrt(x) - -----
sqrt(x)
(%o49) - -----
3
(%i50) ratsimp(%);
x - 1
(%o50) - -----
sqrt(x)

```

この例では、同じ integrate でも $\sqrt{\frac{(x+1)^2}{x}}$ の場合は名詞型を返しており、何も考え

ずに $\frac{x-1}{\sqrt{x}}$ の計算を行ってはいません. 又, assume 函数を使って, 条件 $0 < x < 1$ を

追加した場合でも, $\sqrt{x + \frac{1}{x} - 2}$ の integrate の計算は間違っています.

この様に Maxima の積分は正しい答を返すとは限りませんが, 内部処理を適切に行う事で正しい答を得る事も可能な場合もあります. これは Maxima に限った話ではなく, 数式処理一般でも言える事です. 更に, 記号積分の結果は面倒でも確認した方が安全な事は強調しておきます.

何故, 式の形の違いで計算結果に違いが出るのでしょうか? これは結局, 式の並びの照合等によって処理を行っている為, 式を変形していれば, 並びも勿論異なる為に照合の結果も異なり, それによって処理の流れが違ふ為です. 数値計算で積分を行う場合, 式の並びは無関係で, 函数の値を主に見る為, この様な現象は累積誤差の事を除くとまず生じません.

並び照合の結果が違っていても, 正しく処理が出来ていれば, 最終的に一致する筈ですが, この例の様に何処かの処理を間違えると当然結果が異なります. 積分の計算の場合は特に, expand 函数で式を展開したり, factor 函数で因子分解する等の処理を実行して積分したものと結果を照合する事は, 正しい答を得る為の有効な手段です.

しかし, 実際はこれでも不十分な事があります. 例えば, $\frac{3}{5 - 4\cos(x)}$ の積分です.

```
(%i13) integrate(3/(5-4*cos(x)),x);
                                     3 sin(x)
(%o13)      2 atan(-----)
                        cos(x) + 1

(%i14) trigsimp(diff(%o13),x);
                                     3
(%o14)      - -----
                        4 cos(x) - 5
```

積分した結果を微分すると, 元の式が出ていますので, 正しい結果が出ている様に思えます. しかし, 残念ながら計算は間違っています. 何故なら被積分函数は滑かな連続函数ですが, 結果の方は不連続函数になっています.

この様な代物はグラフを利用して積分した函数を描くと一目で判ります.

この様に, 結果のグラフを描いて見るのも, 計算結果を確認する上では非常に有効な手段です.

Maxima は積分処理の為の函数を幾つか持っています. その中でも, integrate 函数は最もよく使われるものです. この integrate 函数は不定積分も定積分も処理出来ます. 定

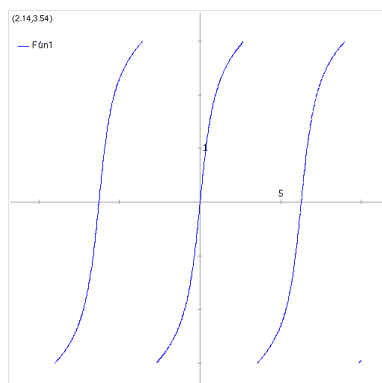


図 8.3: $2\operatorname{atan}\left(\frac{3\sin(x)}{\cos(x)+1}\right)$ のグラフ

積分だけであれば, `defint` 関数もあります. 更に, 極限操作が必要な場合には, `ldefint` 関数もあります. 但し. この関数は非常に曲者で, `integrate` 関数で記号積分した結果に境界値を代入する代物です. 従って, 区間内に極が存在する場合, その極を検出せずに安易に計算を行う為, 注意が必要です.

参考迄に, `defint` 関数の処理を簡単に説明しておきます. 通常は `integrate` 関数は, LISP 内部では `$integrate` です. この関数は内部で `sinint` 関数を呼出し, その結果に上限と下限の値を代入しています. `sinint` でも `risch` 積分を行う `rischint` を呼出す事も出来ませんが, `ev` 関数を用いて `rischint` を用いる様に指定する事が出来ません. この点を修正する為には `defint.lisp` の `antideriv` の修正が最低でも必要となります.

`defint` 関数や `integrate` 関数による定積分の計算で, 下限や上限に記号や式が含まれている場合, その正負を尋ねてくる事があります. この場合, 正であれば `pos;`, 負であれば `neg;`, 零であれば `zero;` と入力します. 更に, `assume` 関数を用いて正負の指定を予め行っていれば, Maxima はこの様な質問を行いません.

```
(%i24) integrate(sqrt(2*x-x^2),x,0,a);
```

```
Is a positive, negative, or zero?
```

```
pos;
```

```
(%o24)

$$\frac{\operatorname{asin}(a-1) + (a-1)\sqrt{2a-a^2}}{2} + \frac{\pi}{4}$$

```

```
(%i25) assume(a>0 and a<1);
(%o25) [a > 0, a < 1]
(%i26) integrate(sqrt(2*x-x^2),x,0,a);
(%o26) 
$$\frac{(a-1)\sqrt{2a-a^2} - \arcsin(1-a)}{2} + \frac{\pi}{4}$$

```

—— 留数を計算する関数 ——

residue(〈式〉,〈変数〉,〈点〉)

residue 関数は〈点〉回りの〈式〉の複素平面上での留数を計算します。猶、留数は式の laurent 級数展開を行った時の $(\langle \text{変数} \rangle - \langle \text{点} \rangle)^{-1}$ の項の係数になります。

```
(%i5) residue(x/(x^2+1),x,%i);
(%o5) 
$$\frac{1}{2}$$

(%i6) residue(sin(x)/x^4,x,0);
(%o6) 
$$-\frac{1}{6}$$

```

—— ラプラス変換に関連する関数 ——

laplace(〈式〉,〈旧変数〉,〈新変数〉)
ilt(〈式〉,〈旧変数〉,〈新変数〉)
delta(t)

laplace 関数は〈旧変数〉と〈新変数〉に対する〈式〉の Laplace 変換を計算します。逆 laplace 変換は ilt です。

〈式〉は多項式に函数 exp, log, sin, cos, sinh, cosh と erf 函数を含むもの, atvalue の従属変数が使われている定数係数の線形微分方程式でも構いません。

初期条件は零で指定されていなければならないので、他の一般解の何処かに押込める境界条件があれば、その境界条件に対して一般解を求めて値を代入して定数消去が出来ます。

〈式〉に畳込み (convolution integral) を含んでも構いません. laplace 関数が適切に動作する為に, 関数の従属性をはっきりと表示していなければなりません. つまり, 関数 f が変数 x と y に従属しているのであれば, `laplace('diff(f(x,y),x),x,s)` の様に関数 f が現れる場合は何時でも `f(x,y)` と記述する必要があります.

猶, laplace 関数は depends 関数で設定される大域変数 dependencies の影響を受けません.

```
(%i106) laplace(%e^(2*t+a)*sin(t)*t,t,s);
```

```

      a
      %e  (2 s - 4)
(%o106) -----
      2      2
      (s  - 4 s + 5)

```

```
(%i107) ilt(%,s,x);
```

```

      2 x + a
(%o107) x %e      sin(x)

```

ilt 関数は〈新変数〉と〈旧変数〉に対する〈式〉の逆 Laplace 変換を計算します. 〈式〉は分母が一次と二次の因子を持った有理式でなければなりません. ilt 関数による処理を効率的に行う為には有理式の展開を予め実行しておくとう良いでしょう.

laplace 関数と ilt 関数の双方による結果に対し, solve 関数や linsolve 関数を上手く使うと, 単変数の線形常微分方程式や畳込積分方程式を解く事が出来ます.

```
(%i11) 'integrate(sinh(a*x)*f(t-x),x,0,t)+b*f(t)=t^2;
```

```

      t
      /
      [
      2
(%o11) I  f(t - x) sinh(a x) dx + b f(t) = t
      ]
      /
      0

```

```
(%i12) laplace(%,t,s);
```

```

(%o12)      a laplace(f(t), t, s)  2
      b laplace(f(t), t, s) + ----- = --
                        2      2      3
                        s  - a      s

(%i13)  linsolve([%],['laplace(f(t),t,s)]);

                        2      2
                        2 s  - 2 a
(%o13)      [laplace(f(t), t, s) = -----]
                        5      2      3
                        b s  + (a - a b) s

(%i14)  ilt(rhs(first(%)),s,t);
Is  a b (a b - 1)  positive, negative, or zero?

pos;

      sqrt(a b (a b - 1)) t
      2 cosh(-----)      2
                  b      a t      2
(%o14)  - ----- + ----- + -----
      3 2      2      a b - 1  3 2      2
      a b  - 2 a b + a      a b  - 2 a b + a

```

delta 関数は Dirac の δ 関数です. 猶,laplace 関数のみが δ 関数を認識しています.

```

(%i38) laplace(delta(t-a)*sin(b*t),t,s);
Is  a  positive, negative, or zero?

```

pos;

```

      - a s
(%o38)      sin(a b) %e

```

この例では変数 a に関して何らの仮定や割り当てが無い為に, "Is a positive, negative, or zero?" と Maxima が尋ねています. assume を用いて $a > 0$ と仮定した場合を以下に示します.

```

(%i39) assume(a<0);
(%o39)      [a < 0]

```

```
(%i40) laplace(delta(t-a)*sin(b*t),t,s);
(%o40) 0
```

—— 定積分を行う関数 ——

```
defint(〈式〉,〈変数〉,〈下限〉,〈上限〉)
ldefint(〈式〉,〈変数〉,〈下限〉,〈上限〉)
tldefint(〈式〉,〈変数〉,〈下限〉,〈上限〉)
```

defint 関数は定積分を実行する関数です. 内部的に integrate を利用しており, 原始関数を求めると, 単純に〈上限〉と〈下限〉の値を代入したものの差を取るものです. 但し, 後述の ldefint と比較して, 区間(〈下限〉,〈上限〉)に於ける〈式〉の極の判定を行っているので, ldefint よりは安全です. 但し, romberg 等の数値成分による手法の方が間違いが少ないので, その点は注意して使う必要があります.

ldefint 関数は〈変数〉の〈上限〉と〈下限〉に関する〈式〉の不定積分に対し, limit 関数を用いた評価を行い,〈式〉の定積分を計算します. 猶, 上限と下限に minf と inf といった値を与える場合には注意が必要です. 又, -inf や -minf の様に符号を付けて用いると, 無意味な結果を得る事があるので注意が必要になります.

```
(%i20) ldefint(exp(-x)*sin(x),x,0,-minf);
          minf          minf
          %e      sin(- minf)  %e      cos(- minf)  1
(%o20)  - ---- - ---- + -
          2          2          2
(%i21) ldefint(exp(-x)*sin(x),x,0,inf);
          1
(%o21)  -
          2
```

猶, ldefint は内部的で関数の極の判別を一切行わずに, integrate で安易に記号積分した結果に, 上限と下限を limit 関数で代入するだけの関数です.

一応, 右極限と左極限を下限と上限で取る様にしていますが, 単純に上限に対しては 'minus, 下限に対しては 'plus を内部的に付けるだけなので, 上限や下限で不連続になる函数の場合, 上限と下限の大小関係を逆にして ldefint 関数を用いて計算すれば無意味になる可能性もあります.

次の例をよく吟味して下さい.

猶, `ldefint` 関数では, `zeroa`, `zerob` が使えます. `zeroa` が 0 の右極限, `zerob` が 0 の左極限を表現します.

```
(%i7) ldefint(1/x^2,x,zeroa,1);
```

$$\lim_{x \rightarrow 0^+} -\frac{1}{x}$$

```
(%o7)
```

```
(%i8) ldefint(1/x^2,x,zeroa,-1);
```

$$\lim_{x \rightarrow 0^+} \frac{1}{x}$$

```
(%o8)
```

$$x \rightarrow 0 - x$$

但し, `1+’zeroa` の様な使い方は出来ないので注意します.

猶, `ldefint` 関数は `defint` 関数と同様に積分で Risch 積分を用いる事が出来ません.

`tldefint` 関数は大域変数 `tlmswitch` が `true` に設定された `ldefint` 関数に相当します. 猶, 大域変数 `tlmswitch` が `true` の場合, 極限の計算で Taylor 展開を利用する事を意味します.

8.7.5 数値積分について

Maxima には記号積分だけではなく, 数値計算による定積分の計算も行えます. この処理を行う関数には `quanc8` 関数と `romberg` 関数があります.

数値積分を行う関数

```
quanc8( '〈被積分関数〉, 実数1, 実数2 )
quanc8( 〈被積分関数〉, 変数, 実数1, 実数2 )
romberg( 〈被積分関数〉, 〈積分変数〉, 〈実数1〉, 〈実数2〉 )
romberg( 〈被積分関数〉, 〈実数1〉, 〈実数2〉 )
```

先ず, `quanc8` 関数は予め `load("qq")` で読み込む必要があります. `quanc8( '〈被積分関数〉, 実数1, 実数2 )` で, 最初の引数で指定された関数の定積分を区間 実数₁ から 実数₂ で計算します. 第一引数に関数名を用いる場合, 関数名に単引用符'を付けなければなりません

`quanc8( 〈被積分関数〉, 変数, 実数1, 実数2 )` で, 関数か式 (最初の引数) の変数 (二番目の引数) で区間 実数₁ から 実数₂ で定積分を計算したものになります.

使われる手法は Newton-Cotes の 8 次多項式による求積法で, ルーチンは適応型です. `romberg` 関数は `romberg` 積分を行う関数です. `romberg` 関数は `quanc8` 関数とは違い, Maxima に自動的に読み込まれます.

最初の引数は `translate` 関数で変換された関数か `compile` 関数でコンパイルされた関数でなければなりません. 猶, `compile` 関数でコンパイルされた関数の場合は, 浮動小数点型, 即ち, `flonum` 型の関数として宣言されていなければなりません. 関数が `translate` 関数で変換されたものでなければ, `romberg` 関数は `translate` 関数で変換せずにエラーを返します. 積分の精度は大域変数 `rombertol` と `rombergit` で制御されます.

この `romberg` 関数は再帰的に呼び出されていても構わないので, 二重, 三重積分が実行可能です.

猶,romberg 関数の〈実数₁〉と〈実数₂〉は, 内部で倍精度の浮動小数点を用いている為, 多倍長精度 (bigfloat 型) に変換した数値は扱えません.

相対誤差が大域変数 rombertol よりも小さければ romberg 関数は計算結果を返します. 諦める前に大域変数 romgergit 倍の刻幅を半分にして試みます.romberg 関数は反復と関数評価の大域変数 romnergabs と rombergmin で制御されます.

一般的に,quanc8 関数の方が計算速度, 精度と安定性で romberg 関数に勝っている事が多いのですが,romberg 関数の方が精度が良好な場合もあります. 以下の $\sqrt{2x-x^2}$ の積分の例では romberg 関数の方が精度で勝り, $e^{-x}\sin(x)$ の積分では計算速度で romberg 関数が勝っています.

```
(%i31) showtime:all;
Evaluation took 0.00 seconds (0.00 elapsed) using 80 bytes.
(%o31)                                     all
(%i32) romberg(sqrt(2*x-x^2),x,0,1);
Evaluation took 0.18 seconds (0.18 elapsed) using 316.305 KB.
(%o32)                                     .7853897937007632
(%i33) quanc8(sqrt(2*x-x^2),x,0,1);
Evaluation took 0.02 seconds (0.02 elapsed) using 35.602 KB.
(%o33)                                     .7849358178522697
(%i34) integrate(sqrt(2*x-x^2),x,0,1);
Evaluation took 0.12 seconds (0.12 elapsed) using 212.039 KB.
(%o34)                                     %pi
                                     ---
                                     4
(%i35) bfloat(%);
Evaluation took 0.00 seconds (0.00 elapsed) using 808 bytes.
(%o35)                                     7.853981633974483B-1
(%i36) bfloat(integrate(sqrt(2*x-x^2),x,0,1));
Evaluation took 0.12 seconds (0.12 elapsed) using 212.875 KB.
(%o36)                                     7.853981633974483B-1
(%i37) romberg(exp(-x)*sin(x),x,0.,1.);
Evaluation took 0.01 seconds (0.00 elapsed) using 28.227 KB.
(%o37)                                     .2458370426035679
(%i38) bfloat(integrate(exp(-x)*sin(x),x,0.,1.));
Evaluation took 0.13 seconds (0.13 elapsed) using 298.562 KB.
```

(%o38)

2.458370070002374B-1

数値積分の方が上記の極を検出し易い事もあり、両者の結果を比較するのも検算の方法としては良いでしょう。

———— romberg 関数に影響を与える大域変数 ————

変数名	初期値	概要
rombergabs	0.0	romberg 関数の終了条件を与える変数
rombergit	11	刻幅を設定
rombertol	1.0E-4	romberg 関数の精度
rombergmin	0	関数評価の最小回数

大域変数 rombergabs は romberg 関数の終了条件を与える大域変数の一つです。romberg 関数内部の反復処理で生成された値の列を $y[0], y[1], y[2], \dots$ とする時、 n 回目の反復処理で、 $(\text{abs}(y[n]-y[n-1]) \leq \text{rombergabs})$ か $\text{abs}(y[n]-y[n-1])/(y[n]=0.0 \text{ ならば } 1.0, \text{ それ以外は } y[n]) \geq \text{rombertol}$ を満した時点で、romberg 関数は答を返します。その為、rombergabs が 0.0 であれば相対誤差の検証が出来ます。この追加変数は小さな値域で積分計算を行いたい時に便利です。そこで、小さな主要な値域で最初に積分する事で相対的精度検証を用い、後に続く残りの値域上の積分は絶対的精度の検証で用います。

romberg 積分命令の精度は大域変数の rombertol と rombergit で支配されます。romberg は、隣り合った近似解での相対差が rombertol よりも小さければ値を返します。諦める前に刻幅の rombergit を半分にして試行します。

大域変数 rombergmin は romberg 関数による関数評価の最小数を制御します。romberg 関数はその第一引数を少なくとも $2^{\text{rombergmin}+2} + 1$ 回評価します。これは通常の収束テストが時々悪い通り方をする周期的関数の積分に有効です。

———— 二重積分を行う関数 ————

`dblnt( ' $\langle F \rangle$ ', ' $\langle R \rangle$ ', ' $\langle S \rangle$ ',  $\langle \text{浮動小数点}_1 \rangle$ ,  $\langle \text{浮動小数点}_2 \rangle$  )`

dbint 関数は二重積分の数値計算を行う関数で、Maxima の処理言語で記述されています。この関数を利用する為には予め `load(dbint)` で dbint 関数を読込んでおく必要があります。

先ず、`dblnt('F','R','S',浮動小数点1,浮動小数点2)` で以下の二重積分を計算します

$$\int_{\text{浮動小数点}_1}^{\text{浮動小数点}_2} \int_{R(x)}^{S(x)} F(x, y) dy dx$$

第一引数の $\langle F \rangle$ は x, y の 2 変数の関数, 第二引数と第三引数の $\langle R \rangle$ と $\langle S \rangle$ は各々変数 x の関数で, `dblint` 関数には関数名のみを名詞型で引渡します. 更に, これらの関数は全て, `translate` 関数で LISP の関数に変換されたものか, `compile` 関数でコンパイルされたものでなければなりません. その為, これらの関数は全て `flonum` 型の関数でなければならず, 多倍長精度は扱えません.

`dblint` 関数は $\langle R \rangle$ と $\langle S \rangle$ の両方に Simpson 則を用います. そして, `dblint` 関数は二つの大域変数 `dblint_x`, `dblint_y` を持ち, 各々の大域変数が x と y の区間の分割数を定めます. 猶, 収束性を改善する為に大域変数 `dblint_x` と `dblint_y` を大きくした場合は計算時間が増大します.

`dblint` 関数は X 軸を大域変数 `dblint_x` の値で分割し, X 軸上の各値に対して最初に $R(x)$ と $S(x)$ を計算します. それから, $R(x)$ と $S(x)$ の間の Y 軸を大域変数 `dblint_y` の値で分割し, Y 軸に沿って積分を Simpson 則を用いて計算します.

それから, X 軸に沿った積分を関数の値を Y の積分で Simpson 則を用いて計算します. この手順は色々な理由で数値的に不安定であるが, それなりに速いものです.

但し, 高周波成分を持った関数や特異点 (領域に極や分岐点) を持つ関数に対する適用は避けて下さい.

Y 軸方向の積分は $R(x)$ と $S(x)$ がどれだけ離れているかに依存し, 距離 $S(X)-R(X)$ が X で急速に変化すれば, Y 軸方向の積分で大きな誤差が発生するかもしれません.

関数値は保存されない為, その関数の計算に時間がかかるものであれば, 何かを変更する度に再計算する羽目になるので, その分時間がかかります.

— `dblint` 関数に影響を与える大域変数 —

変数名	初期値	概要
<code>dblint_x</code>	10	X 軸方向の分割数
<code>dblint_y</code>	10	Y 軸方向の分割数

二重積分を行う関数 `dbiint` 関数は二つの大域変数 `dblint_x`, `dblint_y` を持っています. それらは X, Y の区間の分割数を定め, $2 * \text{dblint\_x} + 1$ 点が X 方向に計算され, Y 方向は $2 * \text{dblint\_y} + 1$ 点となります. これらの変数は勿論, 互いに独立して変更する事が可能です.

8.7.6 antid パッケージ

`antid` パッケージには `antid`, `antidiff` と `nonzeroandfreeof` といった関数が含まれたパッケージです.

antidiff と antid

```
antid(⟨式⟩,⟨x⟩,⟨u(x)⟩)
antidiff(⟨式⟩,⟨x⟩,⟨u(x)⟩)
nonzeroandfreeof(⟨x⟩,⟨y⟩)
```

antid 関数は与式の不定積分を行います。antid の返却値は二成分のリストで、このリスト L とする時、`L[1]+integrate(L[2],x)` が与式の不定積分となります。ここで、関数 $\langle u(x) \rangle$ は未知関数でも構いません。

```
(%i8) load(antid);
(%o8) /usr/local/share/maxima/5.9.2/share/integration/antid.mac
(%i9) antid(sin(3*x+1),x,3*x+1);
                                cos(3 x + 1)
(%o9) [- -----, 0]
                                3
(%i10) antid(sin(u(x)+1),x,u(x));
(%o10) [0, sin(u(x) + 1)]
```

antidiff は内部で antid 関数を用いた関数で、antid 関数で不定積分した結果を `L[1]+integrate(L[2],x)` の形式に変換して表示します。

```
(%i11) antidiff(sin(3*x+1),x,3*x+1);
                                cos(3 x + 1)
(%o11) - -----
                                3
(%i12) antidiff(sin(u(x)+1),x,u(x));
/
[
(%o12) I sin(u(x) + 1) dx
]
/
```

関数 `nonzeroandfreeof` は真偽関数で、 $\langle y \rangle$ が零でなく、 $\langle x \rangle$ が $\langle y \rangle$ に含まれない場合に true を返す関数です。この真偽関数を用いて、antid や antidiff の規則を定めています。

8.8 常微分方程式

8.8.1 常微分方程式の扱い

Maxima での常微分方程式の記述は通常の方程式と同様に、演算子= を挟んで左右に式が記述される形となりますが、この式中に名詞型の微分'diff が含まれる式の事です。Maxima では常微分方程式の一般解を計算する函数に ode2 函数と desolve 函数があります。ここで、desolve 函数で扱える常微分方程式は、未知函数がどの変数に依存するものかを明示的に記述したものでなければなりません。

例えば、次の書式は desolve 函数にとっては正確な書式ではありません。何故なら、函数 f と函数 g の変数が何であるかが明確でないからです。

```
'diff(f,x,2)=sin(x)+'diff(g,x);
'diff(f,x)+x^2-f=2*'diff(g,x,2);
```

desolve 函数の場合、常微分方程式の未知函数が何の函数であるかを下記のように明確に記述しなければなりません。

```
'diff(f(x),x,2)=sin(x)+'diff(g(x),x);
'diff(f(x),x)+x^2-f(x)=2*'diff(g(x),x,2);
```

desolve 函数に対し、ode2 函数は $x^2 \cdot \text{'diff}(y,x) + 3 \cdot y \cdot x = \sin(x)/x$ の様に函数と変数との関係を明記しなくても構いません。

微分方程式の初期値問題を解く場合、desolve 函数を用いる場合と ode2 函数を用いる場合で解き方の手順が異なります。

最初に、desolve 函数を用いて初期値問題を解く場合、atvalue 函数を用いて初期値を函数の属性として与えます。ここで atvalue 函数による初期値設定は desolve 函数で微分方程式を処理する前に行わなければ意味がありません。

この atvalue 函数の構文を以下に示しておきます。

— atvalue 函数の構文 —

```
atvalue(<式>,[<x1>=<a1>,...,<xn>=<an>],<c>)
atvalue(<式>,<x>=<a>,<c>)
```

この atvalue 函数は多変数函数に対しては、第二引数に $[<x_1>=<a_1>,\dots,<x_n>=<a_n>]$ の書式で点を指定し、第三の引数の $<c>$ にその点での値を設定します。

又、一変数函数の場合は第二引数をリストの形式ではなく、 $<x>=<a>$ の書式にしても構いません。

では, 簡単な例を解いてみましょう.

```
(%i66) eq1:'diff(y(x),x,2)+2*x=sin(x);
                2
                d
(%o66)  --- (y(x)) + 2 x = sin(x)
                2
                dx
(%i67) atvalue('diff(y(x),x),x=0,0);
(%o67) 0
(%i68) atvalue(y(x),x=0,0);
(%o68) 0
(%i69) desolve([eq1],[y(x)]);
                3
                x
(%o69) y(x) = - sin(x) - -- + x
                3
```

この例では, 常微分方程式 $\frac{dy(x)}{dx} + 2x = \sin(x)$ を初期条件 $\left[y(0) = 0, \frac{dy(x)}{dx}|_0 = 0 \right]$ で解いています.

次に, ode2 関数を用いる場合では, 与えられた一般解と初期条件から特殊解を計算する bc 関数, ic1 関数や ic2 関数を併用します.

```
(%i14) x^2*'diff(y(x),x)+3*y(x)*x=sin(x)/x;
                2 d
                x (--- (y(x))) + 3 x y(x) = ----- sin(x)
                dx
(%o14)
(%i15) ode2(%,y(x),x);
                %c - cos(x)
(%o15) y(x) = -----
                3
                x
(%i16) ic1(%o15,x=%pi,y(%pi)=0);
                cos(x) + 1
(%o16) y(x) = - -----
                3
```

この例では、微分方程式 $x^2 \frac{dy}{dx} + 3xy = \frac{\sin(x)}{x}$ を ode2 関数を用いてその一般解を求め、ic1 関数から、 $x = \pi$ の場合に y が 0 となる条件で特殊解を求めています。

```
(%i91) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
                2
                d y      dy 3
(%o91)  ---- + y (--) = 0
                2      dx
                dx

(%i92) ode2(%,y,x);
                3
                y  + 6 %k1 y
(%o92)  ----- = x + %k2
                6

(%i93) ratsimp(ic2(%o92,x=0,y=0,'diff(y,x)=2));
                3
                2 y  - 3 y
(%o93)  - ---- = x
                6

(%i94) bc2(%o92,x=0,y=1,x=1,y=3);
                3
                y  - 10 y      3
(%o94)  ----- = x - -
                6              2
```

————— 一般解を求める関数 —————

```
desolve([< 方程式1> , ..., < 方程式n>], [< 変数1> , ..., < 変数n> ])
ode2(< 常微分方程式 > , < 従属変数 > , < 独立変数 > )
```

desolve 関数は、常微分方程式系を与えられた変数に対して解きます。常微分方程式の初期条件を設定する場合、desolve 関数を呼出す前に atvalue 関数で与えなければなりません。desolve の引数としては < 方程式_i> で構成されるリストと従属変数 < 変数₁> , ..., < 変数_n> のリストを指定します。desolve 関数では関数と変数の関連性を明確に指定しなければなりません。

解はリストの形式で返却されますが、解を得られなかった場合、desolve 関数は false を

返します.

```
(%i1) 'diff(f(x),x)='diff(g(x),x)+sin(x);
              d          d
(%o1)      -- (f(x)) = -- (g(x)) + sin(x)
              dx          dx
(%i2) 'diff(g(x),x,2)='diff(f(x),x)-cos(x);
              2
              d          d
(%o2)      --- (g(x)) = -- (f(x)) - cos(x)
              2          dx
              dx
(%i3) atvalue('diff(g(x),x),x=0,a);
(%o3)      a
(%i4) atvalue(f(x),x=0,1);
(%o4)      1
(%i5) desolve([%o1,%o2],[f(x),g(x)]);
              x          x
(%o5)      [f(x) = a %e  - a + 1, g(x) = cos(x) + a %e  - a + g(0) - 1]
(%i6) [%o1,%o2],%o5,diff;
              x          x          x          x
(%o6)      [a %e  = a %e , a %e  - cos(x) = a %e  - cos(x)]
```

この例で, `[%o1,%o2],%o5,diff;` は `ev` 関数による評価の一つの形態です. 詳細は 5.8.2 の小節を参照して下さい. この場合は式 `[%o1,%o2]` の $f(x)$ と $g(x)$ を代入し, `%o1` と `%o2` に含まれる微分の名詞型を評価させ, `desolve` による結果の検証を行っています. この評価の結果 `%o6` で演算子 `=` の両辺の式が等しい事から解が正しく求められている事が判ります.

`ode2` 関数は 3 個の引数を取ります. 最初の〈常微分方程式〉は一階, 又は二階の常微分方程式を与えます. 猶, 常微分方程式の右側 (`rhs(exp)`) が 0 ならば, 左側のみを与えるだけでも構いません. 第二引数には〈従属変数〉, 最後の引数が〈独立変数〉となります.

求解に成功すると, 従属変数に対する陽的な解, 或いは陰的な解の何れかを返します. ここで記号 `%c` は一階の方程式の定数, 記号 `%k1` と記号 `%k2` は二階の方程式の定数を表記する為に用いられます.

ode2 函数が何らかの理由で解が得られなかった場合, エラーメッセージの表示等の後に false を返します.

現在, 一階常微分方程式向けに実装され, 検証されている解法は, 線型, 分離法, 厳密 (積分因子が多分要求されます), 同次, bernoulli 方程式, そして一般化同次法があります. 二階の常微分方程式に対しては, 定数係数, 厳密, 定数係数に変換可能な非定数係数を持つ線型同次方程式, Euler 又は同次元方程式, 仮想変位法, そして変形分離で解ける二つの独立な一階の線型な方程式に縮約可能となる方程式を含まないものがあります. 常微分方程式を解く手順では, 幾つかの変数は純粋に情報的な目的, method が記述する解法の集合です. 例えば, linear, intfactor が記述する積分因子を用い, odeindex は Bernoulli 法や一般化同次法の添字を記述し, yp は仮想変位による特殊な解法を記述しています..

—— 境界値問題を解く函数 ——

```
bc2(〈一般解〉, 〈x の値1〉, 〈y の値1〉, 〈x の値2〉, 〈y の値2〉)
ic1(〈一般解〉, 〈x の値〉, 〈y の値〉)
ic2(〈一般解〉, 〈x の値〉, 〈y の値〉, 〈y の微分値〉)
```

bc2 函数は, 二階の微分方程式の境界条件問題を解きます. ここで 〈一般解〉 は ode2 函数等で計算した微分方程式の一般解です. 二階の方程式の一般解では定数が二つ現われる為, 特殊解を求める為, 異なった二点での連立方程式を解く必要があります.

その為, 〈x の値₁〉と 〈y の値₁〉が一つの点での値 〈x の値₂〉と 〈y の値₂〉がもう一つの別の点での値を定めます. ここで値の与え方は一般解の変数を x と y とすると, 〈x の値₁〉と 〈y の値₁〉を $x=x_0$ や $y=y_0$ の様に 〈対応する変数〉=〈境界値〉の書式で記述します.

ic1 函数は初期値問題 (ivp) と境界値問題 (bvp) を解く為の ode2 パッケージに含まれるプログラムです. 〈一般解〉は ode2 等で計算した微分方程式の一般解です. そして, 後の二つが境界条件を与えます. 一般解の変数を x, y とすると, 〈x の値₁〉と 〈y の値₁〉は $x = x_0$ や $y = y_0$ の様に 〈対応する変数〉=〈境界値〉の書式になります.

ic2 函数は二階の常微分方程式の境界値問題を解く函数です. 〈一般解〉は ode 函数等で計算した微分方程式の一般解となり, 後の二つがその境界条件を与えます.

一般解の変数を x, y とすると, 〈x の値〉と 〈y の値〉は, x が 〈x の値〉の場合, y の値が 〈y の値〉で y を x で微分した函数の, $x = \langle x \text{ の値} \rangle$ での値が 〈y の微分値〉となります.

8.9 特殊関数

8.9.1 Airy 関数

Airy 関数に関連する関数

```
airy_ai(< 変数 >)
airy_bi(< 変数 >)
airy_dai(< 変数 >)
airy_dbi(< 変数 >)
```

airy_ai 関数と airy_bi 関数は、夫々 Airy の Ai 関数と Bi 関数を Maxima に実装したものです。又、airy_dai 関数と airy_dbi 関数は、夫々 Airy の Ai 関数と Bi 関数の導関数になります。この関係は Maxima でも定義されています。

$$\begin{aligned} \text{(a)} \quad \text{diff}(\text{airy\_ai}(z), z) &\Rightarrow \text{airy\_dai}(z) \\ \text{(b)} \quad \text{diff}(\text{airy\_bi}(z), z) &\Rightarrow \text{airy\_dbi}(z) \end{aligned}$$

猶、逆は定義されていません。

ここで Airy 関数は二階の常微分方程式 $y'' - yz = 0$ の線形独立な解として得られる関数です。その為、Maxima でこれらの関数もこの微分方程式を満します。

```
(%i2) diff(airy_ai(z), z, 2) - airy_ai(z) * z;
(%o2) 0
(%i3) diff(airy_bi(z), z, 2) - airy_bi(z) * z;
(%o3) 0
(%i4) diff(airy_dai(z), z) - airy_ai(z) * z;
(%o4) 0
(%i5) diff(airy_dbi(z), z) - airy_bi(z) * z;
(%o5) 0
```

これらの関数は airy.lisp⁴ で定義されています。

⁴©2005 David Billinghurst

8.9.2 Bessel 函数

第一種 Bessel 函数

第一種 Bessel 函数

```
bessel(< 変数 >, < 次数 >)
bessel.j(< 次数 >, < 変数 >)
bessel.i(< 次数 >, < 変数 >)
scaled.bessel.i(< 次数 >, < 変数 >)
scaled.bessel.i0(< 変数 >)
scaled.bessel.i1(< 変数 >)
```

bessel 函数は, 第一種の Bessel 函数です. 但し, この函数は bessel.j 函数を使いましょう. bessel.j 函数は第一種の Bessel 函数です. この函数は引数を二つ取り, 第一引数が次数となります. 具体的には次の式で定義されています.

$$\text{bessel.j}(v, z) =_{\text{def}} \sum_{k=0}^{\infty} \frac{(-1)^k 2^{-v-2k} z^{v+2k}}{k! \Gamma(v+k+1)}$$

bessel.i は第一種の変形 Bessel 函数です. この函数は引数を二つ取り, 第一引数が次数となります. 具体的には次の式で定義されています.

$$\text{bessel.i}(v, z) =_{\text{def}} \sum_{k=0}^{\infty} \frac{2^{-v-2k} z^{v+2k}}{k! \Gamma(v+k+1)}$$

scaled.bessel.i 函数は第一種の変形 Bessel 函数です. この函数は引数を二つ取り, 第一引数が次数となります. 具体的には次の式で定義されています.

$$\text{scaled.bessel.i}(v, z) =_{\text{def}} \exp(-|z|) \text{bessel.i}(v, z)$$

scaled.bessel.i0 函数と scaled.bessel.i1 函数は, scaled.bessel.i 函数の第一引数を夫々 0, 1 とした函数です. その為, 引数は一つだけです.

第二種 Bessel 関数

第二種 Bessel 関数

```
bessel.k(< 次数 >, < 変数 >)
bessel.y(< 次数 >, < 変数 >)
scaled.bessel.y(< 次数 >, < 変数 >)
```

Bessel 関数に関連する大域変数

Bessel 関数に関連する大域変数

```
besselexpand false Bessel 関数の展開を制御
```

大域変数 `besselexpand` は Bessel 関数の自動展開を制御する大域変数です。大域変数 `besselexpand` が `true` の場合、次の変換が実行されます:

- $\text{bessel.j}(1/2)(z) \Rightarrow \sqrt{2/(\pi z)} \sin(z)$
- $\text{bessel.i}(1/2)(z) \Rightarrow \sqrt{2/(\pi z)} \sinh(z)$
- $\text{bessel.k}(1/2)(z) \Rightarrow \sqrt{\pi/(2z)} \exp(-z)$
- $\text{bessel.y}(1/2)(z) \Rightarrow -\sqrt{2/(\pi z)} \cos(z)$

関連する大域変数

<code>iarray</code>	<code>scaled.bessel.i</code> 関数の実行時に生成
<code>yarray</code>	<code>bessel.y</code> 関数の実行時に生成
<code>besselarray</code>	<code>bessel.i</code> 関数の実行時に生成

8.9.3 hypgeo パッケージ

specint 関数

```
specint(exp(-< 変数1 > * < 変数2 >) * < 式 >, < 変数2 >)
```

`specint` 関数は `< 式 >` の `< 変数2 >` に対する Laplace 変換を計算します。

上手く積分が出来なかった場合、内部変数を表示する事がありますが、これは `specint` 関数の虫です。

8.10 楕円函数

8.10.1 Jacobi の楕円函数

Jacobi の楕円函数は第一種楕円積分

$$F_k(x) =_{\text{def}} \int_0^x \frac{1}{\sqrt{(1-t^2)(1-k^2t^2)}} dt$$

の逆函数を用いて定義されます.

$$\text{sn}(u, k) =_{\text{def}} F_k^{-1}(u)$$

ここで $\text{sn}(u, k)$ は k を省略して単に $\text{sn} u$ と一般に表記され, 必要に応じて $\text{sn}(u, k)$ とも表記します.

この sn を基に函数 $\text{cn}, \text{dn}, \text{ns}, \text{sc}, \text{sd}, \text{cd}$ が以下の様に定義されます.

$$\begin{aligned} \text{cn}(u, k) &=_{\text{def}} \sqrt{1 - \text{sn}^2(u, k)} \\ \text{dn}(u, k) &=_{\text{def}} \sqrt{1 - k^2 \text{sn}^2(u, k)} & \text{ns}(u, k) &=_{\text{def}} \frac{1}{\text{sn}(u, k)} \\ \text{nc}(u, k) &=_{\text{def}} \frac{1}{\text{cn}(u, k)} & \text{nd}(u, k) &=_{\text{def}} \frac{1}{\text{dn}(u, k)} \\ \text{sc}(u, k) &=_{\text{def}} \frac{\text{sn}(u, k)}{\text{cn}(u, k)} & \text{cs}(u, k) &=_{\text{def}} \frac{\text{cn}(u, k)}{\text{sn}(u, k)} \\ \text{sd}(u, k) &=_{\text{def}} \frac{\text{sn}(u, k)}{\text{dn}(u, k)} & \text{ds}(u, k) &=_{\text{def}} \frac{\text{dn}(u, k)}{\text{sn}(u, k)} \\ \text{cd}(u, k) &=_{\text{def}} \frac{\text{cn}(u, k)}{\text{dn}(u, k)} & \text{dc}(u, k) &=_{\text{def}} \frac{\text{dn}(u, k)}{\text{cn}(u, k)} \end{aligned}$$

Jacobi の振幅函数 $\text{am}(u, k)$ は $\text{sn}^{-1}(\sin \phi)$ の逆函数として得られる函数です.

— Jacobi の楕円函数 (その 1) —

```
jacobi_sn(<変数1>, <変数2>)
jacobi_cn(<変数1>, <変数2>)
jacobi_dn(<変数1>, <変数2>)
jacobi_am(<変数1>, <変数2>)
inverse_jacobi_sn(<変数1>, <変数2>)
inverse_jacobi_cn(<変数1>, <変数2>)
inverse_jacobi_dn(<変数1>, <変数2>)
```

`jacobi_sn` 函数は Jacobi の sn 函数, `jacobi_cn` 函数は Jacobi の cn 函数, `jacobi_dn` 函数は Jacobi の dn 函数となります. そして, `jacobi_am` 函数が振幅函数になります.

- `diff(jacobi_sn(u,m),u) ⇒ jacobi_cn(u,m)*jacobi_dn(u,m)`
- `diff(jacobi_cn(u,m),u) ⇒ -jacobi_sn(u,m)*jacobi_dn(u,m)`
- `diff(jacobi_dn(u,m),u) ⇒ -m*jacobi_cn(u,m)*jacobi_sn(u,m)`

Jacobi の楕円関数 (その 2)

```
jacobi_ns(⟨変数₁⟩, ⟨変数₂⟩)
jacobi_nc(⟨変数₁⟩, ⟨変数₂⟩)
jacobi_nd(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_ns(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_nc(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_nd(⟨変数₁⟩, ⟨変数₂⟩)
```

$$\text{jacobi\_ns}(u, m) =_{\text{def}} \frac{1}{\text{jacobi\_sn}(u, m)}$$

$$\text{jacobi\_nc}(u, m) =_{\text{def}} \frac{1}{\text{jacobi\_cn}(u, m)}$$

$$\text{jacobi\_nd}(u, m) =_{\text{def}} \frac{1}{\text{jacobi\_dn}(u, m)}$$

Jacobi の楕円関数 (その 3)

```
jacobi_sc(⟨変数₁⟩, ⟨変数₂⟩)
jacobi_sd(⟨変数₁⟩, ⟨変数₂⟩)
jacobi_cs(⟨変数₁⟩, ⟨変数₂⟩)
jacobi_cd(⟨変数₁⟩, ⟨変数₂⟩)
jacobi_ds(⟨変数₁⟩, ⟨変数₂⟩)
jacobi_dc(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_sc(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_sd(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_cs(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_cd(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_ds(⟨変数₁⟩, ⟨変数₂⟩)
inverse_jacobi_dc(⟨変数₁⟩, ⟨変数₂⟩)
```

$$\text{jacobi\_sc}(u, m) =_{\text{def}} \frac{\text{jacobi\_sn}(u, m)}{\text{jacobi\_cn}(u, m)}$$

$$\text{jacobi\_sd}(u, m) =_{\text{def}} \frac{\text{jacobi\_sn}(u, m)}{\text{jacobi\_dn}(u, m)}$$

$$\text{jacobi\_cs}(u, m) =_{\text{def}} \frac{\text{jacobi\_cn}(u, m)}{\text{jacobi\_sn}(u, m)}$$

$$\text{jacobi\_cd}(u, m) =_{\text{def}} \frac{\text{jacobi\_cn}(u, m)}{\text{jacobi\_dn}(u, m)}$$

$$\text{jacobi\_ds}(u, m) =_{\text{def}} \frac{\text{jacobi\_dn}(u, m)}{\text{jacobi\_sn}(u, m)}$$

$$\text{jacobi\_dc}(u, m) =_{\text{def}} \frac{\text{jacobi\_dn}(u, m)}{\text{jacobi\_cn}(u, m)}$$

8.10.2 第一種楕円積分函数

—— 第一種の楕円積分函数 (その 3) ——

```
elliptic_f(⟨変数₁⟩, ⟨変数₂⟩)
elliptic_e(⟨変数₁⟩, ⟨変数₂⟩)
elliptic_kc(⟨変数₁⟩, ⟨変数₂⟩)
elliptic_ec(⟨変数₁⟩, ⟨変数₂⟩)
elliptic_eu(⟨変数₁⟩, ⟨変数₂⟩)
elliptic_pi(⟨変数₁⟩, ⟨変数₂⟩)
```

elliptic_f 函数は次の式で定義された第一種楕円積分函数です

$$\text{elliptic\_f}(\phi, m) =_{\text{def}} \int_0^\phi \frac{1}{\sqrt{1 - m \sin^2 x}} dx$$

elliptic_e 函数は次の式で定義された第二種楕円積分函数です:

$$\text{elliptic\_e}(\phi, m) =_{\text{def}} \int_0^\phi \sqrt{1 - m \sin^2 x} dx$$

elliptic_kc 函数は次の式で定義された第一種楕円積分函数です

$$\text{elliptic\_kc}(m) =_{\text{def}} \int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{1 - m \sin^2 x}} dx$$

elliptic_ec 関数は次の式で定義された第二種楕円積分関数です

$$\text{elliptic\_ec}(m) =_{\text{def}} \int_0^{\frac{\pi}{2}} \sqrt{1 - m \sin^2 x} dx$$

elliptic_eu 関数は次の式で定義された第二種楕円積分関数です:

$$\text{elliptic\_eu}(u, m) =_{\text{def}} \int_0^{\tau} \text{dn}^2(x, m) dx \quad \tau = \text{sn}(u, m)$$

elliptic_pi 関数は次の式で定義された第三種楕円積分関数です:

$$\text{elliptic\_pi}(n, \phi, m) =_{\text{def}} \int_0^{\phi} \frac{1}{(1 - n \sin^2 x) \sqrt{1 - m \sin^2 x}}, x, 0, \phi$$

— make_elliptic_e 関数と make_elliptic_f 関数 —

```
make_elliptic.e(< 式 >)
make_elliptic.f(< 式 >)
```

これらの関数は共に、与式に、inverse_jacobi_sc 関数, inverse_jacobi_cs 関数, inverse_jacobi_nd 関数, inverse_jacobi_dn 関数, inverse_jacobi_sn 関数, inverse_jacobi_cd 関数, inverse_jacobi_dc 関数, inverse_jacobi_ns 関数, inverse_jacobi_nc 関数, inverse_jacobi_ds 関数, inverse_jacobi_sd 関数, inverse_jacobi_cn 関数が含まれている項に対してのみ作用し、それ以外の式はそのままの式を返します。

8.11 dynamics パッケージ

dynamics パッケージを利用する事で、容易に一次元及び二次元の力学系に関連したグラフを描く事が可能になります。この中でも特に plotdf 関数は二次連立微分方程式の解曲線に対話処理を通じて綺麗に描く事が可能な為、力学系のお勉強を行う上で非常に楽しいパッケージとなっています。

dynamics パッケージは Maxima のホームディレクトリ下の share/dynamics に収録されており、complex_dynamics.lisp, dynamics.mac と plotdf.lisp の 3 個のファイルで構成されています。パッケージ全体の Maxima への読込は `load("dynamics");` の様に load 関数を使います。

8.11.1 dynamics.mac に含まれる関数

dynamics.mac は Maxima の処理言語で記述された関数を含んでいます。多くが Maxima を plot2d 関数を流用する関数です。その為、オプションが指定可能な関数では、plot2d 関数のオプションがそのまま使えます。

dynamic.mac

```

evolution(⟨ 一変数関数 ⟩, ⟨ 初期値 ⟩, ⟨ 正整数値 ⟩)
evolution(⟨ 一変数関数 ⟩, ⟨ 初期値 ⟩, ⟨ 初期値 ⟩, ⟨ 正整数値 ⟩, [⟨ plot_options ⟩])
evolution2d(⟨ 一変数関数 ⟩, ⟨ 初期値 ⟩, ⟨ 正整数値 ⟩)
evolution2d(⟨ 一変数関数 ⟩, ⟨ 初期値 ⟩, ⟨ 初期値 ⟩, ⟨ 正整数値 ⟩, [⟨ plot_options ⟩])
staircase(⟨ 一変数関数 ⟩, ⟨ 初期値 ⟩, ⟨ 初期値 ⟩, ⟨ 正整数値 ⟩)
staircase(⟨ 一変数関数 ⟩, ⟨ 初期値 ⟩, ⟨ 初期値 ⟩, ⟨ 正整数値 ⟩, [⟨ plot_options ⟩])
chaosgame([⟨ x1 ⟩, ⟨ y1 ⟩], ⋯, [⟨ xn ⟩, ⟨ yn ⟩], [⟨ x0 ⟩, ⟨ y0 ⟩], β, ⟨ 正整数値 ⟩)
chaosgame([⟨ x1 ⟩, ⟨ y1 ⟩], ⋯, [⟨ xn ⟩, ⟨ yn ⟩], [⟨ x0 ⟩, ⟨ y0 ⟩], β, ⟨ 正整数値 ⟩,
[⟨ plot_options ⟩])
ifs([⟨ r1 ⟩, ⋯, ⟨ rm ⟩], [⟨ A1 ⟩, ⋯, ⟨ Am ⟩], [[⟨ x1 ⟩, ⟨ y1 ⟩], ⋯, [⟨ xm ⟩, ⟨ ym ⟩]], [⟨ x0 ⟩, ⟨ y0 ⟩],
β, ⟨ 正整数値 ⟩)
ifs([⟨ r1 ⟩, ⋯, ⟨ rm ⟩], [⟨ A1 ⟩, ⋯, ⟨ Am ⟩], [[⟨ x1 ⟩, ⟨ y1 ⟩], ⋯, [⟨ xm ⟩, ⟨ ym ⟩]], [⟨ x0 ⟩, ⟨ y0 ⟩],
β, ⟨ 正整数値 ⟩, [⟨ plot_options ⟩])
orbits(f, y0, n1, n2, domain)
orbits(f, y0, n1, n2, domain, [options])
rk(⟨ 連立常微分方程式 ⟩, state, initial, domain)
```

evolution 関数は一次元写像の軌道を描く関数です. 具体的には, 実一変数関数 f に対し, 初期値を x_0 とした場合, x_0 に関数 f を 0 から n 回作用させる事で得られる点列 $(i, f^i(x_0))_{i=0 \cdots n}$ を描きます.

関数 evolution の第一引数〈関数〉は一変数の関数式になります. この関数は直接式を引渡しても, Maxima 上で定義した式を含んでも構いません.

次の引数〈初期値〉は関数変数の初期値となります. この初期値に第一引数の関数作用させる事で得られる軌道を描く事になりますが, 〈 n 〉で, 漸近過程のリストの長さを指定します. 最後に, オプションとして, plot2d 関数の任意のオプションが指定可能となっています.

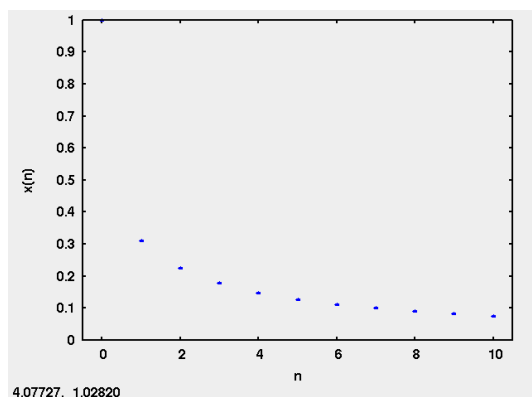


図 8.4: evolution(exp(-x)*sin(x),1,10) の結果

evolution2d 関数は二次元の離散的力学系の evolution を二次元グラフで表示する関数です. evolution 関数では点列 $(i, f^i(x_0))$ を描きましたが, evolution2d の場合, 点 $(u_{n+1}, v_{n+1}) = (F(u_n, v_n), G(u_n, v_n))$ を描きます.

引数の〈関数₁〉と〈関数₂〉は各々, 〈変数₁〉と〈変数₂〉の関数で, その初期値は各々〈初期値₁〉と〈初期値₂〉とします. 〈 n 〉はステップ数になります. ここで, option としては plot2d 関数の任意のオプションが指定可能となっています.

staircase 関数は一次元写像の staircase 図を描く関数です. 引数の〈関数〉は一変数関数の式であり, 〈初期値〉はその変数の初期値, 〈 n 〉はステップ数になります. ここで, option としては plot2d 関数の任意のオプションが指定可能となっています.

chaosgame 関数はカオス ゲームを実行する為の関数です.

ifs 関数は Barnley の IFS(Iterated Function System) 法を用いて二次元のフラクタルを生成する関数です.

orbits 関数 は一階離散力学系に対し, 軌道図を描く関数です.

rk 関数は四次の Runge-Kutta 法を用いて常微分方程式を解く関数です.

8.11.2 complex_dynamics.lisp ファイルに含まれる関数

complex_dynamics.lisp には一次元の複素力学系で有名な Mandelbrot 集合や Julia 集合を描く関数が定義されています. これらの集合の計算では計算時間の短縮の為に, Maxima の処理言語ではなく, 直接 Lisp で関数を定義しています. 猶, これらの関数は基本的に画像ファイルを出力し, 画面への直接出力は行いません.

dynamic.mac

```
mandelbrot(< オプション >)  
julia(<x>, <y>, < オプション >)
```

mandelbrot 関数は Mandelbrot 集合を描く関数で, 図 8.5 に示すグラフを xpm 形式のファイルで出力する関数です. この関数の引数は不要です.

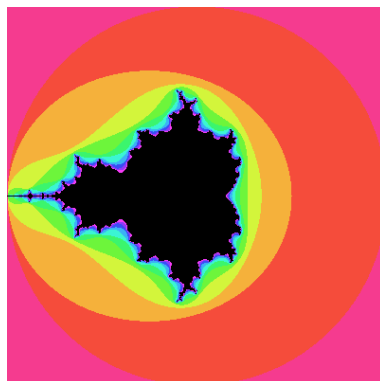


図 8.5: mandelbrot() を実行した結果

julia 関数 も mandelbrot 関数の様に Julia 集合を描く関数です. 但し, mandelbrot とは違い, 引数として原点を最低でも指定しなければなりません.

先ず, julia 関数の第一と第二引数で複素平面上の点を表現します. オプションとしては以下のものが指定可能です.

size
 levels
 huerage
 hue
 saturation
 value
 color
 center
 radius
 filename julia 画像ファイル名 (後に.xpm が付く)
 猶, これらの関数の弱点は計算が非常に遅い事です.

8.11.3 plotdf 関数

plotdf 関数は二次元の力学系の解曲線等を対話的に描く事が可能な関数です. 猶, 対話処理の為に tcl/tk を利用しています.

この plotdf 関数で描画出来る常微分方程式は

$$\frac{dy}{dx} = f(x, y)$$

$$\frac{dx}{dt} = f(x, y) \quad \frac{dy}{dt} = g(x, y)$$

の何れかになります.

この関数は, plotdf.lisp で定義されており, この関数を利用する為に予め `load("plotdf");` で予め plotdf 関数の読込を実行しておく必要があります.

plotdf 関数

plotdf(< 微分方程式 >, < オプション >)

この関数のフロントエンドに指定可能な数値やスライダーの設定を直接 plotdf 関数から直接オプションとして指定する事も可能です. ここで, オプションの指定方法は plot2d 関数と同様に, `[[ 表象 ], < 値 >]` の書式で行います.

— オプション 1 —

表象	初期値	概要
xradius	10	描画する X 軸方向の範囲
yradius	10	描画する Y 軸方向の範囲
xcenter	0	X 座標の中心点
ycenter	0	Y 座標の中心点
tinitial		初期値の時刻を指定
tstep		解曲線の時刻の刻幅. 軌道上の点は nsteps で指定

tstep の値は Plotdf で表示される軌道の描画には直接影響しませんが, versus_t で表示される時刻歴の解曲線の表示には直接影響します.

— オプション 2 —

height		縦の大きさ
width		幅の大きさ
nsteps	100	描画する軌道上の点数
versus_t		時刻歴による軌道の表示の有無
bbox		

versus_t は plotdf 関数のウィンドウメニューの中にも含まれており, plotdf 関数から直接指定する場合には整数値を指定します. この versus_t は trajectory_at で指定した点を通る軌道に対し, 横軸を時間とした X と Y のグラフを別ウィンドウへの出力を指定ものです. その為, ウィンドウメニューから指定する場合には少なくとも一つの軌道が予め生成されていなければなりません.

オプション 3

xfun		
parameters		
trajectory_at		軌道の描画で軌道上の点を指定
direction	both	both, forward と backward が指定可能
linewidth	1.0	描画する軌道の太さを指定
linecolors	green, black, brown, gray	線の色彩を指定
adamsMoulton	red	軌道を描画する場合の色彩を指定
sliders		

direction はある一点をマウスで指定した場合に、その点を始点や終点、或いは通過する軌道を描きます。

trajectory_at には XY 平面上の点を指定します。plotdf 関数は、この指定した点を通る軌道を描画しますが、その方向は、direction で指定した方向になります。猶、plotdf 関数のウィンドウメニュー Config から呼出される変数設定ウィンドウで軌道上の点 (x, y) を指定する場合は、x y の様に空行を入れます。

8.11.4 幾つかの例題

dynamics パッケージを用いた幾つかの例題を示しておきましょう。

Malthus の人口モデル

産業革命時代のイギリスで Malthus は次の人口増加モデルを提唱しました。

- 人口は等比数列的に増加する
- 食料供給は等差数列的に増加する

最初の人口増加の方程式は、時刻 t の人口を N とすると、或る正定数 a を用いて $\frac{dN}{dt} = aN$ と表現される事を意味します。それに対して食料供給は、時刻 t の食料を F とすると、或る正定数 F_0 を使って $\frac{dF}{dt} = F_0$ となる事を意味します。

ここで Malthus の人口モデルを evolve 関数を使って描いてみましょう。例えば、定数 a を 1.5 にし、初期値の人口を 3.0e+7(三千万人、江戸時代末期のおおよその人口)として、第 3 世代迄の人口を描いてみましょう。1.5 にした理由ですが、単純に、一つの

家庭から成人する子供が3人程度と仮定しただけです. 因に Malthus は3人の子持ちだった様です.

```
evolution(1.5*x,3.e+7,3)
```

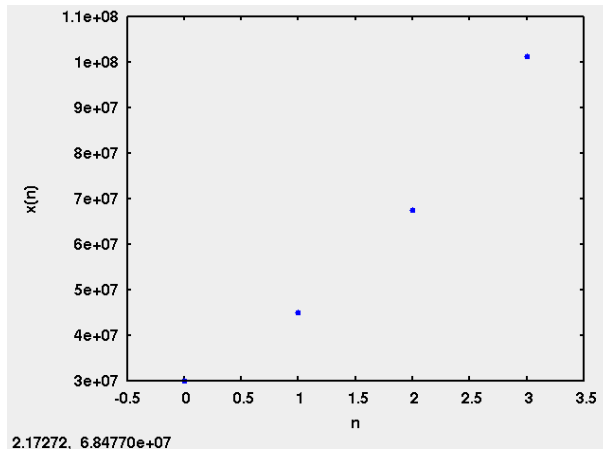


図 8.6: `evolution(1.5*x,3.e+7,3)` のグラフ

図 8.6 のグラフを見ると, 最終的には, 1.1 億人近くになりましたね. 大体一世代で 30 年なので, 江戸末期から昭和末期の人口と言えますが, 現在の人口と比較してそんなに外れていない様ですね. 今度は江戸末期から 10 世代後 (大体 300 年後, 22 世紀) はどうなるでしょうか?

```
evolution(1.5*x,3.e+7,10)
```

図 8.7 を見ると, 今度は 18 億人を突破してしまいました! 食料供給から考えると, 通常の方法では到底間に合わない事は明確でしょう.

因に, `desolve` 関数で, この方程式を解いてみましょう.

```
(%i54) atvalue(x(t),t=0,3.0e+7);
(%o54) 3.0E+7
(%i55) desolve(['diff(x(t),t)=3/2*x(t)],[x(t)]);
```

'rat' replaced -3.0E+7 by -30000000/1 = -3.0E+7

$$\frac{3}{2}t$$

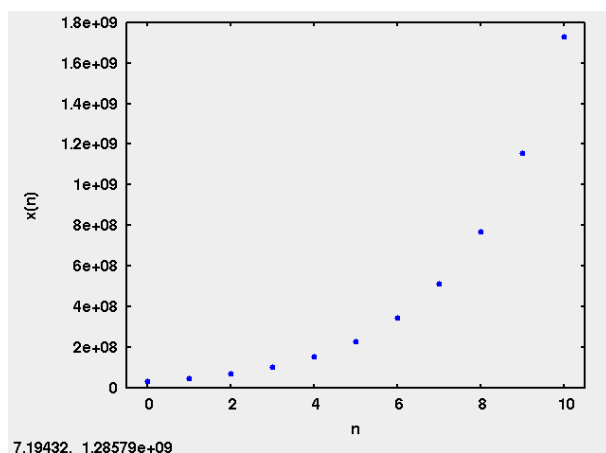


図 8.7: $\text{evolution}(1.5*x, 3.e+7, 10)$ のグラフ

(%o55) $x(t) = 30000000 \%e$

この様に Malthus の人工モデルでは, 人口の増加は指数関数になります. では, 実際にこの様な人口の爆発が生じていない理由は何故でしょうか? Malthus は戦争, 飢饉や貧困といった悪があるお陰だと主張しています. 但し, これらの悪が無くなれば, 人口を抑制するこれらの必要悪が無くなってしまう訳で, そんな場合には Malthus は禁欲的晩婚を主張しています. 無論, それで誰もが納得する訳ではありません. 19 世紀は社会的ダーウィニズムが主張される富国強兵, 弱肉強食の時代の為, この様な禁欲的晩婚による人口の抑制の結果, 国力が相対的に落ちてしまえば途端に困る訳です. そこで帝国主義者は逆に植民地獲得によって人口問題を解決する事を主張しています⁵. この辺は 20 世紀には生存圏 (Lebensraum) といった形でも見え隠れしたりします.

マメゾウムシの個体数モデル (ロジスティック方程式)

先程の Malthus の人口モデルは指数関数的な人口増大になっていました. しかし実際はそうではありません. 例えば, 現在の日本では少子化により人口が減少に転じている程です.

⁵ 出来る事なら惑星も併合したい-Cecil John Rhodes

ここで動物の世界, 特に虫に関しては, 食料やその他の様々な問題によって, 個体数が或る一定の値に飽和する現象が見られます.

その様な現象を表現するモデルの一つとしてロジスティック (兵站) 方程式と呼ばれる形の微分方程式があります.

$$\frac{du}{dt} = (A - ku)u$$

ここで, u が時刻 t に於ける個体数で, A と k は正実数になります. この方程式は時間経過に伴って, 個体数がある一定の飽和値に近づいてゆく現象を表現したものです. 因に, A/k が安定点となり, $u < A/k$ や $u > A/k$ の場合は A/k に漸近します.

この様なモデルとして面白いもので, マメゾウムシの個体数でも知られている方程式を紹介しましょう. マメゾウムシは子孫を残すと親が全て死んでしまう為, 世代交代が明確になります. その為, 上述の微分方程式を離散化したものの方がより良く実際と一致するそうです. 猶, 方程式は n 世代の個体数を u_n とした時に,

$$u_{n+1} = u_n + (A - ku_n)u_n$$

になります.

この方程式は非常に怪しい振舞をする事が知られています.

これはパラメータによって個体数の振動が発生する事です. この例はカオスの特徴を説明するものとして広く知られている現象です.

被食者と捕食者のモデル

こちらは, 捕食者 (例えば狼) と被食者 (例えば鹿) の個体数に関する方程式で有名なものに Volterra の微分方程式があります. この方程式は以下の連立微分方程式です.

$$\begin{aligned}\frac{dx}{dt} &= (A - k_1y)x \\ \frac{dy}{dt} &= -(B - k_2x)y\end{aligned}$$

ここで被食者の個体数が x , 捕食者の個体数が y となっています. この式では, 被食者が増加すると, それを食べる捕食者も増加しますが, 捕食者が増え過ぎると今度は被食者が減ってしまうので, 結果として捕食者が減るといった事を表現したモデルです. この方程式は一つの安定な解と周期解を持ちます. 安定な解は $\frac{dy}{dt}$ と $\frac{dx}{dt}$ の双方が 0 になる点で, この場合は, $(\frac{A}{k_1}, \frac{B}{k_2})$ が安定点, それ以外は周期解上の点になります.

では, このモデルを Maxima で表現してみましょう.

```
(%i34)load("plotdf")$
(%i35) plotdf([(1-2*y)*x,-(1-x)*y],[x,y],
[xcenter,2],[ycenter,2],[xradius,2],[yradius,2],
[nsteps,400],[trajectory_at,1,1],[versus_t,100])$
```

ここでは、軌道を描く為に、trajectory_at で軌道上の点を指定しています。又、軌道上の点数は nsteps で 400 点に指定しています。versus_t については後で詳細を述べますが、こちらは適当な整数値で構いませんが、実数値はエラーになります。

これを実行すると、二つのウィンドウが出現します。一つは図 8.8 に示す plotdf 本来のウィンドウで、Openmath を用いたものです。

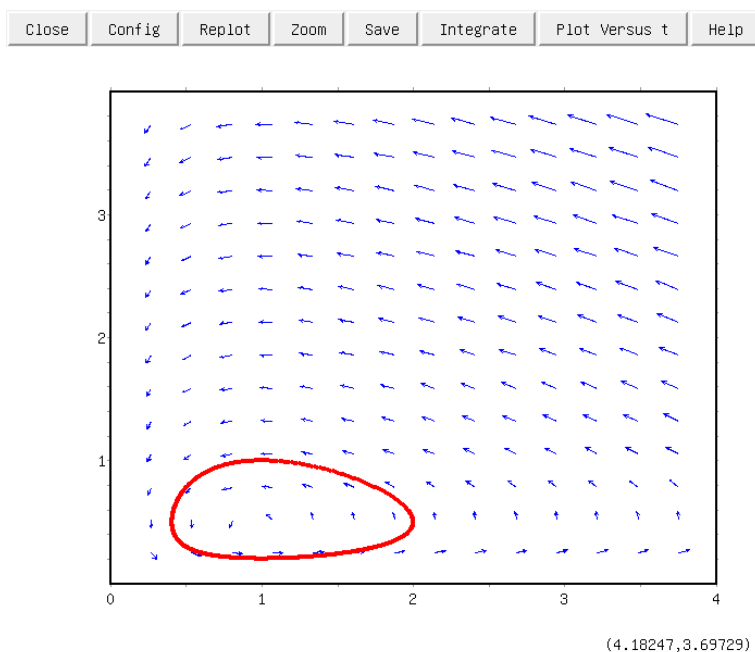


図 8.8: plotdf のグラフ (その 1)

もう一つは `versus_t` を指定した事で出現した X と Y の時刻歴での解曲線の表示を行う図 8.9 に示すウィンドウで, その名を X and Y versus t というものです. こちらは `plotdf` のウィンドウで軌道を新規に表示させると自動的に更新されます.

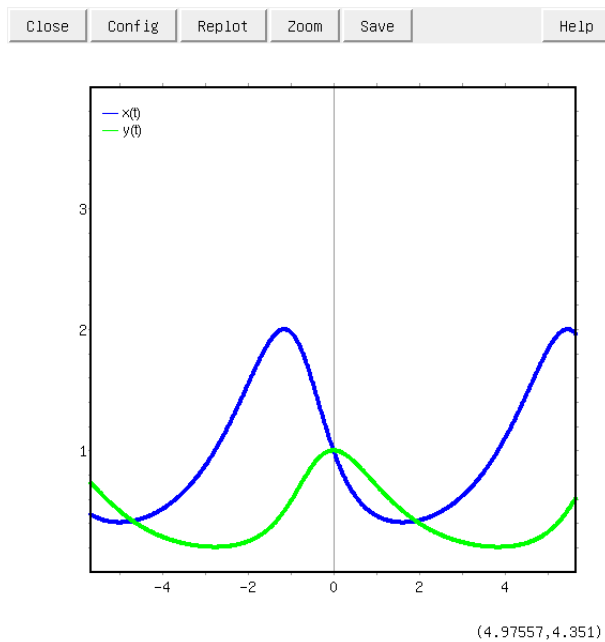


図 8.9: `plotdf` のグラフ (その 2)

図 8.8 で, 解曲線は赤く表示されている様に見えますが, これは解曲線上の点の色です. 実際の解曲線の色は緑です. 例えば, `Plotdf` 上で $(3,3)$ 付近をマウスで押してみてください. すると新規に軌道が表示され, その軌道は緑色, その上に赤い点が載っている事が判りますね.

この色を変更する事も可能です. これは, `Plotdf` の `Config` メニューを押して出てくる `plot setup` ウィンドウで `adamsMoulton` の値を変更すれば良いのです.

第9章 Maximaのシステム関連の函数

この章では Maxima のシステム関連の函数と虫取りに関連する函数について述べます。Maxima は巨大なシステムであると同時に、非常に古いシステムである為、旧弊な事も紛れ込んだ雑多な内容です。

9.1 ディレクトリに関連する大域変数

Maxima ではファイルの読込に関連して、幾つかのディレクトリ/フォルダに関連する大域変数を持っています。

ディレクトリに関連する大域変数

大域変数	概要
maxima_tempdir	
maxima_userdir	
file_search_lisp	
file_search_maxima	
file_search_demo	
file_search_usage	
file_search_tests	

猶、これらの大域変数は利用するシステムや環境によって値が事なる為に、その初期値はここでは表記しません。

これらの大域変数は LISP の文字列型のデータで構成された Maxima のリストです。従って、この文字列を操作する為には、Maxima-5.13.0 以前では直接 LISP 側から指定するか、stringproc パッケージに含まれる函数を用いる必要があります。但し、Maxima-5.14.0 では Maxima の文字列は LISP の文字列型となった為、直接操作が可能となっています。

先ず、maxima_tempdir は Maxima がグラフ表示等で生成するファイルの出力先が指定される大域変数です。

次の、maxima_userdir は、Maxima が利用者定義の Maxima や LISP のパッケージファイルを探す場所を指定する大域変数です。猶、利用者全てが利用可能なパッケージは大域変数 file_search_maxima や大域変数 file_search_lisp 等で置き場所を指定する必要があります。又、maxima_userdir を変更しても、大域変数 file_search_maxima 等の大域変数は影響を受けない事に注意が必要です。

他の大域変数に関しては、§7.4.2 を参照して下さい。

9.2 システムの初期化

9.2.1 maxima-init.mac ファイル

Maxima は起動時にカレントディレクトリ上にあるファイル maxima-linit.mac を自動的に読み込みます. この maxima-init.mac ファイルを上手く利用することで, Maxima の起動時の環境が容易に変更出来ます.

maxima-init.mac には Maxima の関数や大域変数の設定が通常の入力と同様の書式で記述可能です. 更に, setup_autoload 関数や load 関数を用いて, 必要なパッケージの読込も自動的に行えます.

以下に非常に簡単な例を示します.

```
/*--MAXIMA--*/
showtime:all;
put(surfg, d_chain_bisection,root_finder)$
put(surfg, 0.0000000001,epsilon)$
put(surfg, 20000,iterations)$
put(surfg, 500,width)$
put(surfg, 500,height)$
put(surf, yes,do_background)$
put(surf, 5,background_red)$
put(surf, 5,background_green)$
put(surf, 5,background_blue)$
put(surf, 0.14,rot_x)$
put(surf, -0.3,rot_y)$
setup_autoload("surfplot.mc",surfplot)$
load("fox.mc")$
```

Maxima が読込むファイルで, 註釈は C と同様に /* */ の中に記述します. この例では頭に註釈行として /*--MAXIMA--*/ と置いて Maxima 言語のファイルである事を示しています. その後に, 大域変数の設定, 属性の設定等を行っています.

ここで, setup_autoload 関数や load 関数を用いたファイルの読込について簡単に解説しておきましょう. この setup_autoload 関数の構文を以下に示しておきます.

— setup_autoload 関数 —

setup_autoload (< ファイル >, < 関数₁>, ..., < 関数_n>)

この `setup_autoload` 関数は、指定した関数が呼出された時点で関数が未定義の場合、指定したファイルの読み込みを実行します。このファイルの読み込みでは、関数名に付与された `autoload` 属性でファイルの自動読み込みを判断しています。ここで、`setup_autoload` 関数は引数として配列関数を扱えない事に注意して下さい。

この例では、ファイル `surfplot.mc` に含まれている `surfplot` 関数が呼出された時に、未定義であれば、ファイル `surfplot.mc` を読み込む設定となっています。

`load` 関数の場合は予め指定されたファイルの内容を全て Maxima に読み込みます。この場合、ファイルの定義、未定義とは無関係に読み込みを実行します。

猶、読み込むパッケージが大域変数 `file_search_maxima` に登録されたディレクトリと修飾子を持っているのであれば、`load("fox.mc")` の様に二重引用符と修飾子を外して、`load(fox)` とする事が可能です。この詳細に関しては、§7.4.2 を参照して下さい。

ここで `maxima-init.mac` の効果的な置場所は利用環境によって多少異なります。UNIX 系の OS で、命令の直接入力での Maxima のフロントエンドを起動する場合、カレントディレクトリ上に `maxima-init.mac` があれば内容が反映されますが、昨今の様に、デスクトップ環境から Maxima のフロントエンドを立上げる場合には話がやや異なります。

UNIX 環境であれば、環境変数 `$HOME` で指定されたディレクトリ上に `maxima-init.mac` を置くと良いでしょう。但し、MS-Windows の場合は厄介です。基本的に MS-Windows 環境の場合、利用するフロントエンドの `exe` ファイルが収納されているフォルダに `maxima-init.mac` を入れておくのが無難な様です。この事は §17.4.4 を参照して下さい。

又、`maxima-init.mac` で指定したファイルが存在しない場合、読み込みファイルに重大な間違いがある場合、Maxima が起動しない事があるので注意して下さい。

9.2.2 `reset` 関数や `kill` 関数による初期化

Maxima では、`reset` 関数を用いて Maxima を初期状態に戻す事や `kill` 関数を用いて、処理で蓄えたラベルや大域変数等を破棄する事が可能です。

— `reset` 関数と `reset_verbosely` 関数 —

```
reset()  
reset_verbosely()
```

`reset` 関数と `reset_verbosely` 関数は内部的には同じ関数 (`reset-do-the-work`) 関数を用いています。共に引数が不要ですが、`reset_verbosely` は初期化の処理内容が具体的

に表示されます. これらの関数は Maxima の内部変数*variable-initial-values* に登録された変数と値のリストを用いて,Maxima を初期状態に戻す関数です. 因に, 大域変数等の設定で内部関数 defmvar を用いた場合, その大域変数と値のリストが内部変数*variable-initial-values*に登録されます.

猶, これらの関数によって内部変数*variable-initial-values*に保存された値に変数に束縛された値が戻るだけで, 何らのデータが削除される訳ではありません. 従って, 利用者が定義した変数とその値, 以前のラベルに保存されている値は残りますが, ラベルの方はカウンターが 1 に戻されている為, 処理を進めるに従って上書きされてしまいます.

```
(%i2) display2d:false;
(%o2) false
(%i3) nolabels:true;
(%o3) true
(%i4) reset_verbosely();
reset: bind lispdisp to false
reset: bind display2d to true
reset: bind linenum to 1
reset: bind % to %
reset: bind __ to __
reset: bind _ to _
(%o1) [lispdisp, display2d, linenum, %, __, _]
(%i2)%i3;
(%o2) nolabels : true
```

この例では,reset_verbosely 関数による再設定の様子を示しています. 大域変数の値が元に戻されていますが, ラベルは残っている事が判ります.

kill 関数は reset 関数と reset_verbosely 関数とは異なり, 指定された Maxima の対照の削除を実行します.

kill 関数

```
kill(< 引数1>, < 引数2>, ...)  
kill(< 整数 >)  
kill([< 整数1>, < 整数2>])  
kill(all)  
kill(allbut(< 引数1>, < 引数2>, ...))
```

kill 関数は reset 関数と reset_verbosely 関数とは異なり、必ず引数を取ります。kill 関数は引数によって削除する対象が異なります。

- 引数に変数、配列、関数の場合、指定された対象は、その属性を含めて全てが消去されます。
- 引数 が大域変数 infolists に含まれる大域変数の場合、その大域変数に含まれる対象とその属性が全てが削除されます。
- 引数が tellrats の場合、内部変数 tellratslist が nil に戻されます。
- 引数が ratweights の場合、内部変数*ratweights が nil、大域変数 ratweights が空リストに設定されます。
- 引数が feature の場合、大域変数 features に登録された対象で、内部変数 feature の属さないものが削除されます。
- 引数が all の場合、が削除されます。
- 引数が allbut(...) の場合、allbut で指定した対象を除外して、引数が all の場合と同様に削除が行われます。

猶、変数のみを指定して式を削除しても、ラベルの削除を行わない限り、メモリの全体的な占有領域の解放が行われない事に注意が必要です。例えば、大きな式が%i10で変数 x に割当てた場合、占有された保存領域を解放する為には、kill(x) と kill(%o10) の両方を実行する必要があります。

麻矢、kill 関数は与えられた引数から全ての属性を削除します。従って、kill(values) は大域変数 values に割当てられたリストの全ての対象に関連する属性を削除しますが、remvalue,remfunction,remarray,remrule といった関数は指定した属性のみを削除します。これらの関数群はリスト名か指定した引数が存在しなければ false を出力しますが、これに対して、kill 関数は指定した対象が存在しない場合でも、常に done を出力します。

collapse 関数

```
collapse (< 式 >)
collapse ([[< 式1>, ..., < 式n>]])
collapse (listarray(' < 配列 >))
```

全ての共通部分式を同じセルを用いる事で、式のメモリの使用量を削減させる関数です。この collapse 関数は optimize 関数でも用いられています。又、save 関数を用いて保存したファイルは、Maxima の内部表現をそのまま含む式の為、通常の入力式と比較して一般的に大きなファイルとなっています。その為、ファイルの読込んだ後に、collapse 関数を用いると良いでしょう。更に、配列 a に対し、collapse(listarray('a)) とする事で、配列の無駄な成分を潰す事も出来ます。

9.3 処理の中断

9.3.1 中断の制御文字

Maxima の計算が異常に長い場合、間違って計算を実行させた場合、Maxima を一旦中断する必要があります。Maxima の計算を中断したければ、通常は制御文字 ^c (Ctrl+C) を使います。Maxima は ^z (Ctrl+Z) が入力されても中断しますが、この場合は Maxima を出て、UNIX の shell レベルに戻るので、通常は Ctrl+C を使います。

```
(%i11) factor(2137498127943870982374);
Maxima encountered a Lisp error:
```

```
EXT:GC: User break
```

```
Automatically continuing.
```

```
To reenale the Lisp debugger set *debugger-hook* to nil.
```

```
(%i12)
```

この例では、Ctrl+C で因数分解を中断させています。

9.3.2 break 関数による中断

Maxima では break 関数を用いてプログラムを意図的に中断させる事も可能です。

break 関数

```
break(< 式1>, ..., < 式n>)
```

break 関数の返却値は < 式_n> の処理結果となります。break 関数は引数の全ての式を処理した後に Maxima-break に入ります。Maxima-break ではプロンプトとして大域変数 prompt に設定した文字列が表示されます。この Maxima-break では通常の関数が利用可能ですが、ev 関数の省略型や、ラベルを用いた処理が行えません。Maxima-break から抜ける為には `exit;` と入力します。

```
(%i17) break(integrate(sin(x)*x,x),factor(x^2-1));
sin(x) - x cos(x) (x - 1) (x + 1)

Entering a Maxima break point. Type exit; to resume
_diff(sin(x)-x*cos(x),x);
x sin(x)
_expand((x-1)*(x+1));
2
x - 1
_exit;
(%o17) (x - 1) (x + 1)
(%i18)
```

この例では、 $\int x \sin(x) dx$ と $x^2 - 1$ の因子分解を実行した後に Maxima-break に入っています。Maxima-break で出力されるプロンプトはデフォルトで `_` となっています。Maxima-break から抜ける為には、`exit;` と入力します。すると、break 関数は最後に処理した式の結果を返却値として返します。

9.4 ラベルの参照

Maxima は入力を入力ラベル%i に、計算結果を出力ラベル%o に各々保存しています。これらの値の参照は、入力ラベルの場合は%i< ラベル番号 >で、出力ラベルの場合は%o< ラベル番号 >で行います。

更に、最新の結果は%で参照出来ます。これに対し、最新の入力は_で参照します。

これは、大域変数 nlabels が false の場合、入力式と出力式がラベルに束縛され、これらのラベル名は大域変数 labels に割当てられたリストに自動的に追加される仕組み

なっています。入出力の参照は、この仕組みを利用したものです。大域変数 `nolabels` が `true` であれば、この束縛と大域変数 `labels` へのラベルの自動追加が実行されない為、ラベルの参照が行えなくなります。

出力ラベル `%o` に割当てられた値を参照する函数に `%th` があります。この `%th` は `%th(6)` の様に用い、6 個前の結果を参照します。更に、`%i7` や `%o8` とすれば、ラベル (`%i7`) の付いた行に入力した式や、ラベル (`%o8`) に表示された結果を参照する事が出来ます。但し、`i` は、その様な使い方が出来ません。

入出力ラベルは `kill(labels)` で全て削除する事が可能です。これは大域変数 `labels` に登録されたラベル全てを削除する事になり、これを実行すると、入力・出力ラベルに割当てられた全ての値が消去され、各ラベルのカウントも 1 に戻されます、その為、入力ラベルは (`%i1`)、出力ラベルも (`%o1`) から開始する事になります。

```
(%i101) 1+2;
(%o101)                                     3
(%i102) resultant(x-t,y-t^2,t);
                                           2
(%o102)                                y - x
(%i103) algsys([2*x+3*y=1],[x,y]);
                                           2 %r2 - 1
(%o103) [[x = %r2, y = - -----]]
                                           3
(%i104) %;
                                           2 %r2 - 1
(%o104) [[x = %r2, y = - -----]]
                                           3
(%i105) _;
(%o105)                                     %
(%i106) %i101;
(%o106)                                     3
(%i107) %o101;
(%o107)                                     3
(%i108) %i102;
                                           2
(%o108) resultant(x - t, y - t , t)
(%i109) kill(labels);
```

```
(%o0)
done
(%i1)
```

この例では、様々な処理を行い、それらを%や_で確認し、最後に `kill(labels)` でラベル(%i や%o)の内容を全て消去しています。ラベルの消去を行った為に、`kill(labels)`を入力した(%i109)から、(%o1)を経て(%i1)に初期化されている事に注目して下さい。

この他にも、ラベルの処理に関連する大域変数を Maxima は持っています。

—— ラベルに関連する大域変数 ——

変数名	初期値	概要
<code>labels</code>	<code>[]</code>	入出力等のラベル名が登録されたリスト
<code>nolabels</code>	<code>false</code>	入力値と計算結果をラベルへの束縛を制御
<code>%</code>		最新の処理結果
<code>%%</code>		maxima-break の間に処理された最新の値
<code>inchar</code>	<code>%i</code>	入力ラベルで用いる文字を指定
<code>outchar</code>	<code>%o</code>	出力ラベルで用いる文字を指定
<code>linechar</code>	<code>%t</code>	中間表示の際に用いられる文字を指定
<code>linenum</code>		入力番号が設定されている
<code>prompt</code>	<code>-</code>	demo 関数のプロンプトを指定

大域変数 `labels` には、入出力ラベル等のラベルで構成されたリストが割当てられています。この大域変数 `labels` に割当てられたリストに登録されたラベルの照合は、ラベル名を入力する事で行えます。

```
(%i9) integrate(sin(x),x);
(%o9)
- cos(x)
(%i10) labels;
(%o10) [%i10, %o9, %i9, %o8, %i8, %o7, %i7, %o6, %i6, %o5, %i5, %o4, %i4, %o3,
%i3, %o2, %i2, %o1, %i1]
(%i11) [%i9,%o9];
(%o11)
[integrate(sin(x), x), - cos(x)]
```

大域変数 `nolabels` はラベルへの入出力式の束縛と大域変数 `labels` へラベルの自動登録といった操作を制御する大域変数です。大域変数 `nolabels` が `true` の場合、入力値と計算結果はラベルに束縛されません。即ち、大域変数 `labels` に割当てられたリストに入

出力ラベルの追加が行われなくなり,%やラベル名の直接入力等で結果や入力の参照が出来なくなります.

大域変数%%は大域変数%と同じ意味を持ちます. 但し, 大域変数%%は大域変数%が使えない block 文内部のみで利用可能です.

```
(%i1) block(integrate(x^2,x,0,3),%*3);
(%o1)
3 %
(%i2) block(integrate(x^2,x,0,3),%*3);
(%o2)
9 %
(%i3) block(integrate(x^2,x,0,3),%%*3);
(%o3)
27
(%i4) %%;
(%o4)
%%
(%i5) block(integrate(x^2,x),print(%),%%*3,print(%),
diff(%%,x),print(%));
3
x
--
3
3
x
2
3 x
2
(%o5)
3 x
```

この例は大域変数%と大域変数%%の違いを説明するものです. 最初に block 文内部の積分を実行して大域変数%に保存された式を 3 倍にしますが,block 文内部の大域変数%は block 文を実行する直前の結果が設定されているので,block 文内部で書換えられる事はありません. これに対し, 大域変数%%は block 文内部の局所変数で, その値も block 文内部のみで更新されます. その為,block 文が終了すると, 値は消去されます.

大域変数 inchar は Maxima の入力行ラベルで用いられる文字です. デフォルトでは入力行ラベルは (%i1) の様に大域変数 inchar で指定した%i の後に番号が続きます.

大域変数 outchar は出力ラベルで用いられる文字です. 大域変数 inchar と同じ使い方をします.

大域変数 linechar には中間表示に於ける式の前に置かれる文字を指定します.

大域変数 `linenum` にはその時点での入力行番号が割当てられています.

```
(%i17) linenum;
```

```
(%o17) 17
```

大域変数 `prompt` は `demo` 関数のプロンプト記号を指定する大域変数です. この大域変数の値は `playback(slow)` や `break` 関数で Maxima-break に入った時点に表示されます.

```
(%i1) integrate(x^2-1,x);
```

```
3
x  - 3 x
-----
3
```

```
(%o1)
```

```
(%i2) prompt:"(;;)";
```

```
(%o2) (;;)
```

```
(%i3) playback(slow);
```

```
(%i1) integrate(%,x);
```

```
(;;)
```

```
3
x  - 3 x
-----
3
```

```
(%o1)
```

```
(;;)
```

```
(%i2) prompt:"(\;_\;)";
```

```
(;\;_)
```

```
(%o2) (;\;_)
```

```
(%o3) done
```

```
(%i4) break(x-1);
```

```
x - 1
```

Entering a Maxima break point. Type exit; to resume


```
(;_)exit;
(%o4)                x - 1
```

この例では大域変数 `prompt` に Maxima の文字列 `(;_)` を指定しています. `playback(slow)` を実行すると, 過去の入力と出力を一纏めにして出力し, 大域変数 `prompt` を出力して Enter キーの入力待ちとなります.

Maxima にはラベル処理の関数として, `%th` 関数と `labels` 関数を持っています. これらの関数は共に大域変数 `labels` を用いて処理を行う関数です.

ラベル処理に関連する関数

```
%th(< 正整数 >)
labels(< シンボル >)
```

`%th` 関数は大域変数 `labels` に登録された出力ラベルに対し, `< 正整数 >` 番目の出力ラベルに束縛された値を返します. 大域変数 `nolabels` の初期値が `false` なので, 通常は, `< 正整数 >` 番前の計算結果になります. 実際, `%th<i>` を含む式の入力ラベルが `%<j>` であれば, `%th<i>` は `%i<j-i>` の結果, 即ち, `%o<j-i>` の値になります.

この `%th` は batch ファイルでは非常に便利です. これは `%o` ラベルの値が batch ファイルをどの時点で処理するかで異なるのに対し, `%th` はその関数を実行する時点を中心として指定した整数程前の結果を返す事になるからです.

`labels` 関数は, シンボルを引数として取り, 大域変数 `labels` に割当てられたリストを構成する成分と引数を照合し, 適合する成分で構成されたリストを返します. 猶, この照合は引数の最初のアلفベットと大域変数 `labels` に登録されたラベルの最初のアلفベットが一致するかどうかで行います. 従って, `labels(%i1)` でも `labels(i)` でも `labels(imax)` でも引数の最初のアلفベット `i` で照合を行う為, 結果として返されるリストは全ての入力ラベルで構成されたものになります. そして, 適合するものが無ければ空リスト `[]` を返却します.

猶, 入出力等のラベルを設定する大域変数 `inchar`, 大域変数 `outchar` や大域変数 `linechar` を再設定する事で, 大域変数 `labels` に登録されるラベルが切替えられます. そして, `labels` 関数の引数もそれに合せて与えると, その引数に対応するラベルのリストを返します.

9.5 結果の表示

Maxima では入力行に; を付けると, Maxima が評価した値が表示されます. 数値の四則演算はデフォルトの場合, 直ちに処理されて結果が表示されます. 更に Maxima は結果を二次元的に表示するのがデフォルトとなっています.

```

(%i43) 2/5;
                                         2
(%o43)                                     -
                                         5

(%i44) integrate(f(x),x,a,b);
                                         b
                                         /
                                         [
(%o44)                                     I  f(x) dx
                                         ]
                                         /
                                         a

(%i45) expand((x+1)*(x-1));
                                         2
(%o45)                                     x  - 1

```

この表示は式が小さなものであれば良いものですが、式が長くなると非常に判り難いものになります。そこで、表示を1行で済む様に指定が行える大域変数 `display2d` があります。この変数の値がデフォルトの `true` であれば、二次元的な表示を行い、`false` を指定すると、結果を入力可能な書式で表示します。

```

(%i4) display2d;
(%o4)                                     true
(%i5) 'integrate(f(x),x);
                                         /
                                         [
(%o5)                                     I f(x) dx
                                         ]
                                         /

(%i6) expand((x+1)^3);
                                         3      2
(%o6)                                     x  + 3 x  + 3 x + 1
(%i7) display2d:false;
(%o7) false
(%i8) 'integrate(f(x),x);
(%o8) 'integrate(f(x),x)

```

```
(%i9) expand((x+1)^3);
(%o9) x^3+3*x^2+3*x+1
```

Maxima にはこの他にも特殊な表示を行う関数があります。基本的に Maxima はキャラクター端末しかなかった昔の手順に対応したシステムの為、積分記号の様に文字を使って数式を表示するといった、ある意味では涙ぐましい努力の跡があります。しかし、現在の Window システムから見ると非常に古臭く感じるものが多いのが現状です。

9.5.1 表示に関連する大域変数

表示に関連する大域変数

変数名	初期値	概要
ibase	10	入力数値の基数を指定
obase	10	出力数値の基数を指定
absboxchar	!	絶対値を描く際に用いる文字を指定
display2d	true	結果の二次元表示の有無
leftjust	false	結果の左寄せ表示の有無
display_format_internal	false	内部表現に対応した表示切替
lispsdisp	false	LISP のシンボル表示に於ける?の有無
%edispflag	false	%e の冪の表示を指定
exptdisflag	true	負の冪を分数で表示
pfeformat	false	有理数の表示書式を決定
powerdisp	false	多項式の項の表示順を決定
stardisp	false	可換積演算子の表示を指定
linel	79	一行に表示する文字数
stringdisp	false	文字列表示で二重引用符の表示の有無
noundisp	false	名詞型の表示での単引用符の表示の有無
ttyoff	false	出力の停止制御

大域変数 `ibase` は入力数値の基数を指定します。値は十進数で 2 から 35 が指定可能です。10 より大きな値を設定した場合、A から開始する大文字のアルファベットを用います。猶、大域変数 `obase` は表示の際に用いる数値の基数を指定する大域変数となります。

大域変数 `absboxchar` は絶対値を描く際に用いる文字を指定します。猶、絶対値は精々一行の高さしかありません。

大域変数 `display2d` が `false` であれば、結果表示が二次元的書式ではなく、一行に収まる様に表示します。長い式や複雑な式、表示に余裕が無い場合は有効です。

大域変数 `leftjust` は結果の表示の左寄せ表示の有無を設定します。初期値の `false` で、結果は中央に揃えられますが、大域変数 `leftjust` が `true` であれば、結果が左寄せで表示されます。

大域変数 `display_format_internal` が `true` であれば、内部の数学的表現を隠した表示ではなく、内部表現を反映した表示に切替ります。但し、内部表現そのもので表示する訳ではありません。ここでの出力は `part` 関数に対応するのではなく、`inpart` 関数に準じたものになるとも言えます。

内部書式との比較

入力	<code>part</code>	<code>inpart</code>
<code>a-b;</code>	<code>a-b</code>	<code>a+(- 1)*b</code>
<code>a/b;</code>	<code>a/b</code>	<code>a*b^(-1)</code>
<code>sqrt(x);</code>	<code>sqrt(x)</code>	<code>x^(1/2)</code>
<code>x*4/3;</code>	<code>4*x/3</code>	<code>4/3*x</code>

大域変数 `lispdisp` は `false` の場合に LISP のシンボルの表示で?を省略します。

`%edispflag` は自然底 `%e` の冪表示を定める大域変数です。初期値の `false` の場合、`%e` の負の冪は負の冪のまま冪表示されます。例えば、`exp(-x)` は `%e` の負の冪 `%e^(-x)` で表示されます。`true` の場合、`%e` の負の冪は `%e` の冪の商の書式で表示されます。即ち、`%e^(-x)` は `1/%e^x` の書式で表示されます。

```
(%i2) exp(-x);
                                     - x
(%o2)                                     %e
(%i3) %edispflag:true;
(%o3)                                     true
(%i4) exp(-x);
                                     1
(%o4)  ---
                                     x
                                     %e
```

大域変数 `exptdispflag` が `true` であれば,Maxima は負の冪を持った項を分数式で表示します. 例えば, x^{-1} は $1/x$ と表示されます.

大域変数 `pformat` が `true` であれば, 有理数は行の中で表示され, 整数の分母は有理数の積として表示されます. 例えば, 入力が $b/4$ であれば, $1/4*b$ と表示されます. 但し, a/b の様に, a,b の両方が不定元であれば, 通常の表示になります.

大域変数 `powerdisp` は多項式の冪の表示順序を切替える大域変数です. 初期値の `false` の場合,Maxima の項順序 $>_m$ の大きなものから順に表示しますが,`true` に切換えると, 項順序 $>_m$ の小さなものから順に表示されます. 猶, この変数は `taylor` 級数には影響を与えません.

```
(%i1) powerdisp;
(%o1)                                     false
(%i2) expand((x+y)^3+x+y);
                                     3      2      2      3
(%o2)                               y  + 3 x y  + 3 x  y + y + x  + x
(%i3) powerdisp:true;
(%o3)                                     true
(%i4) expand((x+y)^3+x+y);
                                     3      2      2      3
(%o4)                               x + x  + y + 3 x  y + 3 x y  + y
```

大域変数 `stardisp` が `false` の場合, 可換積演算子は出力では省略されています. `true` であれば, 可換積演算子を表示します.

大域変数 `linel` は一行に表示される文字数を設定します. 変更は何時でも可能です.

大域変数 `stringdisp` は `false` の場合に Maxima の文字列に対して二重引用符を表示しません.`true` の場合, 文字列の表示では二重引用符を含めて表示を行います.

大域変数 `noundisp` が `false` の場合,Maxima の名詞型の対象の表示で単引用符を省略します.`true` の場合, 単引用符を含めて表示を行います.

大域変数 `ttyoff` が `true` であれば入力行の表示のみを行い, 通常出力を止めます. 但し, エラー表示は行います. 猶,`writefile` 関数で開いたファイルに対しても, 同じ出力になります.

9.5.2 式の表示を行う関数

—— 式の表示を行う関数 ——

```
display(< 式1>, < 式2>, ...)
disp(< 式1>, < 式2>, ...)
ldisplay(< 式1>, < 式2>, ...)
ldisp(< 式1>, < 式2>, ...)
print(< 式1>, < 式2>, ...)
grind(< 式 >)
```

display 関数式を表示します. その左側が未評価の $\langle \text{式}_i \rangle$ で, その右側の行の中心がその式の値となります. この関数は block や do 文で, 中途結果の表示を行うのに便利です. display の引数は通常, 原子, 添字された変数や関数呼出しになります.

disp 関数は display 関数に似ていますが引数の値のみを表示します.

ldisplay 関数と ldisp 関数は各々, display 関数と disp 関数に似ていますが, 中間ラベルを生成する点が異なります.

```
(%i18) a1[1,2]:128;
(%o18)                                     128
(%i19) display(a1[1,2]);
                                     a1      = 128
                                     1, 2

(%o19)                                     done
(%i20) disp(a1[1,2]);
                                     128

(%o20)                                     done
(%i21) ldisplay(a1[1,2]);
(%t21)                                     a1      = 128
                                     1, 2

(%o21)                                     [%t22]
(%i22) ldisp(a1[1,2]);
(%t22)                                     128
```

(%o22)

[%t23]

print 関数は、 $\langle \text{式}_i \rangle$ がら順番に評価を実行して、その結果を表示します。ここで、 $\langle \text{式}_i \rangle$ に含まれる原子や関数の前に単引用符'が置かれていたり、文字列 (全体を二重引用符で括ったもの) の場合は、評価を行わずに、そのまま表示を行います。

grind 関数は、 $\langle \text{式} \rangle$ を Maxima の入力に適した形式で表示します。grind 関数の返却値は常に done です。

—— 式の内部表現に関連して表示する関数 ——

```
dispterm (⟨ 式 ⟩)
reveal (⟨ 式 ⟩, ⟨ 深度 ⟩)
tcl_output (⟨ リスト ⟩, ⟨ 添字 ⟩, ⟨ 飛幅 ⟩)
tcl_output (⟨ リスト ⟩, ⟨ 添字 ⟩)
```

dispterm 関数は与式を内部表現に合わせて、第一層の部分式を順次表示します。

reveal 関数は $\langle \text{整数} \rangle$ で指定された各々の成分の長さで $\langle \text{式} \rangle$ を表示します。和は $\text{sum}(n)$ 、積は $\text{product}(n)$ として表示されます。

ここで n は和や積の成分の数になります。指数関数は expt で表現されます。

```
(%i11) aa:integrate(1/(x^3+2),x);
```

```

                                1/3
                                2 x - 2
                                atan(-----)
                                1/3
                                2   sqrt(3)
                                log(x - 2  x + 2  )
                                2/3
                                1/3
                                2   sqrt(3)
                                log(x + 2  )
(%o11) - ----- + ----- + -----
                                2/3
                                2   sqrt(3)
                                6 2
                                2/3
                                2   sqrt(3)
                                2
                                2/3
                                2   sqrt(3)
                                3 2
```

```
(%i12) reveal(aa,1);
```

```
(%o12) sum(3)
```

```
(%i13) reveal(aa,2);
```

```
(%o13) negterm + quotient + quotient
```

```
(%i14) reveal(aa,3);
```

```

                                atan          log
(%o14)      - quotient + ----- + -----
                                product(2)   product(2)

(%i15) reveal(aa,4);

                                log          atan(quotient)   log(sum(2))
(%o15)      - ----- + ----- + -----
                                product(2)   expt sqrt          3 expt

(%i16) reveal(aa,5);

                                sum(2)
                                atan(-----)
                                product(2)   log(x + expt)
(%o16)      - ----- + ----- + -----
                                6 expt       2/3           2/3
                                2   sqrt(3)   3 2

```

tclOutput 関数は、〈添字〉を展開した〈リスト〉に対応する tcl のリストを表示します。ここで、飛幅の初期値は 2 で、引数がリストで構成されたリストではなく、数値リスト形式の場合、飛幅から外れた全ての要素が表示されます。

9.5.3 後戻し表示を行う関数

xmaxima の様なグラフィカルな GUI 端末を持たなかった時代では特に重宝されたと思われる機能に、以前処理した結果の表示を再度表示する機能があります。これは playback 関数を用いて指定した入力とその処理結果を再表示させるもので、その際に再計算を行うものではありません。

後戻し表示の関数 playback

```

playback (〈整数〉)
playback ([〈整数1〉, 〈整数2〉])
playback ([〈整数〉])
playback ()
playback(input)
playback(slow)
playback(time)
playback(grind)

```


playback 関数は入力と出力行の後戻し表示を行います。引数が整数 n であれば、最近の n 個の式 (入力行%, 出力行%o と中間行%e を各々1個として数えます) を再表示します。ここで、引数がリスト [$\langle$ 整数 $_1$ $\rangle$, $\langle$ 整数 $_2$ $\rangle$] の場合、 $\langle$ 整数 $_1$ $\rangle$ から $\langle$ 整数 $_2$ $\rangle$ 迄の全ての行を再実行します。 $\langle$ 整数 $_1$ = 整数 $_2$ $\rangle$ であれば、引数として [$\langle$ 整数 $\rangle$] を指定します。

引数が指定されない場合には、全ての行が再表示されます。

その他の引数に, input, slow, time, grind があります。

まず, input の場合は入力行を再表示行します。

slow の場合は example 関数によるデモの処理と同様に一つの入力とそれに対応する処理結果を表示すると Maxima-break に入り, Enter キーの入力待ちになります。又, Enter キー以外のキーが入力されると playback 関数を終了します。猶, Maxima-break に入った時点で表示されるプロンプトは大域変数 prompt で設定されています。

time の場合には計算時間が式と同様に表示されます。

更に, gctime か totaltime であれば, `showtime:all;` を用いたのと同様に、計算時間の詳細が表示されます。

string であれば、全ての入力行の再表示を文字列として返します。

grind の場合は, Maxima で再入力可能な式を出力する grind モードで式の再表示を行います。

猶, playback 関数による行の再表示は, `playback([2,5],10,time,grind)` の様に複数の引数を組合せても構いません。

9.5.4 古風な表示を行う関数

Maxima は歴史のあるシステムです。その為、古風な表示を行う関数が幾つかあります。ここではその様な古風な関数を愛でる事としましょう。

—— 古風な表示を行う関数 ——

```
dpart( $\langle$  式 $\rangle$ ,  $\langle$  整数 $_1$  $\rangle$ , ...,  $\langle$  整数 $_n$  $\rangle$ )
lpart( $\langle$  文字列 $\rangle$ ,  $\langle$  式 $\rangle$ ,  $\langle$  整数 $_1$  $\rangle$ , ...,  $\langle$  整数 $_n$  $\rangle$ )
box( $\langle$  式 $\rangle$ )
box( $\langle$  式 $\rangle$ ,  $\langle$  文字列 $\rangle$ )
rembox( $\langle$  式 $\rangle$ )
rembox( $\langle$  式 $\rangle$ , unlabelled)
rembox( $\langle$  式 $\rangle$ ,  $\langle$  ラベル $\rangle$ )
```

dpart 関数は与えられた式の指定された階層を大域変数 boxchar に割当てられた文字を用いて囲って表示する関数です。昔のテキスト主流の時代では必要な機能だったのですが、現在では、やや際物的な機能です。

lpart 関数も dpart 関数とほぼ同じ機能の関数です。但し、lpart 関数は第一引数に文字列を指定し、これをラベルとして使えます。勿論、第一引数を""の様に空行を指定すれば、dpart 関数によるものと同じ結果が得られます。

```
(%i188) dpart(int(f(x),x),0);
                                "int(f(x), x)"
                                "int(f(x), x)"

(%o188)
                                "int(f(x), x)"
                                "int(f(x), x)"

(%i189) lpart("結構歪だね",int(f(x),x),0);
                                "結構歪だね"
                                "int(f(x), x)"

(%o189)
                                "int(f(x), x)"
                                "int(f(x), x)"
```

dpart 関数と lpart 関数と比較し、box 関数はお気楽です。こちらは式の最上層のみを大域変数 boxchar で指定した文字で囲う関数です。

```
(%i195) box(expand((x+1)^3));
                                " 3      2      "
                                "x  + 3 x  + 3 x + 1"
                                "x  + 3 x  + 3 x + 1"

(%o195)
                                "x  + 3 x  + 3 x + 1"
                                "x  + 3 x  + 3 x + 1"

(%i196) box(expand((x+1)^3),"朝日輝く金亀山");
                                "朝日輝く金亀山"
                                " 3      2      "
                                "x  + 3 x  + 3 x + 1"

(%o196)
                                "x  + 3 x  + 3 x + 1"
                                "x  + 3 x  + 3 x + 1"
```

これに対し、rembox 関数は dpart 関数、lpart 関数や box 関数で生成した ASCII ART(AA) から不要な箱を削除する関数です。

—— 古風な表示を行う関数に関連する大域変数 ——

大域変数	初期値	概要
boxchar	"	AA で用いる文字

大域変数 `boxchar` は `dpart` 関数等で行う箱表示の際に用いる文字を指定する大域変数です.

```
(%i178) dpart(int(f(x),x),1,1);
                                     ""
(%o178)                               int(f("x"), x)
                                     ""

(%i179) boxchar:"漢"$
(%i180) dpart(1/(V I P+1),2,1);
                                     1
(%o180)  -----
          漢漢漢漢漢
          漢V I P漢 + 1
          漢漢漢漢漢
```

猶, 本来の値に戻したければ, `boxchar:"\""` と入力します.

9.5.5 エラー表示

—— エラー表示の関数 ——

```
error(< 式1>, ..., < 式n>)
errmsg()
```

`error` 関数は引数を評価した結果を文字列に変換し, これらの文字列を成分とするリストを大域変数 `error` に割当て, その値を表示する関数です.

`errmsg` 関数は, 大域変数 `error` に割当てた文字列を表示する関数です. 猶, 大域変数 `errmsg` は `error` 関数による大域変数 `error` に割当てた文字列の表示を行うかどうかを指定する大域変数です.

```
(%i27) error("メッ!");
メッ!
-- an error.  To debug this try debugmode(true);
(%i28) error;
(%o28) [メッ!]
(%i29) errmsg();
```

```

メッ！
(%o29)                                     done
(%i30) errormsg:false;
(%o30)                                     false
(%i31) errormsg();
メッ！
(%o31)                                     done
(%i32) error("メッ！");
-- an error.  To debug this try debugmode(true);

```

この様に,error 関数は大域変数 errormsg の影響を受けますが,errormsg 関数は大域変数 errormsg の影響を受けません。

エラー表示に関連する大域変数

変数名	初期値	概要
error	[No Error.]	エラーメッセージが割当てられる大域変数
errormsg	true	エラーメッセージの表示を制御
error_size	10	エラーメッセージの大きさを制御
error_syms	[errexpl,errexpl2,errexpl3]	エラーメッセージ

大域変数 error は error 関数で割当てられたエラーメッセージが登録される大域変数です。

大域変数 errormsg は error 関数で大域変数 error に割当てた文字列を表示させるかどうかを制御する大域変数です。初期値の true であれば表示を行い,false であれば表示を行いません。

大域変数 error_size はエラーメッセージの大きさを制御します。error_size よりも大きな式は文字列に置換され、文字列には式が設定されています。文字列は利用者が設定可能なリストから取られます。

大域変数 error_syms はエラーメッセージで,error_size よりも大きな式は文字列に置換され、その文字列には式が設定されています。文字列は error_syms リストから取られて初期値は errexpl,errexpl2,

errexpl3 等々となっています。エラーメッセージが表示された後、例えば、"the function foo doesn't like errexpl as input."であれば、errexpl;と利用者が入力すると、その式を見る事が出来ます。

error_syms は、必要なら別の文字列を設定しても構いません。

9.6 ヘルプに関連する函数

Maxima にはオンラインマニュアルを持っています. このオンラインマニュアルを閲覧する為に, `describe` 函数, 記号"?"や記号"??"を用います. 更に, デモや例題を実行させる事が可能です.

—— オンラインマニュアルに関連する函数 ——

```
describe(< 事項 >)
demo(< ファイル名 >)
example(< 項目 >) example()
```

`describe` 函数はオンラインマニュアルを表示させる為の函数です. 引数としては, 函数名, 演算子名, 大域変数名等になります. 同様の事は演算子"?"でも可能です. 猶, 函数名や大域変数名があやふやな時は, 演算子"??"を用いた方が良いでしょう.

この `describe` 函数の仕組みについては §12 で `describe` 函数の改造の話を書いているので, そこを参照して下さい.

`demo` 函数は指定されたデモファイルを自動実行する函数です. このデモファイルの構造は通常のバッチファイルと同様の構造, 即ち, 入力と同じ書式になります.

`demo` 函数でデモファイルを実行すると, デモファイルに記述された Maxima の入力行を処理するたびに, プロンプト_を出力して処理を止め, キーの入力待ちになります. 通常は, セミコロン; や Enter キーを入力すると次に進みます.

デモファイルは何処に置いても構いませんが, 大域変数 `file_search_demo` に割り当てられたリストに登録されていないディレクトリに置いた場合には, そのファイルへの経路とファイル名を記述した文字列を `demo` 函数に与えなければなりません.

もし, 函数名のみを与えたいのであれば, 大域変数 `file_search_demo` に登録されたディレクトリ上に置く必要があります. ここでファイルの修飾子としては, `dem`, `dm1`, `dm2`, `dm3`, `dmt` が利用可能です. 猶, デモファイルは全ての函数やパッケージに用意されているとは限りません. ここで大域変数 `file_search_demo` にはデモファイルを検出する為の経路や修飾子が記述された LISP の文字列で構成されたリストが割り当てられています. この LISP の文字列の書式は, 経路/###.{拡張子₁, ..., 拡張子_n} の様な書式です. この実例を次に示しておきましょう.

```
/usr/local/share/maxima/5.13.0/demo/###.{dem,dm1,dm2,dm3,dmt}
```

従って, 自前のデモファイル群を置いたディレクトリを指定したければ, この書式の LISP の文字列を `file_search_demo` に割り当てられたリストに追加しなければなりま

せん.

example 関数は与えられた関数に対応する例題を返す関数です. example 関数で実行される例題は例題ファイルに登録されたもので, 関数名を文字列ではなくシンボルとして example 関数に与えます. 具体的には, algsys 関数の例題を見たければ, `example(algsys);` と入力し, 文字列"algsys"を引数にはしません. もしも, 対応する関数が存在しない場合, example 関数に引数を与えなかった場合, example 関数は実行可能な例題の一覧をリストで返します.

この example 関数の例題ファイル名は通常 maxima.demo ファイルであり, このファイルは大域変数 manual_demo に割り当てられています. そして, このファイルの所在は LISP 側の内部変数*maxima-demodir* に束縛されたディレクトリ名になります. 更に, example 関数向けの例題ファイルの構造は demo ファイルとやや勝手が違うものです. そこで, このファイルの構造を以下に示しておきます,

example 関数向けの例題ファイルの構造

```
ラベル1  <Maxima の入力行1>
          <Maxima の入力行2>
          ⋮
          <Maxima の入力行n>  &&
/*
    註釈行1
    ⋮
    註釈行m
*/
ラベル2  <Maxima の入力行1>
          <Maxima の入力行2>
          ⋮
          <Maxima の入力行n>  &&
```

この様に, 例題ファイルはラベルで開始する項目と注釈として開始する項目の二種類がありますが, 何れも, 項目の末尾には&&を置きます. この&&を省略すると, その項目が終了していないと判断されます. 但し, 例題ファイルの末尾の項目に対しては,&&を省略しても問題はありません.

例題ファイルの注釈はC風に/*と*/で括りますが, この注釈を終えると必ず末尾には&&を入れます. 又, ここでのMaximaの入力行とは通常のMaximaの文で, 行末に;や

\$を入れたものです.

example 関数は例題ファイルで引数に合致するラベルを捜し, 適合するラベルがあれば, そのラベルから&&が現われる迄の Maxima の入力行を処理します. ラベルは関数名である必要はありませんが, 出鱈目なものにすると後で困るので, 関数名や話題に沿ったものにすると良いでしょう.

—— オンラインマニュアル等に関連する大域変数 ——

file_search_demo	demo 関数向けのデモファイルの収納先を指定する大域変数
manual_demo	example 関数向けの例題ファイル名を指定する大域変数
help	help 関数向けの文字列が割り当てられた大域変数

大域変数 file_search_demo には demo 関数で用いるデモファイルが登録されているディレクトリ等がリストとして登録されています. 猶, この大域変数は LISP の文字列で構成された Maxima のリストが割り当てられています.

manual_demo は example 関数で用いる例題が登録されたファイル名が割り当てられています. 初期値としては, manual_demo が割り当てられています.

大域変数 help には次の LISP の文字列が初期値として設定されています.

type 'describe(topic);' or 'example(topic);' or '? topic'

猶, 関数の help もありますが, 大域変数 help に割り当てられた上記の文字列を表示する為だけの関数です.

Maxima の演算子"?"は変った性質を持っています. 関数としてはオンラインマニュアルを表示する演算子の様に振舞いますが, 特殊記号としては裏の LISP に"?"の直後の文字列を流し込む働きを行います.

—— 演算子"?"と演算子"??" ——

演算子	構文	概要
?	? 〈事項〉	〈事項〉に関連するオンラインマニュアルを表示
?	?〈LISP に評価させる関数〉	LISP の関数を Maxima の関数風に処理
??	?? 〈事項〉	〈事項〉に関連するオンラインマニュアルの一覧を表示

演算子”?”は Maxima で二つの意味を持ちます. 一つは, オンラインマニュアルの表示で用います. この場合,?との間に空行や tab を入れて調べたい〈事項〉を入力します. 〈事項〉に関連する項目が複数存在する場合,Maxima は一覧表を表示して事項に対応する数値か none が入力されるのを待ちます.

数値が入力されると該当するヘルプを表示して終了し,none の場合は何もせずにそのまま終了します. 猶, 返却値は false になります.

```
(%i5) ? prefix;
```

```
0: (maxima.info)prefix.
```

```
1: optimprefix :Definitions for Expressions.
```

```
Enter space-separated numbers, 'all' or 'none': 0
```

```
Info from file /usr/local/info/maxima.info:
```

```
5.5 prefix
```

```
=====
```

```
A 'prefix' operator is one which signifies a function of one argument,
which argument immediately follows an occurrence of the operator.
```

```
'prefix("x")' is a syntax extension function to declare x to be a
'prefix' operator.
```

```
See also 'syntax'.
```

```
(%o5)
```

```
false
```

もう一つの?の意味は, 引数の間に空白を空けずに利用した場合, ?の直後の引数を LISP の関数として評価して結果を返そうとします. この場合,?の引数に,LISP に評価させる関数を記述しますが, ここでは LISP の S 式ではなく,Maxima の関数風に表記したものを与えます. ここで, 与える引数も,Maxima の表の表記にする必要があります. 例えば,Maxima の変数 x は, 内部では \$x と表記されています. その x に割当てられた式の car を取りたい場合, `?car(x);` と入力し,?(car \$x) とはしませんここでこの関数を実行した時の返却値は全て Maxima のデータ型に変換されて返されます. ここで LISP の関数で式の評価をする演算子に, 演算子:lisp もあります. こちらは LISP の S 式を引数に取り, その返却値も LISP の書式で返却されますが, 演算子?と違ってラベルには保存されません.


```
(%i14) a1:x^2+y+z;

                                     2
(%o14)                               z + y + x
(%i15) ?car(a1);
(%o15)                               ("+", simp)
(%i16) :lisp (car $a1)
(MPLUS SIMP)
```

Maxima には更に”??”もあります。こちらは最初のオンラインマニュアルを表示する演算子”?”と似た働きをします。この”??”は与えられた事項に関連するオンラインマニュアルの一覧を表示して入力を待つものです。

この演算子”??”と演算子”?”の違いは、事項として明確に指定する必要が無い事です。一種のキーワード検索として用いる事も可能です。例えば、`?? inte`と入力した結果を以下に示しておきます。

```
(%i91) ?? inte

0: Functions and Variables for Elliptic Integrals
1: Functions and Variables for Integration
2: Functions and Variables for interpol
3: Interrupts
4: Introduction to Elliptic Functions and Integrals
5: Introduction to Integration
6: Introduction to interpol
7: askinteger (Functions and Variables for Simplification)
8: display_format_internal (Functions and Variables for Input and Output)
9: integerp (Functions and Variables for Miscellaneous Options)
10: integer_partitions (Functions and Variables for Sets)
11: integrate (Functions and Variables for Integration)
12: integrate_use_rootsof (Functions and Variables for Integration)
13: integration_constant_counter (Functions and Variables for Integration)
14: intersect (Functions and Variables for Sets)
15: intersection (Functions and Variables for Sets)
16: intervalp (Functions and Variables for orthogonal polynomials)
17: linearinterpol (Functions and Variables for interpol)
18: nonnegintegerp (Functions and Variables for linearalgebra)
19: orthopoly_returns_intervals (Functions and Variables for orthogonal polynomials)
20: poly_ideal_intersection (Functions and Variables for grobner)
Enter space-separated numbers, 'all' or 'none':
```

この様にマニュアルで文字列 inte を含む項目を一挙に表示します。ここで、左端の数字を入力すれば、該当する数字のマニュアルを表示し、all と入力すると全ての該当するマニュアルを表示して終了し、none と入力すれば何もせずに終了します。

9.7 システムの状態を調べる

9.7.1 大域変数 infolists

大域変数 infolists

変数名	概要
infolists	Maxima の情報リストの全ての名前を含むリスト

大域変数 infolists は幾つかの大域変数名で構成されたリストが割り当てられています.

infolists を構成する大域変数

values	利用者定義変数のリスト
aliases	利用者定義の変名リスト
arrays	利用者定義の配列リスト
dependencies	利用者定義の依存性リスト
functions	利用者定義関数と演算子のリスト
gradefs	利用者定義の勾配を持つ関数のリスト
labels	ラベルのリスト
let_rule_packages	let 関数で定めた規則パッケージのリスト
macros	利用者定義マクロのリスト
myoptions	利用者設定のオプションのリスト
props	属性のリスト
rules	利用者定義の規則のリスト
structures	その他

この大域変数 infolists に含まれる大域変数は基本的に利用者がシステムに対して行った設定や処理の内容が登録されるものです.

そして, 大域変数 infolists に含まれる大域変数を参照して, 情報の蒐集やデータや属性の変更や削除を行う関数が多くあります. 又, 利用者も, この変数名を直接入力する事で, Maxima に含まれる対象を調べる事が可能です.

ここで, このリストに含まれる大域変数と, その大域変数に割り当てられた情報の解説をしておきます.

aliases

大域変数 `alias` には利用者定義の別名を持つ原子のリストが割当てられています. `alias` 関数, `ordergreat` 関数, `orderless` 関数, 及び, `declare` 関数で名詞型 (noun) として宣言されたもの. Maxima の初期状態で, 空リスト `[]` が割当てられています.

arrays

大域変数 `arrays` には, 利用者が生成した配列のリストが割り当てられています. ここでの配列は, 演算子 `:`, 演算子 `::`, 及び, 演算子 `::=` で設定されたもので, Maxima の初期状態では空リスト `[]` が割当てられています.

dependencies

依存関係を持つ原子で構成されるリストが割当てられています. この依存関係は `depends` 関数や `gradef` 関数で設定されたものです. Maxima の初期状態で空リスト `[]` が割当てられています.

functions

大域変数 `functions` には演算子 `=` を用いて定義された利用者定義関数や演算子が収録されたリストが割当てられています. 猶, Maxima の初期状態では空リスト `[]` が割当てられています.

gradefs

`gradef` 関数で設定された利用者定義の微分を持つ関数のリストが割当てられています. この初期値は空リスト `[]` です.

labels

大域変数 `labels` には, 値が代入されている入力, 出力等のラベルで構成されたリストが割当てられています. このラベルは大域変数 `nolabels` が `false` の場合に自動的に追加されますが, `nolabels` が `true` の場合には追加されません.

let_rule_packages

let 関数を用いて定義された規則パッケージが登録されるリストが割当てられています. ここでの初期値は [default_let_rule_packages] になりますが, これは let 関数による規則の設定で, 利用者が明示的にパッケージを指定しなければ, 大域変数 default_rule_packages に規則が収納される為です. その為, let 関数でパッケージ名を指定すると, 大域変数 let_rule_packages にそのパッケージ名が登録されます.

macros

大域変数 macros には, 利用者が定義したマクロで構成されたリストが割当てられます. 猶, Maxima の初期状態で空リスト [] が割当てられています.

myoptions

大域変数 myoptions には利用者によって再設定された全てのオプションのリストが割当てられています. Maxima の初期状態では空リスト [] になっています. ここでのオプションは実際は Maxima の利用者変数でもラベルでも無いものが登録されます. 即ち, 大域変数 values にも大域変数 labels に含まれない, 値が束縛されたものの書換えが生じると, その大域変数が大域変数 myoptions に登録される仕組みです. 但し, 一旦変更した大域変数を初期値に戻しても, 大域変数 myoption から外されません. この大域変数 myoptions の値は, kill 関数でも削除出来ません.

```
(%i2) nolabels:true;
(%o2)                                     true
(%i3) display2d:false;
(%o3) false
(%i4) myoptions;
(%o4) [nolabels,display2d]
(%i5) display2d:true;
(%o5)                                     true
(%i6) myoptions;
(%o6) [nolabels, display2d]
```

props

atvalue 関数, matchdeclare 関数や declare 関数等によって属性を与えられた原子で構成されたリストが割当てられています.

rules

大域変数 rules は, 利用者定義の並び照合と簡易化の規則で, tellsimp 関数, tellsimpafter 関数, defmatch 関数や defrule 関数で設定されます. Maxima の初期状態で空リスト [] が割当てられています.

structures

大域変数 structure には空リスト [] が割当てられています. 但し, この大域変数は積極的に用いられてはいません.

values

大域変数 values には, 値が束縛されている利用者定義の変数のリストが割当てられています. Maxima の初期状態では空リスト [] が割当てられています.

9.7.2 status 関数と sstatus 関数

—— status 関数 ——

```
status (feature, <機能>)
status (feature)
status (status)
```

status 関数は Maxima の内部変数 *features* に割当てられたリストの一覧を返却したり, 指定した項目が *features* に含まれているかどうかを true と false の真理値で返す関数です. 猶, ここでの *features* は属性の feature や features とは無関係です.

```
(%i7) status(feature);
(%o7) [cl, mk-defsystem, readline, regexp, syscalls, i18n, loop, compiler,
clos, mop, clisp, ansi-cl, common-lisp, lisp=cl, interpreter, sockets,
```

```
generic-streams, logical-pathnames, screen, ffi, gettext, unicode,
base-char=character, pc386, unix]
(%i8) status(feature,powerpc);
(%o8)                                     false
(%i9) status(feature,"unix");
(%o9)                                     true
```

引数が feature のみの場合, status 関数は内部変数*features*に設定されたリストを返却します. このリストには, Maxima のコンパイルで用いた LISP に関するもの, 文字コードや国際化対応, ハードウェアや OS 等の情報があります.

status 関数で第一引数を feature にする場合, 第二引数は機能の各項目が, 内部変数*features*に含まれるかどうかを判断して返却します. ここで, 第二引数は二重引用符で括っていても, 括らずにシンボルのままでも構いません.

status 関数の引数が status の場合, feature と status で構成されるリストを返します. この status 関数の利用としては, 例えば, UNIX 環境で利用を前提としたパッケージを記述した場合, `status(features,unix)` の値が false となれば, 実行出来ない環境である旨を返す様にする事が出来ます.

但し, status 関数では*features*に項目があるかどうかを検出する事しか出来ません. そこで, 次に解説する sstatus 関数と組合せる事で, より自在に使える様になります.

sstatus 関数

```
sstatus (feature,< 情報 >)
```

内部変数*features*にパッケージが必要とする情報が設定出来ます. この sstatus 関数で*features*に情報を追加する事で, status 関数でその情報の検出が行える様になります.

```
(%i5) status(feature,"KNOPPIX/Math");
(%o5)                                     false
(%i6) sstatus(feature,"KNOPPIX/Math");
(%o6)                                     true
(%i7) status(feature,"KNOPPIX/Math");
(%o7)                                     true
(%i8) status(feature);
(%o8) [KNOPPIX/Math, cl, mk-defsystem, readline, regexp, syscalls, i18n, loop,
compiler, clos, mop, clisp, ansi-cl, common-lisp, lisp=cl, interpreter,
```

```
sockets, generic-streams, logical-pathnames, screen, ffi, gettext, unicode,
base-char=character, pc386, unix]
```

この例では, 最初に `status` 関数で `KNOPPIX/Math` が `feature` に含まれているかどうかを調べていますが, 勿論, この様な機能はありません. そこで `sstatus` で `feature` に `KNOPPIX/Math` を追加しています. すると, `status` 関数で `KNOPPIX/Math` があるかどうかを検出すると追加されている事が判ります.

`status` 関数と `sstatus` 関数を用いる事で, 記述したパッケージが必要とする環境の検証等が予め行える様になります.

9.7.3 room 関数

room 関数

```
room ()
room (true)
room (false)
```

`room` 関数は保存領域の状況を詳細な記述で出力します. この `room` 関数は LISP の `room` 関数を利用したものです.

9.8 時間に関連する関数

9.8.1 処理時間に関連する関数

処理時間表示の関数

```
time (<出力ラベル1>, <出力ラベル2>, ...)
```

`time` 関数は <出力ラベル> の結果を得る為に費した計算時間をミリ秒単位で表記したリストを返します. 猶, 大域変数 `showtime` を `true` にすると, 各出力ラベル (%o 行) の内容表示と共に計算時間が表示されます.

```
(%i17) showtime;
(%o17)                                     false
```

```
(%i18) integrate(sin(x)*exp(-x),x,0,inf);
1
(%o18) -
2

(%i19) showtime:true;
Evaluation took 0.00 seconds (0.00 elapsed) using 72 bytes.
(%o19) true
(%i20) integrate(sin(x)*exp(-x),x,0,inf);
Evaluation took 0.02 seconds (0.02 elapsed) using 109.234 KB.
1
(%o20) -
2

(%i21) time(%o18,%o20);
Time:Evaluation took 0.00 seconds (0.00 elapsed) using 96 bytes.
(%o21) [0.015998, 0.015998]
```

—— 時間に関連する大域変数 ——

変数名	初期値	概要
lasttime		直前に入力した計算時間
showtime	false	処理時間の表示の有無

p 大域変数 lasttime は直前に入力した式の計算時間をミリ秒単位で time と gctime の組合せを成分とするリストです。

大域変数 showtime が true であれば、出力式と共に計算時間の自動表示を行います。

`showtime:all;` で CPU 時間も含めて Maxima は計算処理でのゴミ収集 (gc) に費した時間を零でなければ表示します。この時間は `time=` の時間表示に含まれています。猶、`time=` には計算時間のみが含まれ、中間表示時間やファイルを読み込む時間は含まれてません。そこで、gc への反応性に分けて認識する事が難しい為、表示する `gctime` には計算の実行中に費やした全ての `gctime` を含んでいます。その為に、稀に `time=` よりも `gctime` の方が大きくなる事があります。

9.8.2 システムの時間を返す関数

—— システムの時間を返す関数 ——

```
timedate()
absolute_real_time()
elapsed_real_time()
elapsed_run_time()
```

先ず, `timedate` 関数はシステムの日時を返却する関数です. この関数は引数を一切必要としない関数で, 出力する書式も Maxima の利用環境に無関係に次の書式で定まっています:

—— `timedate` 関数が出力する時間の書式 ——

```
HH:   MM:  SS  Day,  mm/  dd/  yyyy  (GMT n)
時間: 分:   秒  曜日,  月/   日/   年   標準時からの差
```

```
(%i75) timedate();
(%o75)                18:50:28 Fri, 3/20/2008 (GMT+9)
```

この `time` 関数は内部の `get-decoded-time` 関数を用いて得られた結果を整形して表示している関数です.

`absolute_real_time` 関数は引数を必要としない関数で, 1900 年の 1 月 (UTC) から経過した時間を整数で返す関数です. この関数は LISP の `get-universal-time` 関数を用いています.

`elapsed_real_time` 関数は引数無用の関数で, Maxima が起動, 或いは再起動した時点からの経過時間を浮動小数点数で返します. この関数は LISP の `get-internal-real-time` 関数を用いています.

`elapsed_run_time` 関数も引数無用の関数で, Maxima が起動, 或いは再起動した時点から, 計算処理に要した時間を浮動小数点数で返します. この関数も `get-internal-real-time` 関数を用いています.

9.8.3 timer 関数, untimer 関数と timer_info 関数

timer 関数

```
timer(< 関数1>, ..., < 関数n>)  
timer(all)  
timer()  
untimer(< 関数1>, ..., < 関数n>)  
untimer()  
timer_info(< 関数1>, ..., < 関数n>)  
timer_info()
```

timer に関連する大域変数

```
timer_devalue false
```

9.9 式の変換を行う関数

Maxima の出力式は $\text{T}_{\text{E}}\text{X}$ や FORTRAN の書式に変換出来ます. この為に `tex` 関数や `fortran` 関数を用います. ここで, Maxima の関数やデータ構造は $\text{T}_{\text{E}}\text{X}$ や FORTRAN の書式と異なったものです. 先ず, $\text{T}_{\text{E}}\text{X}$ に関しては, 通常の C や FORTRAN とは全く異なった書式となります. そこで, Maxima は各関数や変数に予め設定した属性を利用して, $\text{T}_{\text{E}}\text{X}$ への式の内部表現の変換を実行しています. その為, 利用者定義の関数や演算子に対しては, `texput` 関数を用いて $\text{T}_{\text{E}}\text{X}$ の書式との対照関係を入れてやる必要があります.

これに対し, FORTRAN に関しては共通する式や表現も多い為, 比較的簡単な置換で済みます. 但し, `fortran` 関数にとって未知の関数や定義式については利用者が処理する必要があります.

9.9.1 T_EX の書式に変換

—— T_EX の変換に関連する関数 ——

```
tex(< 式 >)
tex(< 式 >, < ファイル名 >)
tex(< ラベル行 >, < ファイル名 >)
texinit(< ファイル >)
texend(< ファイル >)
texput(< 記号 >, < 文字列 >)
texput(< 記号 >, < 文字列 >, < 演算子型 >)
texput(< 記号 >, [< 文字列1>, < 文字列2>], matchfix)
texput(< 記号 >, [< 文字列1>, < 文字列2>, < 文字列3>], matchfix)
```

tex 関数は、与えられた < 式 > や < ラベル行 > を T_EX の書式に変換します。< ファイル名 > を指定すると、出力結果は指定ファイルに保存されます。猶、指定ファイルが既存すれば、その結果はファイル末尾に追加されます。猶、ラベル行を変換する場合、式のラベル番号も生成されます。

猶、tex 関数では Alpha から omega 迄のギリシャ文字読みの変数は全て T_EX のギリシャ文字表記で置換えられます。又、Maxima の関数や変数に関しても、tex 関数によって置換が行われます。

```
(%i15) tex([Alpha,beta,Gamma,Omega,omega]);
$$\left[ {\rm A} , \beta , \gamma , \Omega , \omega \right] $$
(%o15) false
(%i16) tex([exp(2*%pi*i*t)*sin(2*%pi/4*t),cos(u)^2+sin(u)^3]);
$$\left[ e^{2i\pi t}, \sin \left( \frac{\pi t}{2} \right) , \right. \\ \left. \sin ^3u + \cos ^2u \right] $$
(%o16) false
```

この置換は、Maxima の内部変数 defprop 関数を用いて texword 属性等の属性として T_EX の表記や置換関数を予め格納させており、tex 関数を用いて、その texword 属性で置換えたり、置換関数を作用させる事で変換を行っているのです。

猶、Maxima は、現在の Maxima に存在しない幾つかの関数 (cubrt, %j 等を T_EX の書式に変換する機能も持っています。

texend 関数は指定したファイルの末尾に \end を書込むだけの関数です。

texput 関数は指定した記号と $\text{T}_\text{E}\text{X}$ の書式を結び付ける関数です。利用者が定義した関数や変数等を $\text{T}_\text{E}\text{X}$ に自動的に変換させる為に用いる事が可能です。

先ず、第一引数が tex 関数で $\text{T}_\text{E}\text{X}$ の書式に変換すべき記号となります。第二引数はその記号に対応する $\text{T}_\text{E}\text{X}$ の書式の雛形となります。

引数が二つだけの場合、 $\alpha \rightarrow \alpha$ の様な対応付けとなります。変換すべき記号が演算子である場合、その演算子の特性を反映して変換を行う必要があります。この場合、第三引数に演算子の型を指定します。

この例を以下に示しておきましょう。

```
(%i3) infix("|-")$ infix("->")$ nary("|*|")$
(%i6) texput("|-", " \vdash ", infix)$
(%i7) texput("->", " \rightarrow ", infix)$
(%i8) texput("|*|", " \wedge ", nary)$
(%i9) neko:((A->B)|*(B->C))|-(A->C);
(%o9)          A -> B |*| (B -> C) |- (A -> C)
(%i10) tex(neko);
$$A \rightarrow B \wedge \left(B \rightarrow C\right) \vdash \left(A \rightarrow C\right)$$
(%o10)          false
```

ここでは $A \rightarrow B \wedge B \rightarrow C \vdash A \rightarrow C$ を tex 関数に生成させようとしたものです。

この tex 関数による結果を次に示しておきましょう。

$$A \rightarrow B \wedge (B \rightarrow C) \vdash (A \rightarrow C)$$

括弧の処理に問題がありますが、まあまあでしょうか。

texput 関数の最後の二つの型は、変換すべき演算子が matchfix 型の場合です。この場合、演算子は式を挟んで左右に存在する為、左右の記号の変換を指定しなければなりません。

```
(%i20) matchfix("(:-)", "(:-(");
(%o20)          (:-)
(%i21) (:-) x+1 (:-(;
(%o21)          (:-)x + 1(:-(
(%i22) texput("(:-)", ["\\int \\int", "\\int \\int"], matchfix);
(%o22)          [\\int \\int, \\int \\int]
```

```
(%i23) tex((:-) x+1 (:-());
$$\int \int x+1 \int \int $$
(%o23)
false
(%i24) texput("(-)",["\int \int ", "\int \int"],matchfix);
(%o24)
[\int \int , \int \int]
(%i25) tex((:-) x+1 (:-());
$$\int \int x+1 \int \int $$
(%o25)
false
```

ここで、最初の tex 関数による変換では、記号に対応する T_EX の書式に空行が無い為、 $\int x+1$ の様に、 $\int$ と $x+1$ が結合している事に注意して下さい。猶、Maxima-5.14.0 では Maxima の文字列が LISP の文字列型となった為か、演算子にアルファベット以外の文字を用いていると記号ではないとエラーになります。

9.9.2 FORTRAN の書式に変換

— fortran 関数と fortmx 関数 —

```
fortran(< 式 >)
fortmx(< 原子 >, < 行列 >)
fortmx(< 原子 >, < 行列 >, < ストリーム >)
```

fortran 関数は与式を FORTRAN の構文に変換します。一行が長くなり過ぎると、継続文字を用います。この文字は 1 から 9 までが順番に振られ足りなくなると、 $;$, $;$, $<$, $=$, $>$, $;$, $@$ が順番で振られ、それから再度 1 から 9 を繰り返します。

この fortran 関数は Maxima の式を FORTRAN の書式に合せます。例えば、式中の演算子 $\wedge$ は $**$ で置換し、複素数 $a + b\%i$ は (a,b) に置換えます。

与式は方程式であって構いません。この場合、方程式の右辺を左辺に割当ての形で出力します。ここで、右辺が行列名であれば、fortran 関数は、行列の各成分に割当を行う様に変換します。

もし、与式に fortran 関数で解釈出来ないものがあつた場合、grind 関数を用いて式の内部表現の変換を行って、表示されているのと同じ式が出力します。

```
(%i56) expr: (a+b+1)^5;
5
(%o56)
(b + a + 1)
```

```
(%i57) fortran(expand(expr));
      b**5+5*a*b**4+5*b**4+10*a**2*b**3+20*a*b**3+10*b**3+10*a**3*b**2+3
1    0*a**2*b**2+30*a*b**2+10*b**2+5*a**4*b+20*a**3*b+30*a**2*b+20*a
2    *b+5*b+a**5+5*a**4+10*a**3+10*a**2+5*a+1
(%o57)
done
(%i58) prefix("|-");
(%o58)
|-
(%i59) fortran( |- (x^2+y^2-a^2=0) );
      |- (y**2+x**2-a**2 = 0)
(%o59)
done
```

この例では最初に多項式の変換を実行しています。その次の例では前置演算子|-を定義し、その式を fortran 関数に与えています。勿論,fortran 関数にとって、演算子|-は未知の代物の為、単純に出力を通常の式表示と同じものに変換しています。

fortran 関数の引数に行列が含まれている場合,fortran 関数は fortmx 関数を用いて行列の処理を実行します。この場合、第一引数に fortran プログラムで用いる配列名、第二引数に行列を指定します。猶、第三引数にストリームを指定すると、出力が指定したストリームに対して行われます。

```
(%i25) A1:matrix([1,2,3],[4,3,2]);
      [ 1  2  3 ]
(%o25) [
      [ 4  3  2 ]

(%i26) neko:openw("testfile")$
(%i27) fortmx( testarray,A1);
      testarray(1,1) = 1
      testarray(1,2) = 2
      testarray(1,3) = 3
      testarray(2,1) = 4
      testarray(2,2) = 3
      testarray(2,3) = 2
(%o27)
done
(%i28) fortmx( testarray,A1,neko);
(%o28)
done
```

この例では、行列 A1 を fortmx 関数を用いて fortran の書式に変換しています。最後

では, ストリームを指定しており, 結果はストリームの出力先のファイル `testfile` に書込まれます.

— fortran 関数に関連する大域変数 —

<code>fortindent</code>	<code>0</code>	左側の空白を制御
<code>fortspaces</code>	<code>false</code>	80 カラム出力の制御

大域変数 `fortindent` は左側の空白を制御します. デフォルトの `0` で, 通常の空白行 (6 空白行) となります.

大域変数 `fortspaces` が `true` であれば, `fortran` 関数は 80 列で出力を行います.

9.10 便利な関数

— 便利な関数 —

```
alias(< 新名称1>, < 旧名称1>, < 新名称2>, < 旧名称2>, ...)
```

```
apropos(< 文字列 >)
```

`alias` 関数は (利用者, 又はシステム) 関数, 変数, 配列等に別名を与えます. 引数は新名称と旧名称の一組となるので, 偶数個の引数が必要になります.

`alias` 関数は `< 新名称i>` を大域変数 `aliases` に追加します. この時, `< 新名称i>` には `alias` 属性の属性値として `< 旧名称i>` が LISP の `putprop` 関数で付与されます.

`apropos` 関数は `< 文字列 >` を含む Maxima の関数, 大域変数のリストを返します, 例えば, `apropos(exp);` の結果は, `expand, exp, exponentialize` 等の名前の一部に文字列 `exp` を含む全ての大域変数や関数のリストになります.

この関数を使えば, 大域変数や関数の名前の一部だけさえ覚えていれば, 残りの名前を探す事が可能になります.

— システムに関連する大域変数 —

変数名	初期値	概要
<code>aliases</code>	<code>[]</code>	別名リスト
<code>debugmode</code>	<code>false</code>	<code>break loop</code> に入るかどうかを設定
<code>myoptions</code>	<code>[]</code>	オプションを蓄えるリスト
<code>optionset</code>	<code>false</code>	オプション設定時にメッセージの有無

大域変数 `aliases` は `alias` 関数, `ordergreat` 関数, `orderless` 関数や原子を名詞型と宣言する事によって設定された別名を持つ原子のリストです。

大域変数 `debugmode` は `true` の場合, エラーが生じた時や `false` で中断モードに入った時は何時でも Maxima の `break loop` に入ります. `all` が設定されていれば, 実行中の関数のリストに対して `backtrace` を調べる事が出来ます。

大域変数 `myoptions` は利用者が設定した全てのオプションを蓄えるリストです。

大域変数 `optionset` が `true` であれば, Maxima はオプションが再設定された時点でメッセージを表示します. これはオプションの綴りが不確かな場合, 割当てた値が本当にオプションの値となっているかを確認したい時に便利です。

9.11 外部プログラムの起動

Maxima では Maxima 外部のプログラムを起動する事が可能です. この場合, `system` 関数を用います。

— system 関数 —

```
system(< 命令 >)
```

Maxima 外部の命令を実行します. オプションを持つ命令を実行したければ, 命令全体を文字列として `system` 関数に引渡します. 例えば, `ls -a` を実行したければ, `system("ls -a");` と入力します。

猶, UNIX 環境で外部プログラムを長く利用し, その間に Maxima も利用したければ, 命令の末尾に `&` を入れた文字列を `system` 関数に引き渡します. 例えば, `surf` を用いたければ, `system("surf&");` の様に `&` を使います. もし, `&` が無ければ, `system` 関数が外部プログラムが終了するまで実行され続ける為, Maxima による処理は停止したままです。

9.12 Maxima の終了

— Maxima を終了させる関数 —

```
logout()  
quit()
```


Maxima の終了は `logout` 関数や `quit` 関数を用います。これらの関数は引数を必要としませんが、必ず `quit()` の様に後の小括弧 `()` を忘れないで入力します。

先ず、`logout` 関数は単純に LISP の `(bye)` 関数を実行させるだけです。その為、処理系によっては上手く動作しないものもあるかもしれません。

これに対し、`quit` 関数は Maxima の内部変数 `*maxima-epilog*` に束縛した文字列を表示してから終了します。この `quit` 関数では様々な Common LISP の処理系 (KCL, CMUC, SCL, SBCL, CLISP, MCL, GCL) に対し、それらに応じた終了命令を送り込むので、通常の利用では `quit` 関数を用いて終了させるのが問題無いでしょう。

```
(%i1) :lisp (setf *maxima-epilog* "さらば、Maxima の利用者諸君!")
```

```
さらば、Maxima の利用者諸君！
```

```
(%i1) quit();
```

```
さらば、Maxima の利用者諸君！
```

この例では、内部変数 `*maxima-epilog*` に文字列を `setf` 関数を用いて束縛させており、`quit` 関数を実行すると、この内部変数に束縛した文字列が LISP の `princ` 関数を用いて表示されています。

猶、Maxima を一時的に停止させるのであれば、`Ctrl+C (^C)` を入力します。

`to_lisp` 関数を用いて LISP 環境に入った場合に、Maxima を全て終了させたいければ、`($quit)` と入力します。こちらの表記が Maxima の `quit` 関数の LISP 上に於ける表記になります。

9.13 虫取りに関連する関数

Maxima には他のプログラム言語と同様に, 虫取りの関数が幾つか用意されています. ここでの節では, 虫取りに関連する関数と大域変数について解説します. 猶, `mtrace.lisp` にここで解説する関数が定義されています.

9.13.1 trace 関数と untrace 関数

trace 関数の構文

```
trace ( < 関数1 >, ..., < 関数n > )
trace(all)
trace()
trace_it(< 関数 >)
```

trace 関数に関数を指定する事で, 指定した関数の動作追跡を行います. 関数の指定は, `trace(integrate);` の様に一つの関数のみを指定したり, `trace(integrate, factor);` の様に複数の関数を指定する事が可能です. 又, 引数に `all` を指定した場合, 全ての関数に対し, 動作追跡を行う事になります. 猶, 引数を指定しない `trace()` の場合, trace 関数による動作追跡が設定されている関数のリストを返します.

`trace_it` 関数は trace 関数と同じ働きをしますが, 引数は一つの関数だけです.

それでは簡単な例で確認してみましょう.

```
(%i15) trace(integrate);
(%o15) [integrate]
(%i16) integrate(2*sin(x)*cos(x),x);
1 Enter integrate [2 cos(x) sin(x), x]
2
1 Exit integrate - cos (x)
2
- cos (x)
(%o16)
(%i17) trace();
(%o17) [integrate]
(%i18) trace(integrate,risch);
```

integrate is already traced.

```
(%o18)                                [risch]
(%i19) trace();
(%o19)                                [risch, integrate]
```

ここでの例では最初に integrate 函数の追跡を行う様に trace 函数で指定しています. この状態で, integrate 函数を起動すると, integrate 函数への入力と integrate 函数の出力が表示されます. 次に, trace() を実行する事で, trace 函数による追跡が指定された函数のリストを返します.

次に, integrate 函数と risch 函数を trace 函数で指定しています. この場合, integrate 函数は既に指定済みの為に警告が出ますが, risch 函数の登録は実行されます.

さて, integrate 函数ではオプションの risch を ev 函数で指定する事で, Risch 積分を行います. この場合の動作はどの様になるでしょうか. 先程の例の続きを示しておきましょう.

```
(%i25) ev(integrate(3^log(x),x),'risch);
              log(x)
1 Enter integrate [3      , x]
              log(3) log(x)
              x %e
1 Exit  integrate -----
              log(3) + 1
              log(3) log(x)
              x %e
(%o25)          -----
              log(3) + 1

(%i26) integrate(3^log(x),x);
              log(x)
1 Enter integrate [3      , x]
              1
              (----- + 1) log(x)
              log(3)
              3
1 Exit  integrate -----
              1
              (----- + 1) log(3)
              log(3)
```

```

1
(----- + 1) log(x)
log(3)
3
(%o26) -----
1
(----- + 1) log(3)
log(3)

(%i27) risch(3^log(x),x);
log(x)
1 Enter risch [3 , x]
log(3) log(x)
x %e
1 Exit risch -----
log(3) + 1
log(3) log(x)
x %e
(%o27) -----
log(3) + 1

```

ここでの例では, `ev` 関数を用いる事で `risch` 積分も実行させています. 但し, ここでの例では, `risch` 関数の本体を使うものの, Maxima の `risch` 関数そのものは使いません. その為, `trace` 関数では `risch` 関数の事は表に出て来ません. 但し, これは随分と捻くれた使い方をしているので, 現れなくても問題はありません.

`trace` 関数と逆の指定を行うのが `untrace` 関数です. この `untrace` 関数の構文を以下に示しておきましょう.

untrace 関数の構文

```

untrace ( < 関数1 >, ..., < 関数n > )
untrace()

```

`untrace` 関数は, `trace` 関数による動作追跡が指定された関数に対し, その動作追跡の解除を行います. 引数を指定しない場合, 無条件で全ての関数の動作追跡を解除します. 単純に `trace` 関数を関数に作用させるだけでは, その関数への入力と出力だけしか出力されませんでした. その為, より詳細な情報を得る為には, `trace` 関数に対するオプション設定が必要になります. このオプションの設定は `trace_options` 関数を用います.

— trace_options 関数 —

```
trace_options(< 関数 >, < オプション1 >, ..., < オプションn >)
trace_options(< 関数 >)
trace_options()
```

先ず, 基本的な構文としては, 第一引数に trace 関数で追跡を行う関数名を指定し, その後に指定可能なオプションを記入します.

猶, オプション無しの trace_options(< 関数 >) で, 追跡を行う関数に設定したオプションを解除し, 初期状態に戻します.

指定可能なオプションを以下の表に纏めておきます.

— 指定可能なオプション —

noprint	true の場合には表示を行いません.
break	true の場合には breakpoint を与えます.
lisp_print	true の場合,LISP の内部形式で表示を行います.
info	
errocatch	true の場合, エラーが捕捉されます.

猶,trace_options で設定した trace 関数のオプションの情報は get 関数を用いて取出す事が可能です.

```
(%i9) trace_options(integrate,info);
(%o9)                                     [info]
(%i10) get('integrate,'trace_options);
(%o10)                                     [info]
```

この例で判る様に,trace_options によって関数の trace_options 属性にオプションの値が設定されます. その為,get 関数を用いる事で, オプションの情報が入手可能となる訳です.

9.13.2 bug_report 関数と build_info 関数

— bug_report 関数と build_info 関数 —

```
bug_report()
build_info()
```

bug_report 関数は Maxima の虫取りを行う上で必要となる Maxima の情報を出力する関数です。この関数は引数を必要としません。猶、この関数の実体は, build_info 関数で, build_info 関数から返された情報に予め用意した文書を追加して表示させているだけです。

build_info 関数は Maxima を構築する際に用いた Common Lisp の情報等, Maxima の生成に関連する情報を返す関数です。この関数は内部変数の *maxima-build-time* , *autoconf-version* , *autoconf-host* , の内容や, 内部関数の lisp-implementation-type 関数 や lisp-implementation-version 関数 が出力する情報を纏めて返す関数です。

9.13.3 関連する大域変数

虫取りに関連する大域変数

大域変数名	初期値	概要
setcheck	false	変数をリストで指定し, その挙動を追跡する
setcheckbreak	false	
setval	'setval	Maxima break 時に追跡している変数の値を一時的に保全
trace	[]	追跡を行う関数のリスト
trace_max_indent	15	
trace_break_arg		
trace_safety	true	

setcheck の初期値は false ですが, ある特定の変数がどの様に変換されて行くかを追跡したい場合に, その追跡したい変数のリストとして指定します。

それでは実際にその効果を試してみましょう。

```
(%i2) setcheck: ['x, 'y];
(%o2) [x, y]
(%i3) x:128;
x SET TO 128
(%o3) 128
(%i4) y:x*2;
y SET TO 256
(%o4) 256
(%i5) x:'x;
```

```
(%o5)                                     x
(%i6) x:10;
x SET TO 10
(%o6)                                     10
```

この様に, `setcheck` に指定した変数リストに含まれる変数に対して割当てが発生して時点で, `x SET TO 10` の様に表示されます.

`setcheck` は `all` で全ての変数に `true` 設定しても構いません. 但し, `x:'x` の様に `setcheck` で指定された変数がそれ自身に設定されていると, 表示されません.

`setcheckbreak` の値が `true` の場合, `setcheck` リストに収録された変数に値の設定が行われる時に `break` による処理が中断されます. この時点で `setval` 変数に設定されようとする変数の値が保全されます. 猶, `setval` 変数に保全された値を別途の再設定して, `setcheckbreak` に設定された変数の値を意図的に変更しても構いません.

大域変数 `setval` は `setcheckbreak` 時に, 追跡している変数の値を一時的に蓄える為に用いられます. この変数は Maxima Break 時にはじめて値が設定され, 利用者はその値の書換も可能です.

9.13.4 LISP の trace 関数との併用

Maxima の `trace` 関数では Maxima 側の処理しか見えません. 例えば, 演算子 `and` の動作を Maxima の `trace` 関数を用いて見てみましょう.

```
(%i4) trace("and");
(%o4)                                     ["and"]
(%o4)                                     []
(%i5) (x >1+2) and (y-1>2);
1 Enter "and" [x > 1 + 2 and y - 1 > 2]
1 Exit  "and" x > 3 and y - 1 > 2
(%o5)                                     x > 3 and y - 1 > 2
```

この様に, `and` 演算子で処理する述語の入力と, それに対応する出力が Maxima の式で出力されます. では, この内部処理をより深く見る為に, 演算子 `and` に与えられた与式を簡易化して評価する内部関数の `simp-mand` の動作を見ます. その為に, 演算子 `:lisp` を用います. 演算子 `:lisp` は Maxima 側から `:lisp <LISP の S 式>` と入力すると, 与えられた S 式を LISP に引き渡して処理させる関数です. 従って, この場合には `:lisp (trace simp-mand)` と入力します.

```
(%i6) :lisp (trace simp-mand)
WARNING: TRACE: redefining function SIMP-MAND in top-level, was defined in
      /usr/local/maxima-5.13.0/src/binary-clisp/compar.fas
;; Tracing function SIMP-MAND.
(SIMP-MAND)
(%i6) (x >1+2) and (y-1>2);
1 Enter "and" [x > 1 + 2 and y - 1 > 2]
1 Exit  "and" x > 3 and y - 1 > 2
1. Trace:
(SIMP-MAND
'((MAND) ((MGREATERP SIMP) $X 3) ((MGREATERP SIMP) ((MPLUS SIMP) -1 $Y) 2)) '1
'NIL)
1. Trace: SIMP-MAND ==> ((MAND SIMP) ((MGREATERP SIMP) $X 3) ((MGREATERP SIMP) ((MPLUS SIMP) -1 $Y) 2))
(%o6)
      x > 3 and y - 1 > 2
```

このようにする事で、どのような処理が内部で実行されているかが内部表現から明確になり、より詳細な情報を得る事が可能になるのです。

猶、LISP でトレースされている関数の一覧を見る場合には、(trace) と入力し、トレースを除外する場合は、(untrace) で全てを、(untrace < 関数₁>, ..., < 関数_n>) で指定した関数のトレースの除外が出来ます。

9.13.5 システムの検証計算を行う関数

Maxima には標準でシステムの検証を行う関数 `run_testsuite` 関数があります。

run_testsuite 関数

```
run_testsuite()
run_testsuite(t)
```

この関数は引数を必要としない関数で、大域変数 `file_search_tests` に割当てられたリストに登録されたディレクトリ上のテストプログラムを実行します。猶、この大域変数のディレクトリは内部変数の `*maxima-tesitdir*` に束縛された値でもあります。

```
(%i4) run_testsuite();
Running tests in rtestnset: 502/502 tests passed.
Running tests in rtest1: 27/27 tests passed.
Running tests in rtest1a: 24/24 tests passed.
```

… 途中省略


```
Running tests in rtest_sign: 89/89 tests passed (not counting 11 expected errors).
```

```
No unexpected errors found.
```

```
Real time: 137.03793f0 sec.
```

```
Run time: 135.00844f0 sec.
```

```
Space: 3293125704 Bytes
```

```
GC: 587, GC time: 26.913685f0 sec.
```

```
(%o0)                                done
```

ここで,run_testsuite の引数を t とした場合,Maxima は計算の過程を詳細に表示しながら検証を行います. この際に, 既知の虫があれば, その旨も表示しながら処理を行います.

猶, 大域変数 testsuite_files は内部変数*maxima-tesitdir*で示されるディレクトリにある testsuite.lisp ファイルに記述されたリストの値です. この大域変数はテストファイルの名前と, 既知の虫 (数字で表記) で構成されたリストです.

第10章 Maximaでグラフ表示

この章では Maxima のグラフ表示機能について述べます. ここでは, Maxima のグラフ表示関数の解説に留まらず, 実際に使い熟す為に重要と思われる, gnuplot の利用方法, 更には, Maxima の draw パッケージの利用についても解説を行います.

10.1 Maxima のグラフ表示

10.1.1 はじめに

Maxima には目的に応じて様々な描画が可能な函数を幾つか持っています. Mathematica や Maple といった商用の数式処理システムと比較すると, 流石に見劣りする面もありますが, 一寸した式やデータの可視化ではそこその機能は持っています. Maxima での描画の特徴としては, Maxima で生成したデータを外部のアプリケーションに引渡して, そのアプリケーションで描画を行う点です. 函数によっては, この表示用のアプリケーションを目的に応じて切り替える事が可能なものもありますが, 特定のアプリケーションのみに対応したものもあります.

Maxima の標準的な描画函数は `plot2d` 函数と `plot3d` 函数です. これらの函数は各々, 2 次元グラフと 3 次元グラフを専門に描く函数です. `plot2d` 函数や `plot3d` 函数以外の函数で, 機能的にこれらの函数に勝る物は `draw` パッケージのものを除くとありません. 猶, `draw` パッケージでは `gnuplot` のバージョンが 4.2 以上に限定されています. ところで, MS-Windows 版では `draw` パッケージにも対応した `gnuplot` が附属している為に, 最初から利用可能ですが, その他の環境では, `gnuplot` を 4.2 以上に入れ直しておく必要があります. その為, `draw` パッケージについては次の章で詳細を述べる事とし, ここでは上記の `plot2d` 函数と `plot3d` 函数を中心に解説します.

10.1.2 表示アプリケーションについて

Maxima は描画用の特別なモジュールを持っている訳ではありません. Maxima の描画函数の動作手順は, 最初に Maxima 内部で描画用のデータを生成し, それをファイルに出力して Maxima から外部のアプリケーションを呼出し, そのデータの描画を外部アプリケーションに任せて行うという手法を採用しています.

さて, `plot2d` 函数と `plot3d` 函数の大きな特徴としては, 目的に応じて様々な外部アプリケーションを呼出す事が可能な点です. ここで, この `plot2d` 函数と `plot3d` 函数で利用可能な外部アプリケーションで代表的なものを以下に挙げておきましょう.

— `plot2d` 函数や `plot3d` 函数で利用可能なアプリケーション —

<code>gnuplot</code>	汎用 (デフォルト)
<code>openmath</code>	汎用
<code>Geomview</code>	3 次元グラフ描画専用

Maxima-5.13.0 のデフォルトの描画アプリケーションは gnuplot です. 但し, 以前のデフォルトは openmath でした.gnuplot と openmath を単純に比較すると, gnuplot の方が遥かに高機能ですが,Maxima 側からグラフを一寸描く程度の処理であれば大差の無い処理が出来ます. 寧ろ,openmath の方が 3 次元グラフィックスが綺麗かもしれません. 更に,openmath は Maxima のソースファイルに附属している事もあって,Maxima が利用可能な環境であれば,openmath も使えるという長所もあります.

最後の Geomview は 3 次元グラフ専用ですが, 非常に高品質のグラフを表示する事が可能なだけでなく, Euclid 空間以外の空間内部でのグラフの表示も可能です. 但し, Geomview の動作環境は基本的に UNIX 環境に限定されます.

10.1.3 フロントエンド

plot2d 函数や plot3d 函数にはこれといって特殊なフロントエンドがある訳ではありません. 基本的に描画用のデータファイルを出力した後は, 呼出したアプリケーションに全てを任せる為です. 但し, Maxima-5.12.0 から導入された新しい方式を利用すれば, 外部アプリケーションを gnuplot に指定した場合に限って Maxima から再描画の実行といった処理が行える様になっています.

猶, openmath が以前のデフォルトだった事もあって, Maxima のフロントエンドの一つの xmaxima の入力副ウィンドウ内にグラフを表示させる事が可能です.

10.1.4 plot2d 関数と plot3d 関数の動作

最初に plot2d 関数と plot3d によるグラフ表示について簡単に説明しましょう。これらの関数を用いてグラフの表示を行う場合、Maxima 内部で指定された外部アプリケーション向けのグラフデータを最初に生成します。外部アプリケーションの指定は大域変数 `plot_options` の `plot_format` に対して行います。又、データファイルの置かれる場所は UNIX 環境ではホームディレクトリ上、MS-Windows 環境の場合は、Documents and Settings フォルダの中にあるログインユーザー名のフォルダになります。

ここで、データファイルの内容は外部アプリケーションによって詳細は異なりますが、基本的に曲線や曲面を構成する点の座標値を示す数値データを中心に構成されたもので、Maxima で plot2d 関数や plot3d 関数に入力した式そのものが含まれている訳ではありません。

それから Maxima は外部アプリケーションに生成したファイルを引渡し、外部アプリケーションの立ち上げを行います。すると外部アプリケーションが実際の描画を行います。この様に、グラフデータファイルを引渡ししてしまうと後は外部アプリケーション任せになります。

外部アプリケーションの切り替えは前述の様に、`plot_format` に値を設定する事で行いますが、大域変数 `plot_options` に含まれる `run_viewer` の設定で外部アプリケーションを起動させずに、外部アプリケーション向けのデータのみを生成させる事も可能です。さて、ここで外部アプリケーションに引渡されるファイルの名前は、外部アプリケーション毎に異なっています。具体的には、`openmath` で `maxout.openmath`、そして、`Geomview` では `maxout.geomview`、`gnuplot` の場合は、`maxout.gnuplot` と `maxout.gnuplot_pipes` と `maxout` の後に `plot_format` の値に対応するアプリケーション名が付いたものです。ここで `gnuplot` には `gnuplot` と `gnuplot_pipes` の二種類の設定可能な `plot_format` の値があります。最初の `gnuplot` を選択すると、一旦、グラフを表示させた場合、Maxima 側から設定の変更や再描画は行えず、マウス操作によるグラフの拡大や縮小、3 次元グラフの回転といった操作が行なえません。それに対して `gnuplot_pipes` の場合、Maxima 側からの設定の変更や再描画やマウスによる画像操作が可能になります。

ここでマウスによる画像の直接操作では `gnuplot` の `mouse` を `on` にする事で行っています。逆に `mouse` を `off` にしてウィンドウの直接操作を停止したければ、グラフを表示しているウィンドウ上で、キーボードから直接 `m` と入力します。そして、`off` から `on` にする場合も同様に `m` と入力します。猶、`gnuplot` を立ち上げて処理を行っている場合は、`gnuplot` の命令入力ウィンドウから、`set mouse` を実行すると `on` に、`unset mouse` を入力すると `off` に出来ます。猶、MS-Windows 版固有の機能として、`mouse` を `off` にした状態からメニューを呼出して、表示画像をクリップボードに保存する事が可能で

す. その為には, mouse を off にした状態で右マウスボタンをクリックします. するとメニューが現われるので, そのメニューから Copy to Clipboard を選択すると画像がクリップボードに貼り付けられます.

次に maxout.gnuplot.pipes ファイルと maxout.gnuplot ファイルの場合の違いを解説しておきましょう. 先ず, maxout.gnuplot.pipes には描画する為の数値データのみが格納されており, 描画命令や設定等の gnuplot の命令文は Lisp のストリームを用いて gnuplot に送り込まれます.

それに対し, maxout.gnuplot ファイルの内容は, maxout.gnuplot.pipes に含まれている曲面や曲線のデータに加え, gnuplot の描画命令や設定を含む命令文がその先頭に入っています. その為, 別途立ち上げた gnuplot から `load 'maxout.gnuplot'` で読込めば, Maxima の時と同じグラフが表示されます. 更に, gnuplot で mouse を on にしておけば, マウスによる画像の回転や拡大が可能になります.

猶, MS-Windows のみは gnuplot ではなく wgnuplot を利用します. 但し, ここでは gnuplot と wgnuplot を区別せずに, gnuplot と呼んでいます. 猶, この wgnuplot は MS-Windows 版のバイナリに同梱されているので別途入手する必要はありません. この wgnuplot は Maxima のフォルダ (通常は Program Files フォルダにある筈です) の bin フォルダにあります. 因に, この wgnuplot は独立して動かす事も可能です. 但し, 最初に立ち上げた時点では, フォントが潰れて読めない状態なので, gnuplot のウィンドウで右マウスボタンをクリックし, Choose Font を選択して適当なフォントに設定し直せば読める様になります.

10.1.5 plot2d 関数

plot2d 関数は平面曲線の描画を行う関数です. 以下にその構文を示しておきます.

plot2d の構文

```
plot2d (< 式 >, < 定義域 >, < オプション1 >, ..., < オプションn >)
plot2d (< 式 >, < 定義域 >, < 値域 >, < オプション1 >, ..., < オプションn >)
plot2d (< 媒介変数式 >, < オプション1 >, ..., < オプションn >)
plot2d (< 媒介変数式 >)
plot2d (< 離散式 >, < オプション1 >, ..., < オプションn >)
plot2d (< 離散式 >)
plot2d ([< 式1 >, ..., < 式n >], < 定義域 >, < 値域 >, < オプションn >)
plot2d ([< 式1 >, ..., < 式n >], < 定義域 >, < 値域 >)
plot2d ([< 式1 >, ..., < 式n >], < 定義域 >)
```

plot2d 関数は二次元グラフの表示を行います. plot2d 関数で描ける式は, 特に指定しない場合, 1 変数を含むの Maxima の式ですが, 利用者が記述した一変数の Maxima の関数, 複素値関数で実部が 1 実数変数を含む式の実部, 関数を媒介変数を用いて記述したもの, そして, 離散的なデータの表示が可能です.

まず, 通常の Maxima の式を描画する場合, その変数の定義域が必要になります. 定義域は [x , < 下限 >, < 上限 >] で構成されたリストです. 又, グラフの縦方向の表示範囲を指定する値域もこれと似たリストですが, ここでの変数は y に固定されている為, [y , < 下限 >, < 上限 >] になります. 但し, 離散式に対する値域の指定は無効になります. 複素関数の実部を表示する場合は大域変数 plot_options の plot_realpart の項目を true に予めしておくか, [plot_realpart, true] をオプションとして引渡します. 猶, 大域変数 plot_options の詳細については後の小節で詳細を述べます.

次に, 媒介変数式の書式は以下の二通りの書式が許容されます.

媒介変数式の書式

```
[parametric, < X 座標 >, < Y 座標 >, < オプション1 >, ..., < オプションn >]
[parametric, < X 座標 >, < Y 座標 >]
```

まず, X と Y 座標は変数 t を唯一の変数として持つ式でなければなりません. 因に, この書式で媒介変数は t に固定されています. その為, 半径 1 の円を描く場合, [parametric, cos(t), sin(t)] と表記します. オプションは通常の plot2d 関数のオプションが入れられます. 猶, 媒介変数 t の定義域は, 媒介変数式の中に入れても, plot2d 関数で描く式の定義域として与

えても構いません. 図 10.1 に

`plot2d([parametric,cos(2*%pi*t/6),sin(2*%pi*t/6)])` の結果を示しておきましょう.

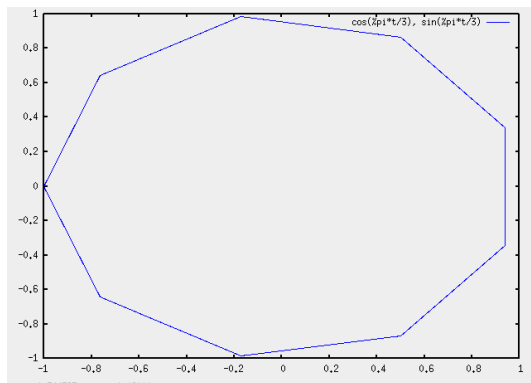


図 10.1: `plot2d([parametric,cos(2*%pi*t/6),sin(2*%pi*t/6)])` の結果

猶, t の定義域を省略する事も可能です. これは大域変数 `plot_options` に予め t の定義域が `[t,-3,3]` で設定されている為です. ここで t の定義域を変更したい場合は, 大域変数 `plot_options` の項目 t の定義域する方法と `plot2d` 関数で定義域を指定する方法があります. 又, 媒介変数式でグラフの粗さが目立つ場合には, 大域変数 `plot_options` の項目の `nticks` を 10 よりも大きな値, 例えば, 100 にするか, 或いは, `[nticks,100]` の様に `plot2d` 関数のオプションとして引渡します.

`plot2d` 関数は Maxima の式だけではなく, 点列の描画も可能です. 点列の場合は, 媒介変数式と似た書式になります. この場合, 二つの書式が可能です.

離散式の書式

`[discrete, <X 成分リスト>, <Y 成分リスト>]`

`[discrete, [[<X 成分1>, <Y 成分1>], ..., [<X 成分n>, <Y 成分n>]]]`

先ず, 双方の書式共に `discrete` で開始し, それから, X 成分のリストとそれに対応する Y 成分のリストか, X 成分と Y 成分の対, $[x_i, y_i]$ で構成されるリストの二種類になります. `plot2d` 関数は一度に複数の式のグラフを表示する事も可能です.

この場合, 表示する複数の式を 1 つのリストで与えます. ここで, リストに含まれる媒介変数式や離散式以外の Maxima の式は, 定義域が全て同じものでなければなりません. 媒介変数式を含む場合, その媒介変数式毎に定義域やその他のオプションが設定可能

です. 図 10.2 に媒介変数式を含む式リストの処理結果を示します.

```
plot2d([x^3+2,[parametric,cos(t),cos(t)*sin(t)]],
[x,-3,3],[y,-2,4],[t,-5,5],[nticks,100])
```

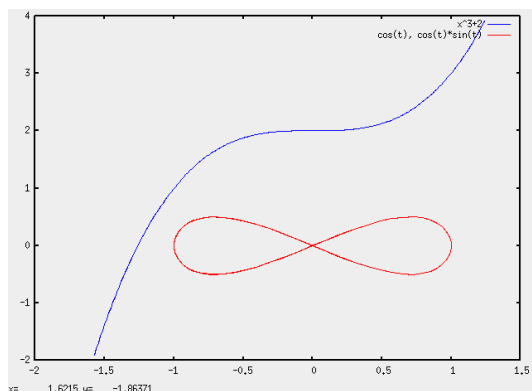


図 10.2: 媒介変数式と通常の式の混在

又, 複数の媒介変数式がある場合には, 媒介変数式内部にオプションを持たせて媒介変数式毎に設定を行う事が可能です.

```
plot2d([[parametric,cos(t),sin(t),[t,-%pi,%pi],[nticks,20]],
[parametric,2*cos(t),sin(-t),[t,-%pi/2,%pi/4],[nticks,5]]]);
```

この例では, 最初の半径 1 の円周の描画を 20 個の点で描画しますが, その次の楕円の描画では, 5 点で描画しています. ここでは, nticks の位置に注意して下さい. 即ち, 各々を媒介変数式の中に記述しているので, その影響は個々の媒介変数式にのみ影響されます.

では以下の様に変更するとどうなるでしょうか?

```
plot2d([[parametric,cos(t),sin(t),[t,-%pi,%pi],[nticks,20]],
[parametric,2*cos(t),sin(-t),[t,-%pi/2,%pi/4],[nticks,5]]],
[nticks,40]);
```

この場合, 媒介変数式内部の設定の方が優先される為に, 外の `[nticks,40]` は無視され, 結果として図 10.3 と同じ絵になります.

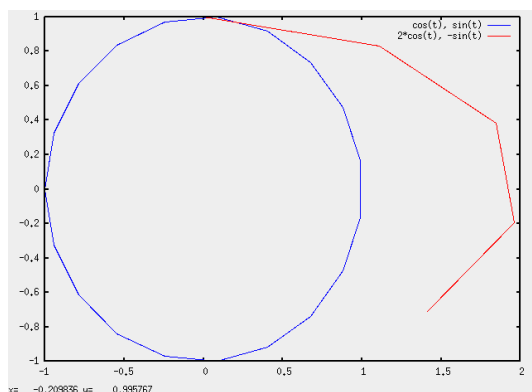


図 10.3: 複数の媒介変数式

ところで, `plot_format` で `openmath` を指定する場合, 複数の媒介変数式の中に定義域を記述した時に描画が上手く出来ない事があります. その為, `plot2d` 函数を用いる場合は, `plot_format` をデフォルトの `gnuplot` にしたままの方が無難でしょう.

さて, `plot2d` 函数のオプションは大域変数 `plot_options` を構成するリストでもあります. この大域変数 `plot_options` の諸項目リストを設定する事で, グラフのタイトルやラベル, X, Y 軸のラベル等の細かなグラフの指定が行えます. 特に, `gnuplot` を用いる場合, 即ち, `plot_format` が `gnuplot`, 或いはオプションとして `[plot_format, gnuplot]` を指定した場合, `plot_options` の `gnuplot_preamble` を上手く使う事で, `gnuplot` の命令文を実行させる事が可能です. この大域変数 `plot_options` には多くの説明すべき事項がある為, 次の節で詳細を述べる事とします.

10.1.6 plot3d 函数

Maxima では $f(x, y)$ の形式の 2 変数の式の描画が可能です. 更に, 一変数複素数値函数の実部 $f(x + iy)$ のみの表示も可能です.

但し, $x^2 + y^2 + z^3 = 1$ の様な式のグラフはそのままの形式では描けません. `plot3d` 函数の構文は基本的に `plot2d` 函数の構文を 3 次元にそのまま拡張したものになります.

plot3d の構文

```

plot3d (< 式 >, < 定義域1 >, < 定義域2 >, < オプション1 >, ..., < オプションn >)
plot3d (< 式 >, < 定義域1 >, < 定義域2 >)
plot3d ([< 式1 >, < 式2 >, < 式3 >], < 定義域1 >, < 定義域2 >, < オプション1 >, ..., < オプションn >)
plot3d ([< 式1 >, < 式2 >, < 式3 >], < 定義域1 >, < 定義域2 >)
plot3d ([< 式1 >, < 式2 >, < 式3 >], < 定義域1 >, < 定義域2 >, < 定義域3 >, < オプション1 >, ..., < オプションn >, [transform_xy, < 関数 >])

```

plot3d 関数は plot2d 関数と似た構文を持っていますが、plot2d 関数とは違い、複数の曲面を同時に描く事は出来ません。又、gnuplot で曲面を描く場合、pm3d の設定を行うと曲面を張りますが、無設定の場合はワイヤフレーム表示になります。詳細は、大域変数 plot_options の gnuplot_pm3d の項目を参照して下さい。

plot3d 関数は三次元空間内の曲面だけではなく、空間曲線も描けます。ここで空間曲線を描く場合は、x,y,z 成分の各成分を、< 定義域₁ > か < 定義域₂ > の変数として記述します。極座標系の場合も空間曲線を描く方法に似ていますが、この場合には、大域変数 plot_options の項目 transform_xy に座標変換を行う関数を指定する必要があります。ここで通常のデカルト座標から極座標への変換は単純に polar_to_xy を指定するだけで済みます。但し、利用者独自の座標変換関数の指定が必要であれば、make_transform 関数を併用する必要があります。この詳細は、大域変数 plot_options の transform_xy の個所で述べます。

10.2 大域変数 plot_options

Maxima はグラフ表示に gnuplot, openmath 等の外部アプリケーションを用います。アプリケーションに様々な設定を行いたい場合にはどうすればよいでしょうか？先程の実例に示した様に二つの方法がありました。先ず、通常の描画を行う plot2d 関数や plot3d 関数を用いる場合にオプションとして引渡す方法、そして、セッションを通じて設定しておきたい場合に用いる、set_plot_option 関数による大域変数 plot_options の内容変更を行う方法です。

この小節では、大域変数 plot_options の解説を行いましょう。

先ず最初に plot_options の値を見てみましょう。これは直接 `plot_options;` と入力すれば見る事が出来ます。例えば、Maxima-5.10.0 で入力した例を以下に示しておきましょう。

```
(%i8) plot_options;
(%o8) [[x, - 1.75555970201398E+305, 1.75555970201398E+305],
[y, - 1.75555970201398E+305, 1.75555970201398E+305], [t, - 3, 3],
[grid, 30, 30], [transform_xy, false], [run_viewer, true],
[plot_format, gnuplot_pipes], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]], [gnuplot_curve_styles,
[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,
with lines 6, with lines 7]], [gnuplot_default_term_command,
set term x11 font "Helvetica,16"], [gnuplot_dumb_term_command,
set term dumb 79 22], [gnuplot_ps_term_command,
set size 1.5, 1.5;set term postscript eps enhanced color solid 24],
[gnuplot_pipes_term, x11], [plot_realpart, false]]
```

ここで示す様に大域変数 plot_options は二成分リストから構成されたりストです。この大域変数を構成する二成分リストは [plot_format,gnuplot] の様に先頭が plot_options の項目で、その後に項目の値が続きます。例えば,[plot_format,gnuplot] は外部アプリケーションが gnuplot であり、生成するファイルも gnuplot のデータファイルになる事を示しています。

この例では,plot_options と入力して全ての設定を表示しています。これでは流石に見難いので,get_plot_option 関数を用いると、大域変数 plot_options の必要な設定値だけを調べる事が出来ます。ここで,get_plot_option 関数は1つの項目だけを引数とし、項目とそれに対する値の二成分リストを返す関数です。

ここで大域変数 plot_options の各項目に対応する値を変更したい場合、代入の演算子:=を使って、このリストを一々全て記述する事はあまり頭の良い方法とは言えません。指定した項目を変更する目的の為,set_plot_option 関数が用意されています。この set_plot_option 関数の引数として、項目とその値で構成されるリストを引渡します。但し,set_plot_option 関数に指定可能な項目は一件に限られます。これらの関数の構文を以下に纏めておきましょう。

—— 大域変数 plot_options に関連する関数 ——

```
set_plot_options( [〈項目〉,〈値〉])
get_plot_option(〈項目〉)
```

では、これらの関数の実例を示しておきましょう。この実例では,plot_format の値を

get_plot_option 関数で取出し、この項目の値を set_plot_option 関数で openmath に変更させるといふものです。

```
(%i9) get_plot_option(plot_format);
(%o9) [plot_format, gnuplot_pipes]
(%i10) set_plot_option([plot_format, openmath]);
(%o10) [[x, - 1.75555970201398E+305, 1.75555970201398E+305],
[y, - 1.75555970201398E+305, 1.75555970201398E+305], [t, - 3, 3],
[grid, 30, 30], [transform_xy, false], [run_viewer, true],
[plot_format, openmath], [gnuplot_term, default], [gnuplot_out_file, false],
[nticks, 10], [adapt_depth, 10], [gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]], [gnuplot_curve_styles,
[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,
with lines 6, with lines 7]], [gnuplot_default_term_command,
set term x11 font "Helvetica,16"], [gnuplot_dumb_term_command,
set term dumb 79 22], [gnuplot_ps_term_command,
set size 1.5, 1.5;set term postscript eps enhanced color solid 24],
[gnuplot_pipes_term, x11], [plot_realpart, false]]
(%i11) get_plot_option(plot_format);
(%o11) [plot_format, openmath]
```

この例等で示した様に、大域変数 plot_options には色々な項目があります。ここでは、大域変数 plot_options の各項目について解説しましょう。そこで、外部アプリケーションに依存しない表示設定に関する項目を以下に示しましょう。

大域変数 plot_options の一般的な項目

項目	初期値	概要
plot_format	[plot_format, gnuplot]	グラフ表示アプリケーションを設定
run_viewer	[run_viewer, true]	true の場合に plot_format で指定したアプリケーションを起動
colour_z	[colour_z, false]	カラーの PS ファイルを出力するかどうかを指定するフラグ
view_direction	[view_direction, 1, 1, 1]	plot_format が ps の場合に三次元グラフの視点を指定
transform_xy	[transform_xy, false]	三次元グラフ表示で座標変換を行うかどうかを指定するフラグ
grid	[grid 30, 30]	三次元グラフの解像度を指定
nticks	[nticks, 10]	二次元グラフの解像度を指定
logx	[logx, false]	二次元グラフでの X 座標の対数目盛フラグ
logy	[logy, false]	二次元グラフでの Y 座標の対数目盛フラグ
plot_realpart	[plot_realpart, false]	複素関数実部の表示フラグ
adapt_depth	[adapt_depth, 10]	
x	[x,-a,a]	a はシステムによって異なる浮動小数点. 表示可能な領域の目安
y	[y,-a,a]	a はシステムによって異なる浮動小数点. 表示可能な領域の目安
t	[t,-3,3]	助変数表示に於ける助変数の定義域

Maxima は現在,gnuplot をデフォルトの描画用のアプリケーションとしています. そ

の為,gnuplot を制御する項目が沢山存在します.

gnuplot に関連する plot_options の項目		
項目	初期値	概要
gnuplot_pm3d	[gnuplot_pm3d, true]	gnuplot の PM3D(曲面に面を張るかどうか) のオプション
gnuplot_curve_titles	[gnuplot_curve_titles, [default]]	描画する曲線の表題を設定.
gnuplot_curve_styles	[gnuplot_curve_styles, [with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,with lines 6, with lines 7]]	曲線の様式を指定
gnuplot_out_file	[gnuplot_out_file, false]	gnuplot の画像出力ファイルを指定.
gnuplot_term	[gnuplot_term, default]	gnuplot の term を設定し, 対応するデータ出力する.
gnuplot_default_term_command	[gnuplot_default_term_command, set term x11 font "Helvetica,16"]	gnuplot で実行する term に関連する命令文を記述する
gnuplot_dumb_term_command	[gnuplot_dumb_term_command, set term dumb 79 22]	term を dumb とした場合の設定
gnuplot_ps_term_command	[gnuplot_ps_term_command, set size 1.5, 1.5; set term postscript eps enhanced color solid 24]	PS ファイルに追加する命令
gnuplot_pipes_term	x11	gnuplot の端末を指定
gnuplot_preamble	[gnuplot_preamble,]	gnuplot に引渡す諸設定を文字列で指定

この plot_options の各成分の値は set_plot_options 関数で設定する事も可能ですが, plot2d や plot3d 関数のオプションとして引渡す事も可能です. 又, Maxima-5.12.0 から gnuplot_default_term_command にフォントを Helvetica とし, サイズを 16 ポイントとする命令が初期値として入っています. 更に, plot_options では, 一般的なグラフ表示の為の設定の他に, gnuplot 専用の設定もあります. その場合には, gnuplot_xxx の様に頭に gnuplot が付くので分かり易いかと思います. それでは各成分の意味について解説しましょう.

plot_format

plot_format で Maxima がグラフ表示で用いる外部アプリケーションを指定します。指定可能な値は gnuplot_pipes,gnuplot,openmath,geomview,ps と zic です。

Maxima-5.12.0 から plot_format の初期値として gnuplot が指定されています。描画に用いる外部アプリケーションは基本的にここでの値と同じ名前のアプリケーションになりますが,ps を指定した場合だけは ghostview がグラフ表示用の外部アプリケーションとして設定されます。

ここで gnuplot,openmath と geomview については今迄色々と解説しているので問題は無いと思います.mgnuplot は Tcl/Tk で記述したフロントエンドを gnuplot に被せたものです。こちらは三次元空間内の曲面をスライドバーを使って回転させる事が出来ますが、基本的に gnuplot を操作する plot_option の項目は反映されません.mgnuplot は基本的に MS-Windows 環境で用いられます。そして、大域変数 gnuplot_command に gnuplot を用いるのか,mgnuplot を用いるのかが設定されています。

最後の zic は Izic を外部アプリケーションとするものです。この Izic 自体は Tcl/Tk でフロントエンドを記述した古いアプリケーションの為、実際に使われる事は無いでしょう。

Maxima には openmath も同梱されているので、指定すれば使えますが,ps(ghostview),geomview,mgnuplot に Izic になると、各々のアプリケーションが利用可能な環境でなければ使えません。

ここで Maxima の plot2d 関数と plot3d 関数は,plot_format で指定したアプリケーション向けのデータファイルを UNIX 環境の場合は、利用者のホームディレクトリに生成します。それから run_viewer の値が true であれば、そのファイルを指定した外部アプリケーションに引渡し、この値が false の場合、外部アプリケーションの立上げを行いません。

猶、ここで生成されるデータファイルの名前は maxout.gnuplot の様に maxout. の後に plot_format の値が付いたファイル名になります。又、続けて描画を行う場合、データファイルをそのまま上書きします。

このファイルは各アプリケーションから開く事も可能です。初期値の gnuplot の場合、データの他に描画命令やオプションの諸設定も記述されたファイルとなっているので、単純に,gnuplot から `load 'maxout.gnuplot'` で読み込むと、グラフの表示を行います。この時、マウスが ON になっていればマウスで直接三次元画像を把持して、回転や拡大が行えます。もしも、マウスが on でなければ、`set mouse` で on 出来ます。

mgnuplot の場合は,gnuplot 向けの曲線、或いは曲面のデータのみが含まれたファイルになります。その為,plot2d 関数の場合は `plot 'maxout.mgnuplot'`,plot3d 関数の場

合, `splot 'maxout.mgnuplot'` を実行すれば描画を行います.

colour_z

`colour_z` は, カラーの PostScript ファイルを出力するかどうかを指定する項目です. `true` の場合にカラーの PostScript ファイル出力を行い, `false` の場合は白黒の PostScript ファイルを出力します. この `colour_z` は `plot_format` として `ps` が指定された場合のみに有効です.

view_direction

`view_direction` は視点の位置を指定します. この指定は `plot_format` が `ps` の場合の時だけ有効です.

transform_xy と make_transform 関数

`transform_xy` は通常のデカルト座標系から別の座標への変換を指定します. ここで設定する値は, 通常のデカルト座標系を用いる場合に初期値の `false`, 極座標系を用いる場合には `polar_to_xy` を設定します. この `transform_xy` に利用者定義の座標変換を利用する事も可能です. この場合は, `make_transform` 関数を用いて指定しなければなりません.

以下に `make_transform` 関数の構文を示しておきます.

— `make_transform` 関数の構文 —

```
make_transform((〈変数リスト〉, 〈 $f_x$ 〉, 〈 $f_y$ 〉, 〈 $f_z$ 〉))
```

ここで f_x, f_y, f_z は各々デカルト座標系の X, Y, Z 成分となる 〈変数リスト〉に含まれる変数の関数です. 例えば, 通常の円筒座標であれば, `make_transform([r, th, z], r*cos(th), r*sin(th), z)` の様にします. 実際に利用する場合は `make_transform` 関数を用いた利用者定義の関数を生成し, それを `transform_xy` に設定するか, `make_transform` 関数をそのまま設定するか何れかが使えます. 但し, 利用者定義の変換関数を用いる場合には `polar_to_xy` の様に関数名だけを指定する事は出来ず, 変数も一緒に書かなければなりません. これだけでは分かり難いので以下に実例を示しておきます.

```
(%i6) neko(r,th,z):=make_transform([r,th,z],
r*cos(%pi*th/180),r*sin(%pi*th/180),z);

                                     %pi th
(%o6) neko(r, th, z) := make_transform([r, th, z], r cos(-----),
                                     180
```

```

                                %pi th
                                r sin(-----), z)
                                180

(%i7) plot3d(r*th^2, [r,1,2], [th,0,360], [gnuplot_pm3d,true],
[transform_xy,neko(r,th,z)]);

```

この例では, plot3d 関数のオプションとして [transform_xy,neko(r,th,z)] を引渡し
ていますが, このオプションを [transform_,make_transform([r,th,z], r*cos(%pi*th/180),r*sin(%pi*th/180),z)]
としても同じ結果になります. ここで定義した関数は, 通常の polar_to_xy では角度が
ラジアンになるのに対し, こちらは度を用いる様にしている点です.

ここで注意点として, [transform_xy,neko] の様に利用者定義の変換関数に対し, その
関数名だけを記述する事は出来ません. 必ず, [transform_xy,neko(r,th,z)] の様に
変数も含めて記述しなければなりません. 但し, polar_to_xy に関しては, [transform_xy,polar_to_xy]
と変数を省略して表記出来ます. これは, polar_to_xy の実体が plot.lisp 内部で定義さ
れた LISP の関数の為だからです.

では, 上記の実行結果を以下に示しておきましょう.

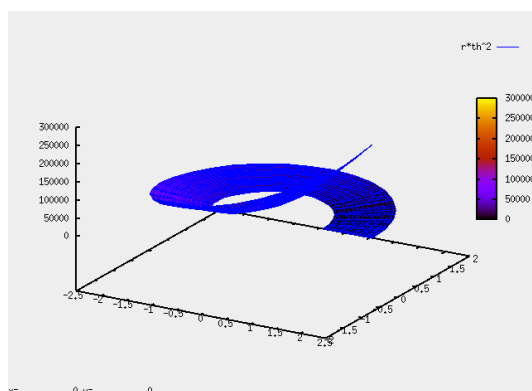


図 10.4: transform_xy と make_transform の組合せ

nticks

nticks は二次元グラフの解像度になります. 初期値は 10 です. 描いた曲線が粗い
場合に, この nticks の値を大きくすると良いでしょう. 特に媒介変数式の表示では初
期値を予め大きく指定しなければ綺麗な曲線は描けないでしょう.

grid

grid は三次元グラフの解像度で, 初期値は [30,30] です.grid の値は X 方向と Y 方向の解像度の対で記述し,[grid,50] の様に記述する事は出来ません. 必ず,[grid,50,50] の様に記述します. 描いた曲線が粗い場合, この grid に大き目の整数値を成分に持つリストを設定します.

x と y

x と y に設定される値は Maxima の土台にある Common Lisp 処理系の (/ most-positive-double-float 1024) の処理結果となります. この値は Common Lisp で扱える数値の限界を示すものの為, この値を越える数値は Maxima では扱えないので, 当然, グラフ表示も出来ません. 猶, グラフ表示可能な数値の上限は Maxima で利用可能なグラフ表示アプリケーション毎に異なる為, ここでの設定以下であっても表示が出来ない場合があります. 例えば,gnuplot では, $\sqrt{x}$ のグラフをこの x の値以下であれば表示が可能ですが,openmath では 10^{203} よりも大きな数値になると表示がなくなります.

t

これは関数の媒介変数式で利用する変数 t の定義域を定めます.Maxima では媒介変数式の変数は t に固定されています. この定義域を定めておけば, グラフ表示で媒介変数の定義域を省略する事が可能になります.

xlog と ylog

xlog と ylog は各々 X 軸と Y 軸の目盛を対数目盛にします. この設定は二次元グラフの場合に有効で, 三次元グラフの場合には無効です.

gnuplot_pm3d

gnuplot_pm3d は gnuplot の pm3d を有効にします. 初期値は false の為,gnuplot で描く曲面はワイヤフレーム表示となっていますが,true を設定した場合,maxout.gnuplot に set pm3d を書込みます. 又, 必要に応じて,b,s,t といった文字で構成された文字列も与えられます. これらの文字にはそれ相当の意味がありますが, この詳細については gnuplot の pm3d の 10.5.5 節で改めて解説します.

実例を以下に示しておきましょう。

pm3d の例

```
plot3d(sin(x*y),[x,-3,3],[y,-3,3],[gnuplot_pm3d,true],[grid,50,40])
```

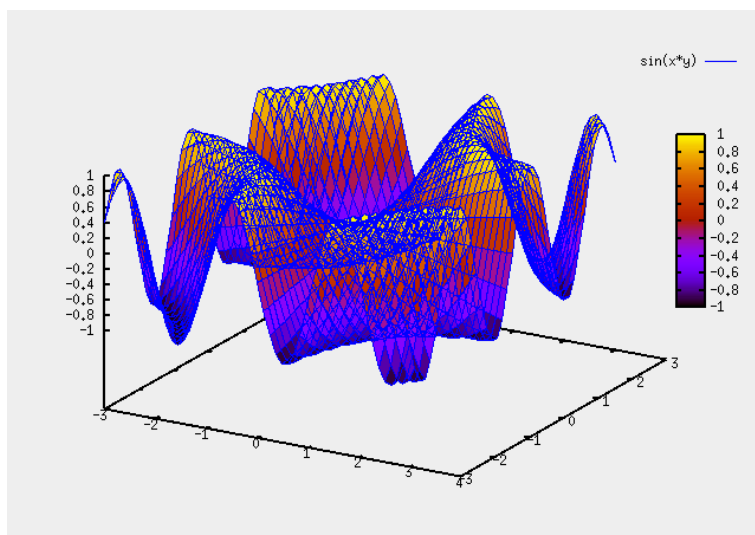


図 10.5: pm3d の例

しかし,gnuplot ではより高度な表示が可能です.set pm3dにしても, `set pm3d at bs` とする事で,最初に底面への曲面の投影を行い,それから曲面を描くといった指定が行えます. この様な細かな設定は, gnuplot_preamble を用いればより詳細に行えますが,gnuplot_pm3d に引渡す事も可能です. 例えば, 底面への投影を意味する文字 b と曲面の描画を意味する s を組合せた文字の列 bs を Maxima の文字列にせず,[gnuplot_pm3d,bs] の様にして引渡せます. 以下に例を示しておきましょう.

gnuplot_pm3d の設定例

```
plot3d(sin(x*y),[x,-3,3],[y,-3,3],
[gnuplot_pm3d,bs],[gnuplot_preamble,"unset surf"],[grid,50,40])
```

この例では,gnuplot の pm3d のオプションとして bs の他に,gnuplot_preamble を用いて,gnuplot に `unset surf` を実行させます. この設定は, 曲面上の網目を消す効果があります. この式を実行して得られた図を図 10.6 に示しておきます.

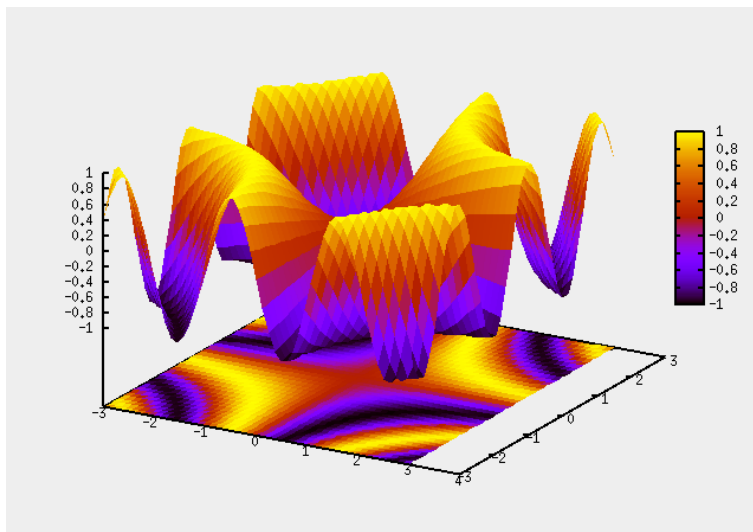


図 10.6: gnuplot_preamble を使った例

猶, 図 10.6 は次の様に gnuplot_preamble をだけを用いても描けます.

—— gnuplot_preamble を使った例 ——

```
plot3d(sin(x*y), [x, -3, 3], [y, -3, 3],
[gnuplot_preamble, "set pm3d at bs; unset surf"], [grid, 50, 40])
```

gnuplot_term

gnuplot_term は gnuplot の出力端末の設定を行います. これは指定した端末に対応したデータを出力するもので, gnuplot 内部では set term による端末の設定が行われます. ここで設定可能な端末の型には, x 端末に対応する X11, MacOS の aqua と gnuplot の画像を表示する際に必要な端末の情報や, postscript や gif といった画像データ等と非常に多い為, ここでは一々説明しません. この端末の詳細は参考文献 [39] か, 或いは gnuplot 上で `? term` と入力すれば, オンラインマニュアルと term に設定可能な値の一覧が表示されるので, そちらを参照して下さい.

猶, gnuplot_term に 'default' 以外の値を設定した場合に Maxima で描画を行うと, gnuplot に引渡すデータファイル (maxout.gnuplot, 或いは maxout.gnuplot_pipes) に加えて, 別途, グラフ画像のデータファイルが出力されます. この場合, maxplot.< 端末名 >

といった名前のファイルになります。例えば,terminal として tgif を設定した場合で解説しましょう。

この時,グラフ出力時に,set term tgif と set out '/home/yokota/maxplot.tgif/' (ここで/home/yokota/がホームディレクトリです) が gnuplot に設定されます。そして,同時にグラフ出力ファイルとして,maxplot.tgif が生成され,画像データが書込まれます。猶,maxplot.tgif を maxplot.obj に変更して tgif に読込ませた結果が図 10.7 に示すものです。

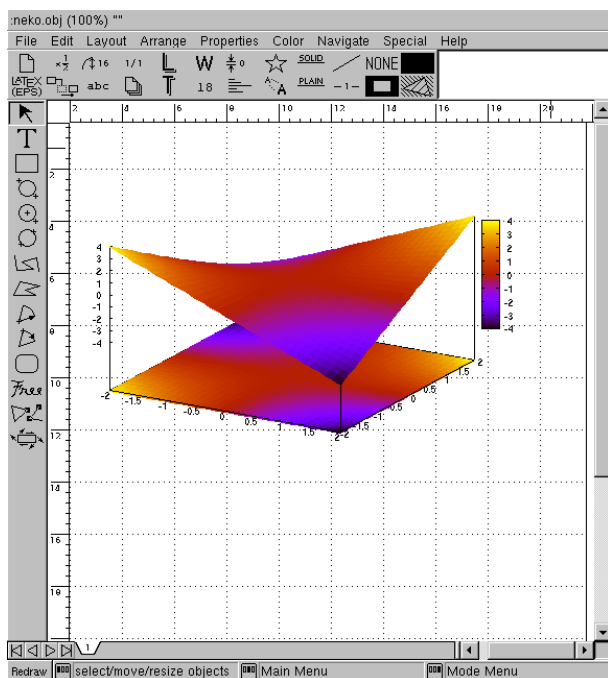
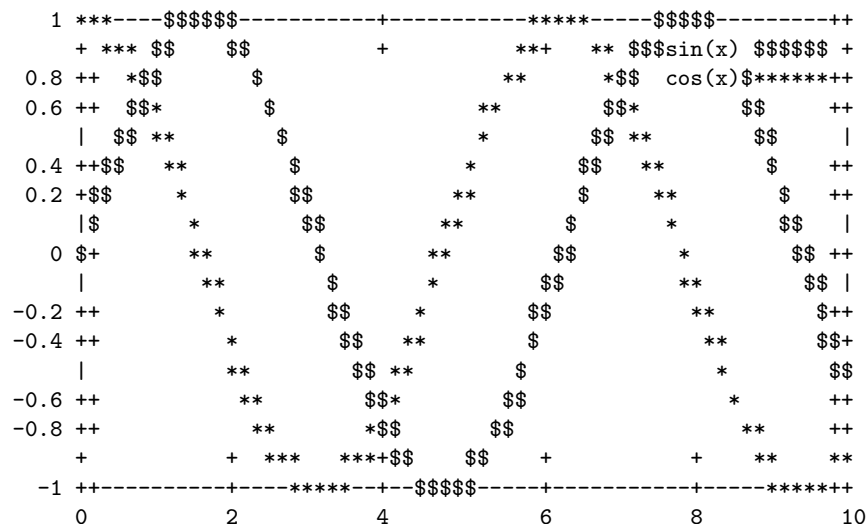


図 10.7: term を tgif にした結果 (maxplot.tgif) を tgif で表示

gnuplot_dumb_term_command

gnuplot_dumb_term_command は gnuplot の term として dumb を選択した時の設定が行えます。以下に実例を示しておきます。

```
(%i86) plot2d([sin(x),cos(x)], [x,0,10], [gnuplot_term,dumb],
[gnuplot_dumb_term_command,["set term dumb 70 20"]]);
```



```
Output file "/home/yokota/maxplot.dumb".
(%o86)
```

この例では,dumb 端末を 70 列 20 行として表示しています。猶,term を dumb にしなければ,この設定は無効です。

gnuplot_curve_titles

gnuplot_curve_titles には描く曲線や曲面の表題をリストで追加します。この時,構文は gnuplot の plot 命令や splot 命令に引渡す表題の設定に準じます。例えば,二曲線に A1,A2 と設定したい場合には,gnuplot_curve_titles,['title 'A1'",'title 'A2"']] とします。

gnuplot_curve_styles

gnuplot_curve_styles に描画する線分の書式を設定します。これは gnuplot の with 命令の文を文字列として引渡す為に用います。例えば,[gnuplot_curve_styles,['with lines

7", "with lines 2"] の様に行います. 因に, maxout.gnuplot 内部では, gnuplot の描画命令 plot のオプションとして引渡されます. その為, 後述の gnuplot_preamble で曲線のスタイルを幾ら設定しても, こちらの設定が優先されるので注意が必要です.

gnuplot_pipes_term

gnuplot_pipes_term で gnuplot が用いる端末 (terminal に相当) を指定します. この変数は, Maxima-5.12.0 より導入されています. 猶, MS-Windows 環境でも初期値として x11 が指定されています. 通常, この値を変更する必要は無いでしょう.

gnuplot_preamble

gnuplot_preamble は gnuplot の制御文を, gnuplot に直接引渡す為の仕組みで, gnuplot の制御文の間には, ; を入れた Maxima の文字列を指定します. この preamble の設定は plot_option で gnuplot に関連する設定と, gnuplot の数値データの上に置かれます.

preamble の内容がどの様に書込まれるかを plot_format を gnuplot に設定した状態で以下の描画をさせて, maxout.gnuplot の内容を調べてみましょう.

```
(%i4) nekoneko:"set title 'mike';set xlabel 'X';\
set ylabel 'Y';set zlabel 'height';";
(%o4) set title 'mike';set xlabel 'X';set ylabel 'Y';set zlabel 'height';
(%i5) plot3d(sin(x*y), [x,0,10], [y,0,10],
[gnuplot_pm3d,true], [gnuplot_preamble,nekoneko]);
```

では, 以下に maxout.gnuplot の頭の部分を示しておきます.

gnuplot_preamble を反映した maxout.gnuplot のヘッダ部分

```
set pm3d

set title 'mike';set xlabel 'X';set ylabel 'Y';set zlabel 'height';
splot '-' title 'sin(x*y)' with lines 3
0.0      0.0      0.0
0.3333333333333333      0.0      0.0
0.6666666666666666      0.0      0.0
1.      0.0      0.0
```

この様に gnuplot の splot 命令の直前に gnuplot_preamble の内容が書込まれている事が分りますね. この事を利用すれば gnuplot の諸設定を行う事が可能になるのです.

10.2.1 gnuplot に関連した関数

— gnuplot に関連した関数 —

```
gnuplot_start()  
gnuplot_restart()  
gnuplot_close()  
gnuplot_pipes  
gnuplot_reset()  
gnuplot_replot(〈 オプション 〉)
```

gnuplot_start 関数は引数を必要としない関数です。MS-Windows でこの関数を実行すると gnuplot が立ち上がります。

gnuplot_restart は内部的に gnuplot_close を実行し、それから gnuplot_start を実行するだけの関数です。その為、グラフを表示している場合には、グラフのウィンドウが消滅し、MS-Windows であれば、gnuplot のウィンドウが現われます。

gnuplot_reset 関数は引数を必要としない関数です。この関数は gnuplot で設定した諸設定を初期化してしまう関数です。この関数をグラフ表示中に実行すると、それ迄のマウス操作による変更は全て無効になり、最初の設定のグラフが表示されます。

gnuplot_replot 関数はその名前のとおり、gnuplot で再描画を行う関数です。 `gnuplot_reset();`

を最初に実行し、 `gnuplot_replot();` を実行すれば、それまでのマウスによる操作は無効になり、最初に描画したグラフが再描画されます。

10.3 その他の描画函数

Maxima の描画函数には上述の `plot2d` 函数や `plot3d` 函数の他に, 幾つかの描画函数があります. とは言え, どちらかと言えば補助的な函数が多く, それも `openmath` 専用や PostScript への出力のみといった機能や出力が限定されたものが殆どです. 最初に, 比較的汎用性がある `openplot_curves` 函数の解説をしましょう.

10.3.1 `openplot_curves`

`openplot_plot` 函数は大域変数 `plot_options` の `plot_format` とは無関係に `openmath` を用いて与えられた点列の描画を行う函数です. とは言え, オプションを変更する事で, 実数値一変数函数の描画も可能です.

— `openplot_curves` —

```
openplot_curves ([<点列リスト1>, ..., <点列リストn>])
openplot_curves ([<オプション1>, <点列リスト1>, ..., ..., <オプションn>, <点列リストn>])
openplot_curves (<点列リスト>)
openplot_curves (<オプション>, <点列リスト>)
openplot_curves (<xfun を含むオプション>)
```

ここでの点列リストの書式は, $[x_1, y_1, \dots, x_n, y_n]$ か $[[x_1, y_1], \dots, [x_n, y_n]]$ の二種類に限定されます.

`openplot_curves` のオプションは `plot_options` とは無関係で, `openmath` のメニューからも設定可能な事項になります. 又, 点列リストの直前にあるオプションがその点列の表示で用いられます.

`openplot_curves` の設定項目を以下に示しておきます.

openplot_curves の設定項目

項目	初期値	概要
xfun	なし	指定した変数 x を持つ式を指定
color	blue から開始	曲線の色を指定
plotpoints	0	直線や曲線上に点を打つ為のフラグ. 初期値は 0
linecolors	blue	直線/曲線の色を指定.
pointsize	0	点の大きさを指定
nolines	0	点列を繋ぐ線分表示の為のフラグ
bargraph	0	棒グラフ表示への切替の為のフラグ
xaxislabel	無指定	X 軸のラベルを指定
yaxislabel	無指定	Y 軸のラベルを指定

openplot_curves のオプションには独特の書式があります. これは plot_options に似たもので, `{項目 値}` の書式の文字列を空行で区切って, 全体を引用符で括ったリストになります. 例えば, 各点を表示して, その点の大きさを 6 にする場合には, `["{plotpoints 1} {pointsize 6}"]` をオプションとして与えます.

次の例では, 折線毎に点の大きさと, 折線と点の色を変更し, 各軸にラベルを設定しています.

```
openplot_curves(["{plotpoints 1} {pointsize 6} {color red}",
[1,2,3,4,5,1],
["{pointsize 10} {xaxislabel time} {yaxislabel neko} {color black}"],
[10,9,9,2,4,2]])
```

又, xfun を指定する事で, Maxima の式をグラフに追加する事も可能です. 先程の例に正弦関数を追加した例を以下に示しておきましょう.

```
openplot_curves(["{plotpoints 1} {pointsize 6} {color red}",
[1,2,3,4,5,1],
["{pointsize 10} {xaxislabel time} {yaxislabel neko} {color black}"],
[10,9,9,2,4,2],
["{xfun sin(x)} {color green} {plotpoints 1} {pointsize 1}"]])
```

ここで注意する事は, 描画する式の変数は x に限定される事です. 又, 関数の描画だけを行いたければ, 点列リストを持たないオプションのみの引数を与えます. 但し, open-

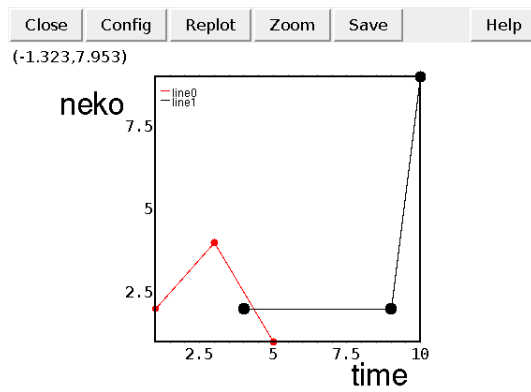


図 10.8: openplot_curves のオプション例

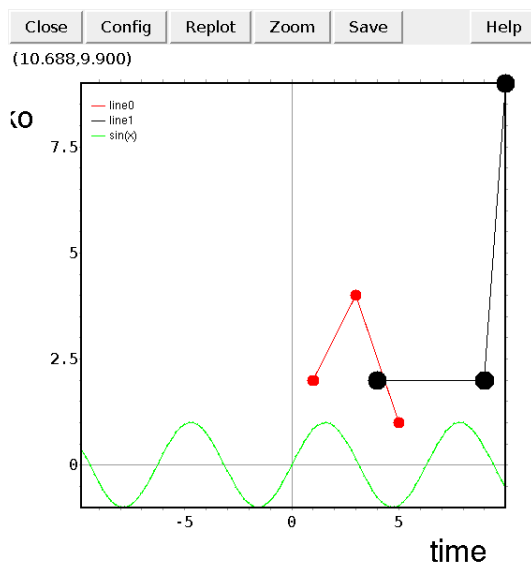


図 10.9: openplot_curves のオプション例 (その 2)

plot_curves 関数で式の定義域を与える事は出来ない様です.

10.3.2 contour_plot

contour_plot

```
contour_plot(< 式 >)
contour_plot(< 式 >, < オプション >)
```

contour_plot は与えられた 2 変数実数値関数の等高線を描画する関数です. オプションは plot_format と gnuplot_preamble に限定されます. 従って, グラフに様々な細工を行いたければ, plot_format は gnuplot でなければなりません.

10.3.3 Postscript に関連する関数

Postscript に関連する関数

```
plot2d_ps(< 式 >, < 定義域 >)
viewps(< PostScript ファイル名 >)
viewps()
psdraw_curve(< 点列リスト >)
psdraw_points(< 点列リスト >)
pscom(< PostScript 命令文 >)
closeps()
```

plot2d_ps 関数は plot2d 関数に似た構文を持ちますが, デカルト座標系での関数の表示のみが可能です. 具体的には関数と定義域の二つの引数のみを取ります. この関数はカレントディレクトリ上に maxout.ps ファイルを生成し, 内部で viewps 関数を呼出して, maxout.ps ファイルの表示を行います.

viewps 関数は単純に ghostview に PostScript ファイルを引渡す関数です. ファイル名を指定しない場合には, ホームディレクトリ上の maxout.ps ファイルの表示を行います. 猶, ghostview 命令を持たない環境では, この関数は使えません.

psdraw_curve は点列リストのグラフを PostScript 形式で出力します. ここで点列リストの書式は $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$ の書式と $[[x_1, y_1], [x_2, y_2], \dots, [x_n, y_n]]$ の書式の何れかになります. psdraw_curve 関数を実行する事でホームディレクトリ上の maxout.ps へのストリームを開きます. このストリームは `closeps()` で閉じられます.

`psdraw_points` 関数は点列リストのグラフを PostScript 形式のファイルで出力します。この関数は `psdraw_curve` 関数に似ていますが、この関数の引数となる点列リストの書式は $[[x_1, y_1], [x_2, y_2], \dots, [x_n, y_n]]$ のみに限定されます。`psdraw_curve` 関数と同様に `psdraw_points` 関数を実行するとホームディレクトリ上のファイル `maxout.ps` へのストリームが開かれます。又、`closeps()` を実行すると、ストリームが閉じられて、`maxout.ps` ファイルが生成されます。

`pscom` 関数は与えられた PostScript の命令文をホームディレクトリ上の `maxout.ps` へのストリームに出力する関数です。

`psdraw_curve` 関数、`psdraw_points` 関数と `pscom` 関数はどちらからと言えば原始的な関数で、これらは全てホームディレクトリ上の `maxout.ps` へのストリームを開き、そのストリームに対して出力を行う関数です。そして、`closeps()` 関数を実行する事で、ストリームが閉じられて、`maxout.ps` ファイルが生成されます。

因に、ストリームが開いているかどうかは、大域変数 `pstream` で調べられます。ストリームが開いていない場合に、`pstream` を入力すると結果として `false` が返り、開いている場合には、その情報を返します。

実際に、`pstream` の働きを見ておきましょう。

```
(%i32) psdraw_curve([[1,2],[3,3],[5,1]]);
(%o32)                                     false
(%i33) pstream;
(%o33)  #<OUTPUT BUFFERED FILE-STREAM CHARACTER /home/yokota/maxout.ps>
(%i34) closeps();
(%o34)                                     true
(%i35) pstream;
(%o35)                                     false
```

この例では、最初に `psdraw_curve` 関数を実行する事で、ホームディレクトリ (ここでは `/home/yokota`) 上のファイル `maxout.ps` へのストリームを開きます。この事は、`pstream` を見ると、`maxout.ps` へのストリームが開いている事が分ります。それから `closeps()` を実行した後に、`pstream` の値を見ると `false` になっているので、この事からストリームが閉じている事が分ります。

10.4 plot_option 以外の描画に関連する大域変数

Maxima の描画関数で最も重要な大域変数としては `plot_options` がありました. Maxima にはこの大域変数の他にも多くの大域変数があります. 但し, これらの変数はどちらかと言えば, Maxima の描画関数で内部的に用いる事を想定したものが多く, 用途も特殊なものが多いのが実情です. その為, ここでは一覧表に纏めて解説しておきます. 因に, 以下で解説する変数は Maxima のソースファイル `plot.lisp` の中を見れば色々ある事が分ります.

plot_option 以外の描画に関連する大域変数

変数名	初期値	概要
<code>in_netmath</code>	false	<code>plot_format</code> の値が <code>openmath</code> の場合に限り, <code>maxout.openmath</code> の内容を画面に表示
<code>show_openplot</code>	false	<code>plot_format</code> が <code>openmath</code> の場合は true, それ以外は false
<code>viewps_command</code>	(ghostview "a")	<code>plot_format</code> を <code>ps</code> にした場合の, PS ファイルの表示命令を設定
<code>ps_scale</code>	[72,72]	PS ファイルの解像度
<code>ps_translate</code>	[0,0]	原点の移動に利用
<code>window_size</code>	[612.0,792]	PS ファイルのウィンドウの大きさを指定

10.5 gnuplot による描画について

この章は Maxima を使う上で必要な gnuplot の事項のみを記述します。猶,gnuplot 単体の解説ではなく,Maxima から見た gnuplot の解説になります。この際に,maxout.gnuplot ファイルから gnuplot の事を開始します。

gnuplot 全般の使用法や例題を知りたいければ,「使いこなす GNUOPlot」 [39] 等の gnuplot の解説本や gnuplot に付属するマニュアル等の文献を参照して下さい。

10.5.1 maxout.gnuplot の内容について

maxout.gnuplot ファイルは plot.format を gnuplot に指定した場合に生成されるファイルで,UNIX 環境であればホームディレクトリ上,MS-Windows 環境であれば Documents and Settings 以下の個人フォルダに生成されます。

このファイルの先頭には数値データを描画する為の gnuplot の命令文や様々な設定文が置かれ,ファイルの末尾に描画される式の数値データが記述されています。その為,別途 gnuplot を起動させて `load "maxout.gnuplot"` と gnuplot の入力行に入力すると,gnuplot 単体で Maxima から処理したのと同じグラフが表示出来ます。

この事は,Maxima 側で行ったグラフの諸設定がどの様に maxout.gnuplot に反映されているかを観察すれば,gnuplot がどの様に処理を行っているかを理解する事が容易になるとも言えるのです。

そこで,手始めに plot2d 函数による maxout.gnuplot の構造を簡単に解説しておきましょう。

```

plot2d 函数による maxout.gnuplot
set log x      /* gnuplot_logx が true の場合に記述 */
set log y      /* gnuplot_logy が true の場合に記述 */

<gnuplot_preamble の内容>
plot '-' <gnuplot_curve_titles の内容> <gnuplot_curve_styles の内
容>
<実データ>

```

ここで示す様に gnuplot_logx と gnuplot_logy の設定は plot 命令の前に記述されます。更に,plot 命令の直前の行に,gnuplot_preamble の内容が書込まれます。そして,plot 命令の直後の行に描画するデータが置かれるのです。ここで,plot '-' とありますが,

ここでの `'-'` で次の行にある数値データを表示する事を `gnuplot` に指示しています。それから、曲線の表題や曲線の型の指定が並んでいます。

さて、今度は `plot3d` 函数による `maxout.gnuplot` の構造を示しておきましょう。 `plot3d` 函数で生成した `maxout.gnuplot` も `plot2d` 函数と似通った書式になります。

```
plot3d 函数による maxout.gnuplot

set pm3d          /* gnuplot_pm3d が true の場合に記述 */

<gnuplot_preamble の内容>
splot '- ' <gnuplot_curve_titles の内容> <gnuplot_curve_styles の内
容>
<実データ>
```

今度は `gnuplot_pm3d` に対応する `set pm3d` がファイルの先頭に置かれていますね。その他は `plot2d` 函数で生成した `maxout.gnuplot` と似ていますが、描画命令が `plot` ではなく `splot` 命令になっている事に注意しておいて下さい。

これらのファイルの内容を眺めるだけでも、`plot` 命令や `splot` 命令が `gnuplot` の描画命令である事が何となく分るかと思います。これらの描画命令の前に、`set` 命令から開始して、`xlog`、`ylog` や `pm3d` が続く命令文があります。これらは `gnuplot` で様々な設定を行う命令文で、先ず、`set` 命令で様々な設定を行います。そして設定内容は `show` 命令で確認が行え、`unset` 命令で設定内容を無効にする事が出来ます。

さて、Maxima から単純に `plot2d` 函数や `plot3d` 函数を用いて `maxout.gnuplot` を生成しても、

`set pm3d at bs` の様な `gnuplot` の命令文を埋込む事は出来ません。`gnuplot` から直接描画する場合は、この様な命令文を入れて `replot` 命令で再描画すれば良いのですが、Maxima から操作したければ、`plot_format` を `gnuplot_pipes` にした場合のみ有効になります。又、MS-Windows 版では、`gnuplot_pipes` が選べない為、この身軽な操作は出来ません。

その為、より高度なグラフ表示を行う場合、直接 `gnuplot` の命令文が書き込める `gnuplot_preamble` に色々と設定すると結構楽に処理が行えます。この事から、`gnuplot` を利用する場合、`gnuplot_preamble` の使いこなしが非常に重要である事が理解出来るかと思います。

それでは、先程の `maxout.gnuplot` に表われた命令の意味を簡単に紹介する事にしましょう。

10.5.2 set 命令

先ず gnuplot で様々な設定を行う上で必要な set 命令について簡単に説明しておきましょう.

この set 命令は前述の様に gnuplot の様々な設定を行う命令です. set 命令で設定した内容は show 命令を使って確認する事が可能です. 更に, set 命令による設定を無効にする場合は unset 命令を用います. これらの命令は非常に重要な命令です. 以下に, set 命令, show 命令と unset 命令の基本的な構文を示しておきます.

— set 命令, show 命令, unset 命令 —

```
set <項目>           <項目> を有効にする
set <項目> <設定内容> <項目> に <設定内容> を設定する
show <項目>          <項目> の内容を表示
unset <項目>         <項目> を無効にする
```

ここで示した様に set 命令には単純に項目を指定する場合と, その項目に対して値を指定する場合の二種類があります. この具体的な例は Maxima の諸設定と絡めて詳細を述べる事とします.

10.5.3 plot 命令による曲線の表示

先程の maxout.gnuplot の例で示した様に, 曲線を描く場合には plot 命令を用います. この plot 命令は変数 x の式や 2 次元の数値データを描画する事が可能です. 以下に gnuplot での plot 命令の構文を示しておきます.

— plot 命令の基本的な構文 —

```
plot <表示範囲> <式> title <文字列> with <曲線の様式>
plot <式> axes <軸の設定> title <文字列> with <曲線の様式>
plot <式> title <文字列>
plot <式>
plot <データファイル名> title <文字列> with langle 曲線の様式
plot <データファイル名> title <文字列>
plot <データファイル名>
```

式の表示範囲は複数の式を表示する場合でも, 一つの plot 函数に対し, 一つだけが設定可能です. ここで表示範囲は `plot [-3:3] [-2:2] cos(x)` の様に定義域と値域の

順番で MATLAB 風のベクトル表記で記述します。ここで `plot [-3:3]` の様に `[-3:3]` だけを設定した場合、`[-3:3]` が式の定義域となり、グラフの値域は式全体が収まる様に gnuplot 側で自動的に調整します。次に gnuplot の式には gnuplot で表記可能な数式、或いは表示すべきデータが含まれているデータファイル名や記号 `'-'` を設定します。ここで plot 命令に与える数式は基本的に変数 x の gnuplot の式に限定されます。複数の式を同時に表示させる場合、式とその式に対する設定を一組として、それらをコンマで区切ったものを与えます。例えば、正弦函数と余弦函数を同時に表示したい場合、最も簡単な表示の場合は `plot sin(x),cos(x)`、曲線の名前や曲線の様式を指定する場合、以下の様に入力します。

```
plot sin(x) title 'Sin' with lines 1,cos(x) title 'Cos' with points 5
```

データファイル名は引用符、或いは二重引用符で括られた文字列として与えます。猶、Maxima で gnuplot で描画させるファイル名やラベルの指定の為に文字列を gnuplot の命令文に組込む際に注意する事として、Maxima の文字列が二重引用符に限定される為、ファイル名やラベル等の gnuplot の命令文中の文字列は単引用符で括ったものを用いなければならない事です。例えば、曲線の表題を `test` にしたければ、Maxima の大域変数 `plot_options` の項目 `gnuplot_curve_titles` への設定を `[gnuplot_curve_titles,"title 'test'"]` で行います。即ち、二重引用符で括った場合、二重引用符内部の文字の列に二重引用符が出現すると、文字列がそこで一旦途切れてしまいますが、単引用符の場合は途切れずに gnuplot に引渡される為です。

あるファイルに plot 命令とデータ本体を記載する場合、plot 命令で指定するファイル名の箇所を `'-'` と記述すれば、plot 命令の次の行から開始するデータ列の読込を行い、ファイルの EOF を検出した時点で描画を行います。この方法が `maxout.gnuplot` で採用されています。

gnuplot の式とデータファイルの内容を混在して表示する事も可能です。この場合、引数として、数式とデータファイルをコンマで区切ったものを第一引数として引き渡します。例えば、先程の正弦函数と余弦函数に加え、データファイル `test1` のデータを表示させたい場合、`plot sin(x),cos(x),'test1'` の様に入力します。

曲線の名前は `title` の後に文字列を指定します。これは gnuplot の右上に表示される凡例 (key) で用いられる曲線の名前です。Maxima では大域変数 `plot_options` の項目 `gnuplot_curve_titles` の内容がここに記入されます。因にこの凡例を調整する場合には `set key` を用いて行います。

ここで、Maxima から gnuplot を利用する場合、この `gnuplot_curve_titles` の内容が標準入力を示す記号 `'-'` の直後に置かれます。そこで、この事を利用した阿漕な使い方とし

では, 別のグラフデータを指定したり, gnuplot の式を入れる事が挙げられます. この事を早速確認してみましょう.

```
plot2d(1/x, [x, 1.0e-8, 1.0e-5],
[gnuplot_curve_titles, "title 'Maxima', 1/x title 'GNUPlot' "]);
```

この例では, $\frac{1}{x}$ のグラフを描きますが, 一方は Maxima で描いたもので, もう一方は gnuplot の式として gnuplot に計算させて描画するものです. その結果を図 10.10 に示しておきます.

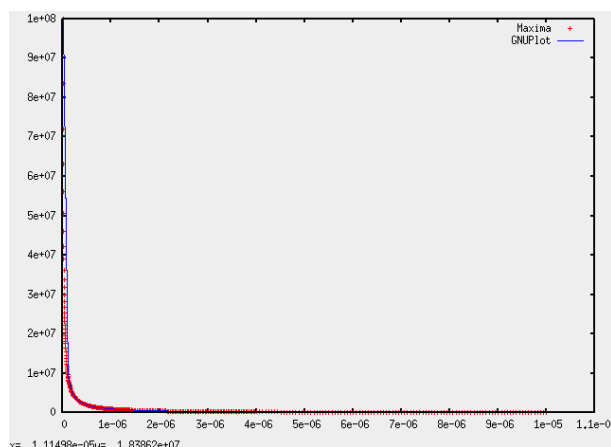


図 10.10: gnuplot_curve_titles の応用で gnuplot の式も描く

ここで, gnuplot の式は `[gnuplot_curve_titles, "title 'Maxima', 1/x title 'GNUPlot' "]` で与えている事に注目して下さい. この処理で生成された maxout.gnuplot のヘッダの部分を下に示しておきますが, 上手く埋め込まれている事が分ります.

maxout.gnuplot の様子

```
plot '-' title 'Maxima',1/x title 'GNUPlot' with lines 3
1.00000000E-8 100000000.
1.195117187500000100E-8 83673802.90897205
1.390234375000000400000000E-8 71930317.5049171
1.585351562500000600000000E-8 63077491.68411973
1.780468750000000500000000E-8 56164984.642386995
1.975585937500000700000000E-8 50617894.216510125
2.170703125000000600000000E-8 46068022.3141983
2.365820312500000500000000E-8 42268637.001568556
2.560937500000000600000000E-8 39048200.12202562
---- 以下略 ----
```

最後に、曲線の書式は with の後に記入します。Maxima では gnuplot_curve_styles に with から開始する文字列を設定しておけば、maxout.gnuplot にその文字列が記入されます。通常の実線で色を赤にする為には、with lines 1 の様に設定します。

10.5.4 splot 命令による曲面の表示

splot 命令は数値データや変数 x, y の式で表現された曲面を描く命令です。基本的な構文は plot 命令と同様で、splot を単純に 3 次元に拡張した面を持ちます。

splot 命令の基本的な構文

```
splot < 式 > title < 文字列 > with langle 曲面の様式 >
splot < 式 > title < 文字列 >
splot < 式 >
splot < データファイル名 > title < 文字列 > with langle 曲面の様式 >
splot < データファイル名 > title < 文字列 >
splot < データファイル名 >
```

この splot 命令でも plot 命令と同様に曲面の表題 (title で指定) , そして曲面の様式 (実際はワイヤーフレームを構成する線分の様式に対応) の指定が行えます。ファイル名や文字列に関しては、plot 命令と違いはありません。又、gnuplot で扱える式は変数 x, y の式でなければなりません。

猶, 曲面の様式には, lines, point, linespoints, dots, impulses があります. この曲面の様式は曲面上に表示される網目に対するもので, 後述の pm3d に関連する項目に影響を及ぼしません.

gnuplot の 3.8 版より pm3d が標準になった事で, pm3d の様々な設定を色々行う必要が出てくるでしょう. 次に, この pm3d の基本的な設定について述べる事とします.

10.5.5 pm3d

gnuplot では pm3d を有効にする事によって, 面を貼った曲面表示が行えます. この pm3d は従来は gnuplot のパッチとして配布されていましたが, 3.8 以降から標準で付属しています.

pm3d の設定

```
set pm3d set pm3d at b
set pm3d at s
set pm3d at t
set pm3d map
unset pm3d
show pm3d
```

最初の `set pm3d` で pm3d を有効にします. 因に, Maxima の `[gnuplot_pm3d, true]` がこの設定に対応します. Maxima の大域変数 `plot_options` には, その他に pm3d に関する項目はありませんが, 他にも細かな指定が出来ます. 例えば, 面を張った曲面の表示だけではなく, 曲面と底面や上部への投影と一緒に表示したりする事が可能です. この投影の設定は pm3d の設定にて at の後の文字列で指定を行います.

次の `set pm3d at b` で曲面を底面に投影します. 後述の map に似ていますが, こちらは斜め上から曲面と投影の両方を眺める形になります. この指定だけでならば曲面はワイヤーフレーム表示になります.

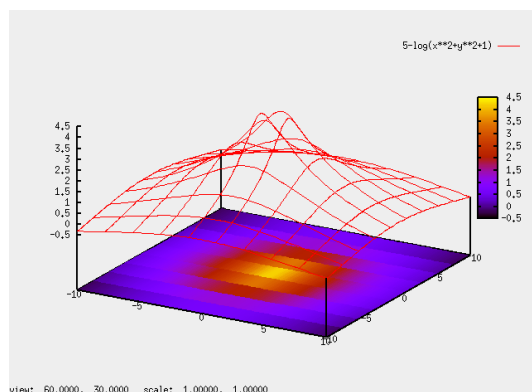


図 10.11: set pm3d at b の場合

これに対し、`set pm3d at s`を実行すると、今度は曲面に面を張り、この面には連続的に等高線で変化する色彩が付けられた曲面が表示されます。この場合、曲面の底面への射影は実行されません。又、gnuplot で hidden3d を有効にしたり、surface を無効にしていなければ、曲面とワイヤーフレームが同時に表示されています。

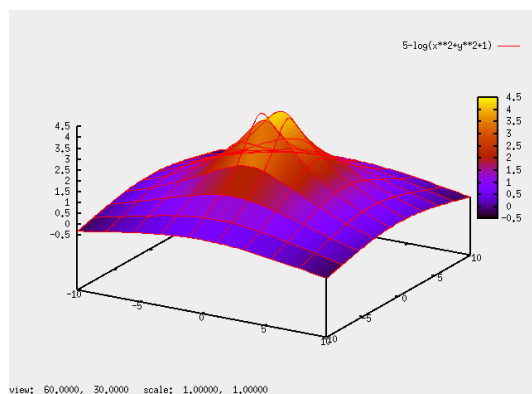


図 10.12: set pm3d at s の場合

gnuplot では底面だけに射影が行える訳ではありません。 `set pm3d at t` とする事で, 曲面の等高線表示の射影を今度は上面に行います。 . この場合, 射影が上側に行われる事を除いて, `b` を指定した場合との違いはありません (勿論, 射影が上に出る違いはあります…).

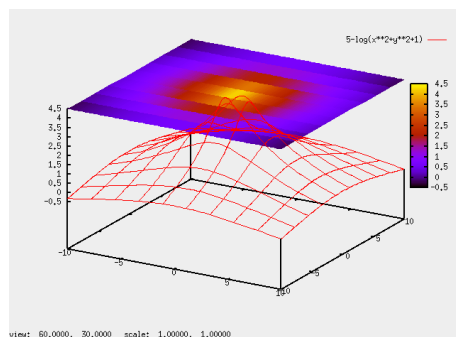


図 10.13: `set pm3d at t` の場合

この様に上や下側への射影を行うと, 射影には綺麗な等高線図が得られます。では, この等高線図だけが欲しい場合はどうすれば良いのでしょうか? それは `set pm3d map` で得られます。次の図 10.14 に示しておきましょう。

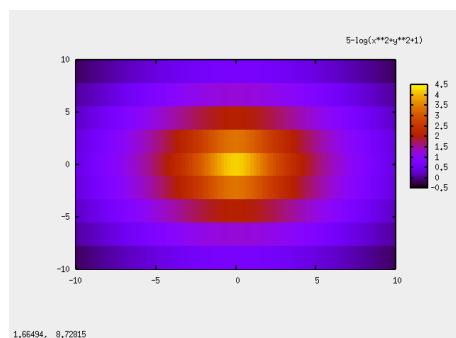


図 10.14: `set pm3d map` の場合

これらの b, s, t の指定を組合せる事も gnuplot では可能です. 指定の方法は単純に, `set pm3d at bs` の様に `at` の後に並べるだけです. では gnuplot で `splot x*y` の描画を, `set pm3d at bs` を行った場合を図 10.15 に示し `set pm3d at sb` を行った場合の結果を図 10.16 に示しておきましょう.

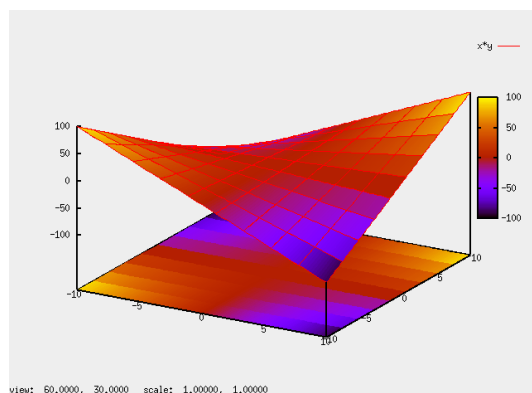


図 10.15: `set pm3d at bs` の場合

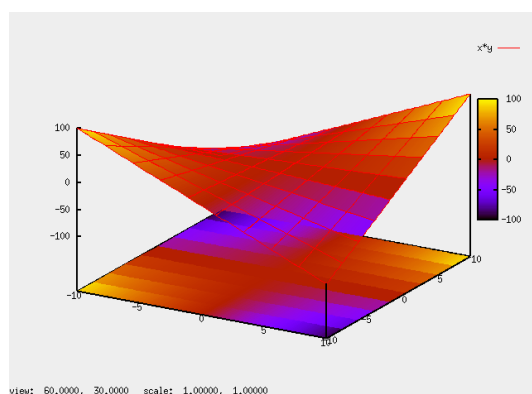


図 10.16: `set pm3d at sb` の場合

図 10.15 と図 10.16 の違いが判ったでしょうか? 先ず, 図 10.16 では, 底部の絵の一部が上にある筈の曲面に被さっているのに対し, 図 10.15 では逆に底部の絵を曲面が覆い被さっていますね. これは, `at` の後が `bs` の場合は最初に底面 (b) を描いてから曲面

(s) を描く為に, 曲面が底面を覆う事になりますが, sb の場合は, 曲面 (s) から描いて次に底面 (b) を描く為に底面が曲面を覆ってしまう事になります.

この様に, at の後の文字列は左側から実行される為, 射影を描いた時に, 上にある側が覆い被されている場合は, この設定の順序を間違えていないか確認すると良いでしょう. 猶, 底面や上面への曲面の投影図の位置を調整する事も可能です. この場合, ticslevel で数値を指定して行います.

投影面の位置の変更

```
set ticslevel < 数値 >
show ticslevel
```

ここでの数値は $\frac{\text{投影面の } Z \text{ 座標} - \text{曲面の最高値}}{\text{曲面の最低値} - \text{曲面の最高値}}$ に等しくなります. 則ち, 0 を指定した場合, 投影面は底辺側で, -1 を指定すると投影面は頂部側に配置される事になります.

これを確認する為に, 以下に $1 - \log(x^2 + y^2)$ のグラフを描いた例を示しておきます.

ここで図 10.17 では `set pm3d at bs` として曲面が投影面を覆う様にし, 図 10.18 では `set pm3d at sb` として投影面が曲面を覆う様に設定しています.

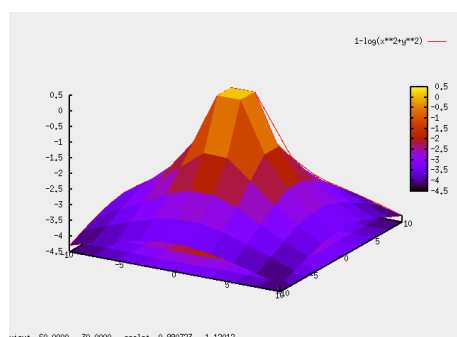


図 10.17: ticslevel が 0 の場合

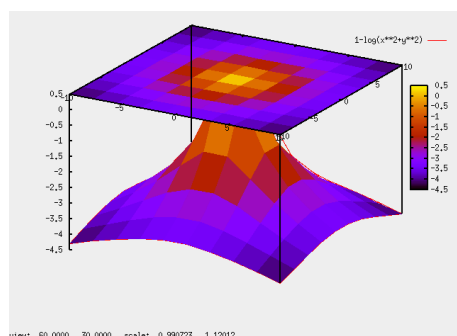


図 10.18: ticslevel が -1 の場合

10.5.6 等高線の階調の変更

pm3d を有効にしていると, 曲面は等高線状に色付けられています. ここで, この階調は変更が可能です. 階調の変更は palette に以下の様な値を設定する事で行います.

palette の変更

```
set palette color
set palette color positive
set palette color negative
set palette gray
set palette gray positive
set palette gray negative
set palette rgbformulae < 数値1>, < 数値2>, < 数値3>
show palette
```

では例として以下の命令文を実行した得られたグラフを使って palette の指定を変更してみましょう.

```
set hidden3d;
set isosample 50;
splot 10-log(sqrt(x**2+y**2+1));
```

この命令文では, hidden3d を設定する事で曲面上の網目を非表示にし, isosample を

50 とする事で X 軸,Y 軸側の分割を各々50 にして曲面の解像度を上げています. 因に,gnuplot では描画した曲線や曲面の設定を変更しただけであれば, `replot` 命令を実行するだけで再描画を行います.

この状態から `set palette color positive` を実行させて再描画したものが図 10.19 に示すグラフになります. この設定が通常の等高線の階調になります.

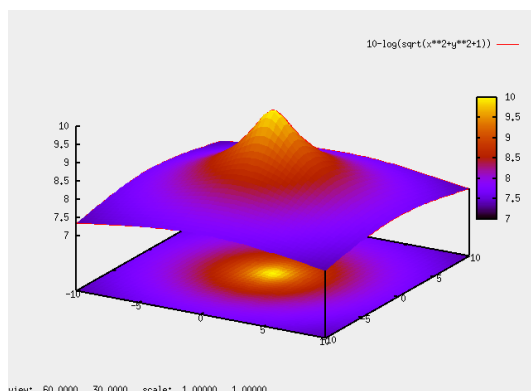


図 10.19: `set palette color positive;replot`

この図に示す様に, 明い色の方が Z 座標が高くなります. もしも, 階調を逆にしたいければ, 今度は `color` の後に `negative` を指定して `palette` の再設定を行い, それから描画を行えば良いのです. 図 10.20 に階調を逆に設定した場合の再描画の様子を示しています.

階調には通常の `color` の他に白黒の `gray` もあります.`gray` を指定した時の様子を以下の図 10.21 に示しておきます.

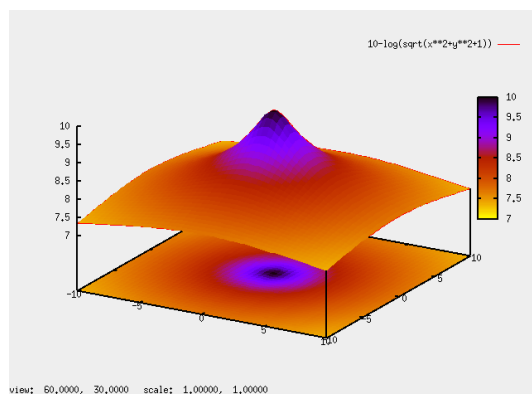


図 10.20: set palette color negative;replot

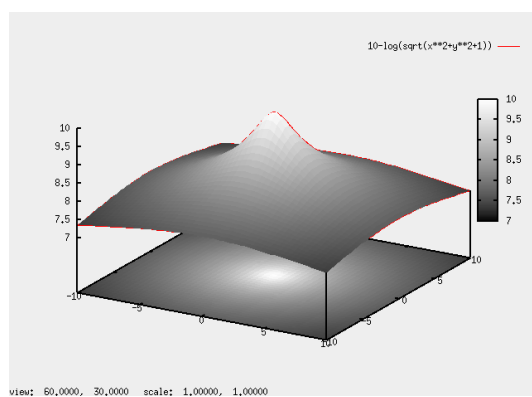


図 10.21: set palette gray positive;replot

gray の場合も, 逆の階調にする場合に gray の後に negative を指定して, palette を設定するだけで済みます,

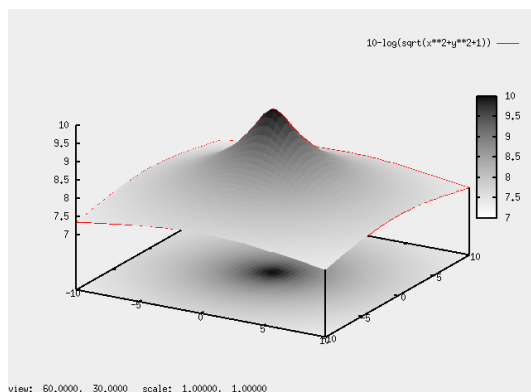


図 10.22: set palette gray negative;replot

この palette は他に色々と指定が可能です. 例えば,rgbformulae を使って次の図 10.23 に示す様な階調も得られます.

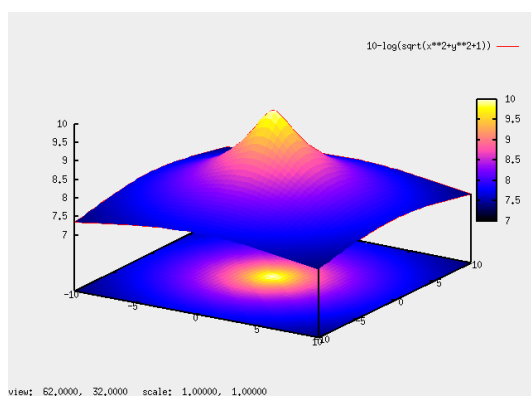


図 10.23: set palette rgbformulae 30,31,32;replot

10.5.7 等高線の調整

さて、等高線の表示自体は pm3d を有効にしなくても表示されます。この場合は本当に線だけになりますが、pm3d とも共存する事が可能です。

先ず、等高線を表示する為には、contour を設定します。

等高線の設定

```
set contour base
set contour surface
set contour both
set contour
show contour unset contour
```

等高線は `set contour` で有効になります。猶、この場合は等高線の射影が描かれるだけで、曲面の側には等高線が現われません。これは contour に base を指定する `set contour base` と同じです。次に contour に surface を指定すれば、今度は曲面の側だけに等高線が表示されます。そして最後の both を指定した場合、曲面の等高線と射影の両方が表示されます。

この事を実際に確認してみましょう。その為に、ここでは以下の設定を始めに実行してから順次、contour の設定を切替えて replot 命令を用いて再描画してみましょう。

```
set isosamples 50
set hidden3d
splot [-4:4] [-4:4] -x*y*cos(x**2+y**2)
```

isosamples を 50 に設定し、splot の XY 座標の範囲は、[-4,4] に限定しています。hidden3d を指定する事で、曲面の陰線処理を行わせる様にしています。

先ず、set contour を指定して、replot を実行した結果を図 10.24 に示します。

この図 10.24 では等高線が投影面に描かれていますね。次に set contour surface set contour を指定して再描画した結果を図 10.25 に示しましょう。

今度は投影面が消えて、曲面に等高線が描かれています。ここで `unset surf` を実行しておけば、曲面のワイヤーフレームも消えて等高線だけが表示されます。

最後に、set contour both set contour を実行して再描画した結果を図 10.26 に示します。

この図に示す様に contour に both を指定する事で、今度は曲面と投影面の両方に等高線が描かれていますね。

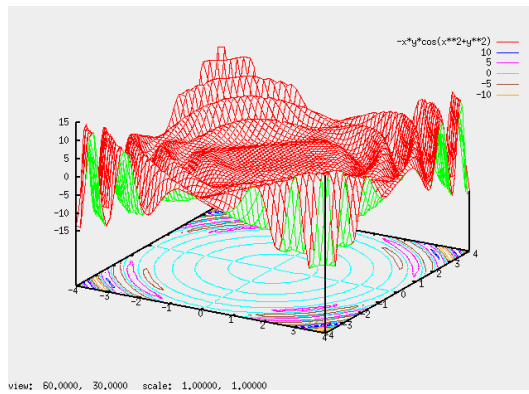


図 10.24: set contour;replot

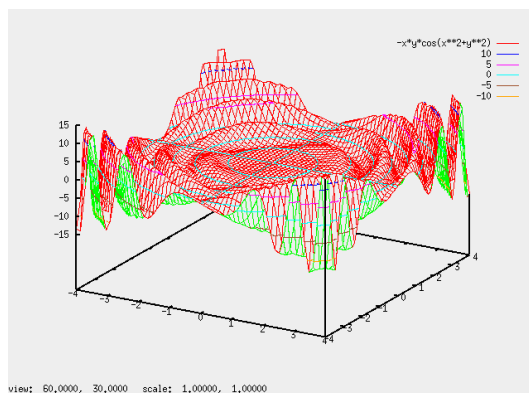


図 10.25: set contour surface;replot

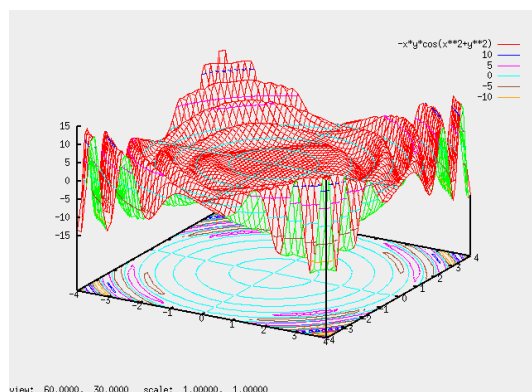


図 10.26: set contour both;replot

contour の設定は pm3d の設定とも共存する事が可能です. 例えば, `set pm3d at s` を指定し, `set contour both` を設定した場合のグラフの様子を見る為に, 以下の入力を試みましょう.

```
set isosamples 100
set hidden3d
set contour
set pm3d
unset surface
splot -x*y*cos(sqrt(x**2+y**2))
```

この結果が図 10.27 になります.

この例では, `unset surf` を実行する事で, 等高線以外の曲面上の曲線を消去しています. gnuplot では等高線の間隔の調整も行えます. この等高線の間隔等の微調整は `cntrparam` を設定する事で行えます. 以下にその構文を示しておきましょう.

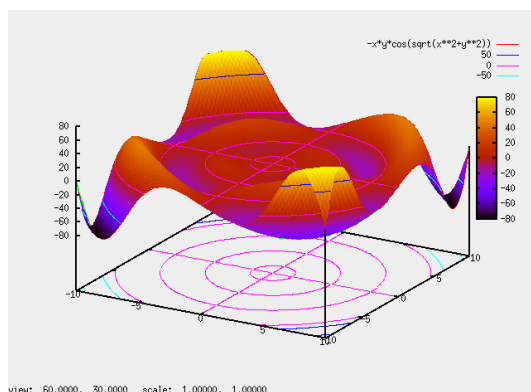


図 10.27: pm3d との共存

等高線の調整

```

set cntrparam linear
set cntrparam cubicspline
set cntrparam bspline
set cntrparam points < 個数 >
set cntrparam order < 次数 >
set cntrparam levels < 等高線総数 >
set cntrparam levels discrete < 高度1 >, ..., < 高度n >
set cntrparam levels incremental < 開始 >, < 増分 >, < 終点 >
set cntrparam levels incremental < 開始 >, < 増分 >

```

先ず, linear, cubicspline, bspline, は等高線の補間式, の指定になります. 初期値としては線形補間 linear が設定されています. cubicspline や bspline を利用する場合, points に指定する数値が補間の点数になり, order で指定する整数値が bspline の補間式の次数を定めます. これらの数値は大きい程, 滑かな曲線が描ける事になります.

次に, 等高線の総数は初期値が 5 となっています. この場合, 最低値から最高値を 5 本の等高線で高度を等分する事になります.

この様な等高線の設定は levels で指定出来ます. この levels では等高線の総数を指定したり, 指定した高さに等高線を描いたり, 指定した間隔で定められた範囲の等高線を描く様に指定する事が可能です.

先ず, 等間隔の等高線を描く場合には levels auto で等高線の総数を指定します. こ

の時, 等高線は最低点と最高点の 2 点を含めて, 指定した数だけ高度に対して等間隔に描かれます.

次に, 指定した高度の等高線だけを描かせる場合は, `levels discrete` で等高線を描く高度を指定します.

例えば, 以下の入力を行った場合, 図 10.28 に示す絵が得られます.

level discrete の例

```
set contour
set isosamples 50
set cntrparam levels discrete -20,0,20
splot x*y*sin(sqrt(x**2+y**2))
```

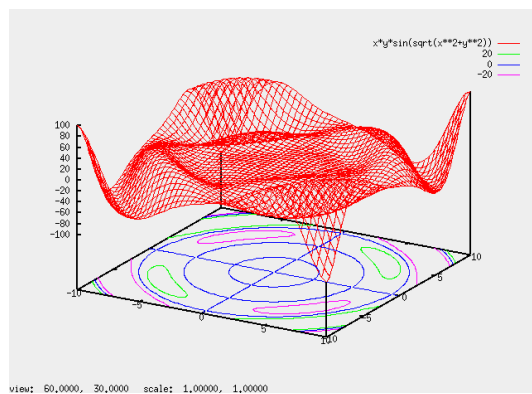


図 10.28: 指定した高さの等高線を描画

ここでの入力, 等高線を高さ-20,0,20 の三種類描く事を

`set cntrparam level discrete -20,0,20` で指定

しています. その為, 図 10.28 の凡例には曲面と 3 個の等高線が出ています.

最後に, 一定の間隔で決められた高度の範囲で等高線を描画する場合は `incremental` を用います. この `incremental` の引数は始点, 増分, そして終点の順で 3 個の引数を記述します. ここで増分は始点と終点が矛盾さえしなければ正でも負でも構いません.

先程の例に以下の命令を実行した結果を図 10.29 に示しておきます。

```
set cntrparam level incremental -20,2,20
replot
```

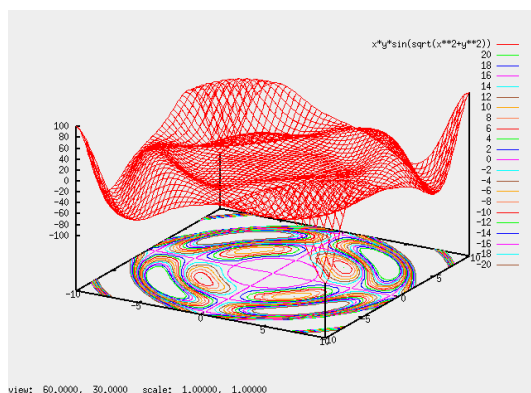


図 10.29: 指定した高さの等高線を描画

10.5.8 視点の変更

gnuplot で 3 次元グラフの視点は `view` で設定出来ます. その構文を以下に示しておきましょう.

view の設定

```
set view <X 軸の回転角>, <Z 軸の回転角> <拡大率>, <Z 軸方向の拡大率>
set view <X 軸の回転角>, <Z 軸の回転角>
set view map
show view
```

ここで, `view` の初期値は X 軸回りの回転角が 60 度, Z 軸回りが 30 度で, 各軸方向の拡大率は 1 になっています.

又, `view` には `map` があります. こちらは効果としては, `set pm3d map` と同じで, 曲面を真上から眺める形になります. これは X 軸回りと Z 軸回りの回転角が 0 度の場合に一致します.

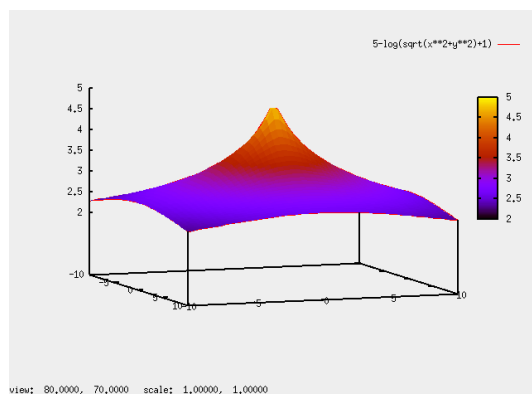


図 10.30: set view 80,70,1,1

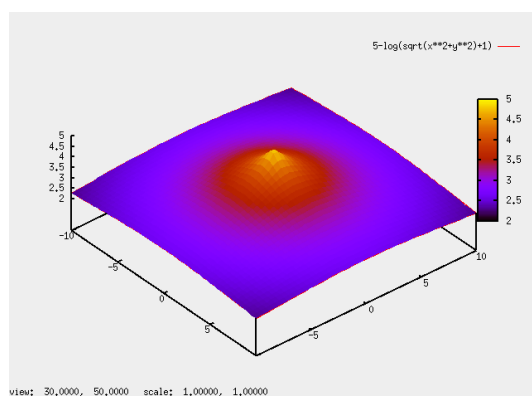


図 10.31: set view 30,50,1,1

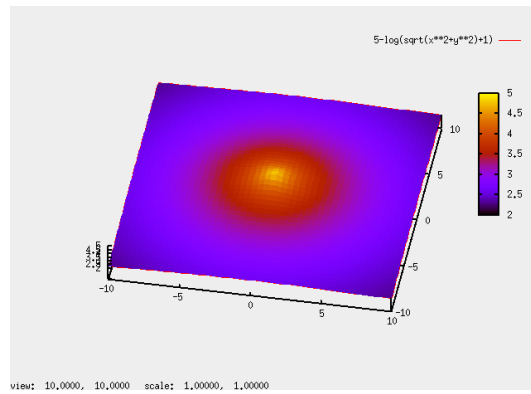


図 10.32: set view 10,10,1,1

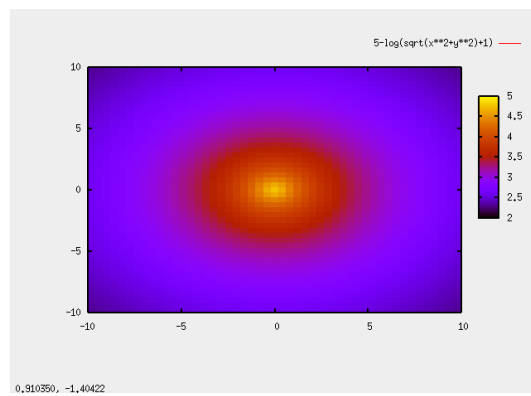


図 10.33: set view map

10.5.9 cbrange と cblabel

pm3d を有効にしていると、右側に曲面の階調を示す箱が表示されています。この階調の調整は cbrange で行いますが、その構文は xrange と同じです。先ず, cbrange は曲面の高さと階調の対応関係を指定するものです。曲面の最低値と最高値が、この cbrange の下端と上端であれば、きめ細かな階調表示になりますが、この cbrange の幅が曲面の高低差と比較して大き過ぎたりすれば、曲面の階調が殆ど出ない、のっぺら坊な曲面になってしまいます。

cbrange の設定

```
set cbrange [〈下限〉:〈上限〉]
unset cbrange
show cbrange
```

この cbrange の設定で、階調の範囲を指定する方法は、MATLAB のベクトルの表記の様な `[a:b]` を用います。これは通常のリスト `[a,b]` とは異なるので注意が必要です。更に、図の右側に通常出現しているこの階調ボックスにラベルを表示させる事も可能です。このラベルの表示も X 軸等の各軸にラベルを配置する xlabel 等と同様の構文になります。

cblabel の設定

```
set cblabel 〈文字列〉〈x 座標〉, 〈y 座標〉
set cblabel 〈文字列〉
unset cblabel
show cblabel
```

ここで、〈x 座標〉, 〈y 座標〉は何も指定しない位置を基準とした文字列表示位置を指定します。ここでの座標は縦上方向を Y 軸正方向、横軸右向きを X 軸の正方向とするもので、1 で一文字分になりますが、この値は表示フォントやターミナル等の環境で異なります。

実例として先程の図 10.23 に対して以下の処理を行ったものを図 10.34 に示しておきます。

```
set cblabel 'Height' -13,6;  
set cbrange [8:10];  
replot
```

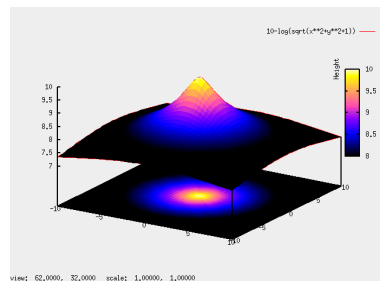


図 10.34: set cbrange [8:10];replot

ここでは cbrange を狭くした為、高さが 8 以下は真っ黒になっています。次に、以下の処理を行ったものを図 10.35 に示します。

```
set cblabel 'Height' -13,6;  
set cbrange [5:10];  
replot
```

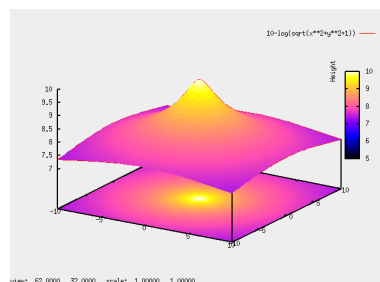


図 10.35: set cbrange [5:10];replot

今度は下に長く取った為に、曲面全体が明るくなっている事が分ります。

10.5.10 陰線処理

gnuplot で三次元グラフを行うと、通常は陰線処理を行っていません。陰線処理を行うためには、hidden3d を有効にします。因に、この hidden3d は pm3d が無効でも、有効になっていても無関係に使えます。

又、pm3d が有効の場合、`unset surface` を実行する事で pm3d の曲面上の網目を非表示にする事が可能です。但し、こちらは pm3d の曲面の設定を無効にする副作用があります。

ここでは set hidden3d と unset surface の違いを確認しましょう。比較で用いるグラフは以下の設定とします。

```
set pm3d at bs
set cbrange [-0.05:1]
splot 1/(x**2+y**2+1)
```

この処理結果を初期状態とし、図 10.36 にそのグラフを示しておきます。

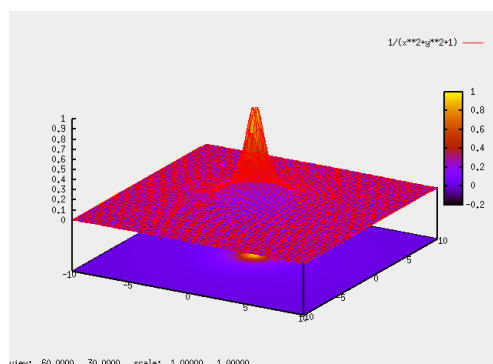


図 10.36: 初期状態

次に、set hidden3d を実行した後に replot した結果を図 10.37 に示します。

最後に、unset surface を実行して replot した結果を図 10.38 に示しておきます。

hidden3d と unset surface の違いは hidden3d では稜線や境界に線分が残っていますが、unset surf では一切の線分が消えている事です。

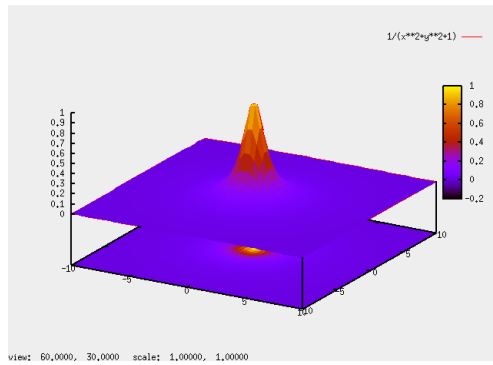


図 10.37: set hidden3d

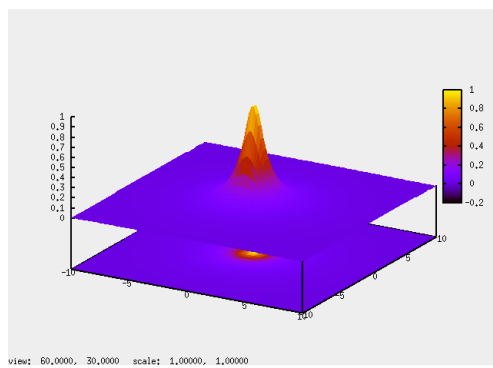


図 10.38: unset surface

猶,gnuplot の式を描画する場合,陰線処理を有効にした場合,解像度の問題から,極が埋没する傾向が強くなります.これは Maxima の様に高精度で計算したデータを用いる場合ではグラフの形が大きく異なるので問題が良く分ります.

実際に確認してみましょう.ここで,描く函数は $\log(x^2y^2)$ とします.

最初に gnuplot で安易に `splot log(x**2*y**2)` を実行した結果を図 10.39 に示します.こう見ると結構綺麗な曲面ですね.

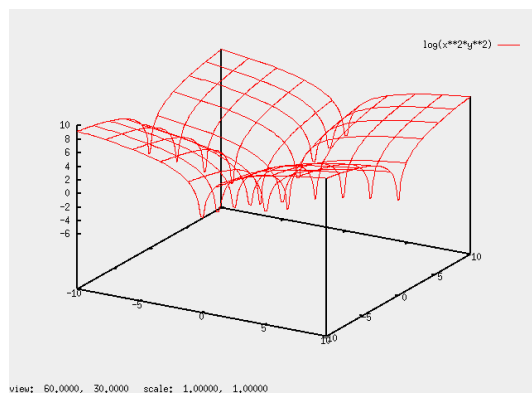


図 10.39: `splot log(x**2*y**2)`(その 1)

次に `set pm3d at s` で `splot log(x**2*y**2)` を実行した結果を図 10.40 に示しておきます。この図からは、面を張ったが為に、粗さが目立っています。

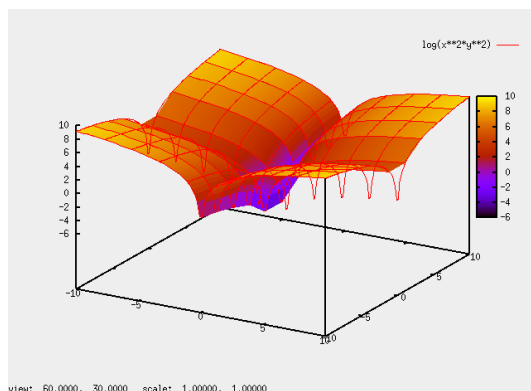


図 10.40: `splot log(x**2*y**2)`(その 2)

そこで、`set isosample 100` に変更して、より細かな分割で曲面を `replot` した結果を図 10.41 に示します。

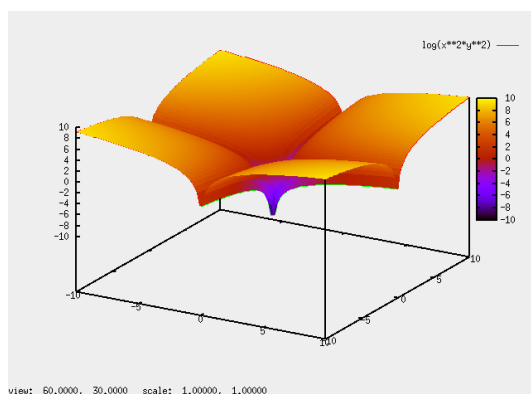


図 10.41: `splot log(x**2*y**2)`(その 3)

こう見ると、原点付近が極みたいです。

では、この函数を Maxima から表示してみましょう。gnuplot と同様の細かさを得る為に、ここでは以下の処理を Maxima に実行させます。

Maxima での描画処理

```
plot3d(realpart(log(x^2*y^2)), [x, -10, 10], [y, -10, 10],
[grid, 100, 100], [gnuplot_preamble, "set pm3d at s; unset surf"]);
```

この処理結果を図 10.42 に示しておきます。

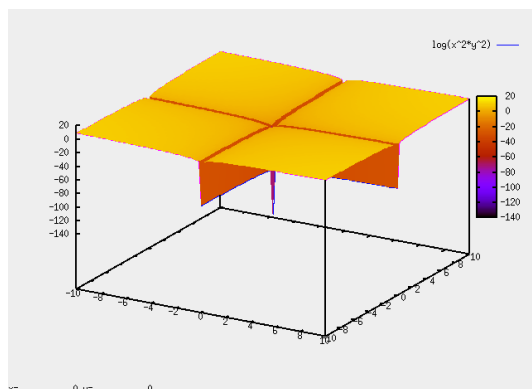


図 10.42: Maxima から $\log(x^2y^2)$ の描画

この図になると、今迄の gnuplot の式のものとは比べて谷と山が遥かに深くなっています。何故、同じ gnuplot でも Maxima から描くと違った形の絵になるのでしょうか？単純にスケールの違いでしょうか？

これは、Maxima では maxout.gnuplot を生成する為に数値データを生成している為に、gnuplot の splot 命令だけで式を記述する方法よりも良好な結果が得られているのです。実は式を良く見れば分る簡単な事です、X 軸上の点と Y 軸上の点が函数の極になっています。即ち、gnuplot の表示は大人し過ぎるのです。この意味では、Maxima の処理の方がまだ雰囲気良く出ていると言えます。

さて、この式を Maxima で描画する際には、Maxima から色々質問が出ています。

———— $\log(x^2y^2)$ の Maxima での描画処理 ————

```
(%i6) plot3d(realpart(log(abs(x))+log(abs(y))),[x,-10,10],[y,-10,10],
[grid,100,100],[gnuplot_preamble,"set pm3d at s;set hidden3d;"]);
Is x zero or nonzero?

nonzero;
Is y zero or nonzero?

nonzero;
(%o6)
```

ここで Maxima が聞いている事は、 x と y が零か零でない数値であるかを尋ねているのです。この様な事を Maxima が聞いて来る時の多くは、 x や y の何れかが零になると非常に都合の悪い事が生じる可能性がある事を示唆しています。この様な質問が出た場合には、函数の定義域に関して吟味する必要があるのです。この点から、Maxima の `plot.lisp` で記述されている内容の方が gnuplot よりも遥かに安全と言えます。

10.5.11 曲線と曲面の細かさの指定

gnuplot の式を表示して粗さが目立つ場合, 曲線の場合は sample , 曲面の場合は isosamples の設定を大きなものに変更します.

sample と isosamples の設定

```
set isosample < 数値1>, < 数値2>
set sample < 数値1>
set isosamples < 数値1>, < 数値2>
set isosamples < 数値1>
show isosamples
```

sample と isosamples には二つの数値か一つの数値を指定します.sample の初期値は 100 で, isosamples の初期値は 10 です.

sample の場合,X 軸上の総点数がこの sample で指定した値になります. 実例として, 以下で描画した $\sin\left(\frac{1}{x}\right)$ のグラフの sample を変更して replot したものを示します.

```
set xrange [-1:1]
plot sin(1/x)
```

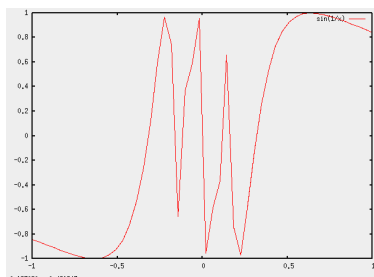


図 10.43: set sample 50 の場合

isosamples の場合, < 数値₁>, < 数値₂> で X 軸と Y 軸の分割数を定め, < 数値₁> だけが指定された場合には, X 軸も Y 軸も, この分割数になります.

但し, Maxima から曲線や曲面を描画する場合, plot2d 関数や plot3d 関数に与える Maxima の式から gnuplot の描画に必要な数値データを計算してしますが, この計算の際に大域変数 plot_options の nticks や grid に設定した値が反映されます. その

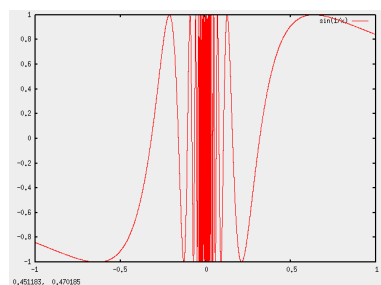


図 10.44: set sample 10000 の場合

為,gnuplot_preamble で sample や isosample を指定しても, この設定は gnuplot の式の描画にだけ有効な為, 直接役には立ちません. とは言え,gnuplot_curves_titles を使って,gnuplot の式を埋込む場合には有効です.

10.5.12 描画の領域設定

plot 命令や splot 命令で描画する gnuplot の式に対し, その描画する範囲は xrange, yrange や zrange で指定します.

描画する領域を指定する場合

```
set xrange [< 下限>:< 上限> < オプション1> < オプション2>
set xrange [< 下限>:< 上限> < オプション1>
set xrange [< 下限>:< 上限>]
set xrange restore
show xrange
set yrange [< 下限>:< 上限> < オプション1> < オプション2>
set yrange [< 下限>:< 上限> < オプション1>
set yrange [< 下限>:< 上限>]
set yrange restore
show yrange
set zrange [< 下限>:< 上限> < オプション1> < オプション2>
set zrange [< 下限>:< 上限> < オプション1>
set zrange [< 下限>:< 上限>]
set zrange restore
show zrange
```

cblabel の小節でも解説しましたが, ここで領域の設定は MATLAB のベクトルの定義に似た書式となります. 即ち, [-1:1] と記述し, [-1,1] でない事に注意が必要です. ここで下限の初期値は -10, 上限の初期値は 10 になっています. その為, 何も指定せずに plot 命令や splot 命令で描画する時は, この領域で描画が実行されます.

この領域の設定には, オプションとして軸の向きを逆にする `reverse`, 元に戻す `noreverse` があります. 早速, `reverse` を `plot` と `splot` で試してみましょう.

```
set xrange [0:10] reverse
plot cos(x)
```



図 10.45: `plot cos(x)`

```
set zrange [0:10] reverse
splot sqrt(x**2+y**2)
```

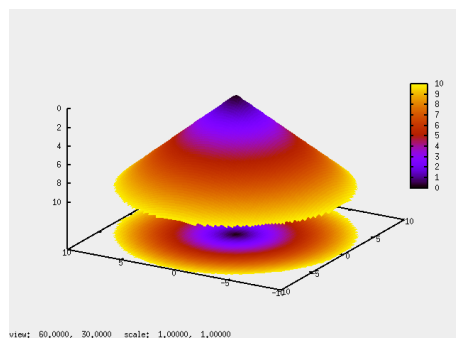


図 10.46: `splot sqrt(x**2+y**2)`

図 10.45 では X 軸の向きが逆になっていますね. 又, 図 10.46 の場合は, Z 軸が通常の逆になっています.

猶, 領域を初期値に戻したければ, `set xrange restore` の様に区間を記述せずに `restore` を指定すれば良いのです.

10.5.13 ラベル表示と注釈に関連する事項

gnuplot の便利の良さの一つに, 曲線や曲面に色々とラベルや注釈を入れる事が容易な事が挙げられるでしょう. 実際, この辺の機能は非常に豊富にあり, この小冊子で把握出来る様なものではありません. その為, ここでは本当に一部の機能の解説に留めておきます. 詳細は文献 [39] を参照して下さい.

では手始めに座標軸のラベルの指定について解説します.

軸のラベル指定

軸のラベルを指定する場合

```
set xlabel < 文字列 > < X 座標 >, < Y 座標 >
set xlabel < 文字列 >
show xlabel
set ylabel < 文字列 > < X 座標 >, < Y 座標 >
set ylabel < 文字列 >
show ylabel
set zlabel < 文字列 > < X 座標 >, < Y 座標 >
set zlabel < 文字列 >
show zlabel
```

先ず, `xlabel`, `ylabel`, `zlabel` は X 軸, Y 軸, そして Z 軸のラベルを指定します. 文字列の後に入れる X, Y 座標は, 通常の文字列が表示される位置を基準として, その位置から何文字分移動するかを指定します. 但し, ここでの値はフォントの指定等で異なる値になります.

曲線の表題表示

複数の曲線を表示した際に、右上側に線分と曲線の表題が並んで表示されています。gnuplot ではこれを key と呼んでおり、この key の位置、key に表示する線分の長さ等の指定が可能です。

最初に key の表示位置を設定する構文を以下に示しておきます。

—— グラフの key の位置を設定 ——

```
set key right
set key left
set key top
set key top left
set key top right
set key bottom
set key bottom left
set key bottom right
set key outside
set key below
```

これらの設定は、key のグラフ上での位置を指定するものです。更に、key の表示自体を調整する事も可能です。

—— グラフの key を設定 ——

```
set key reverse
set key noreverse
set key samplen < 線分の長さ >
set key box linetype < 線分の様式 > linewidth < 線分の幅 >
unset key box
show key
```

ここで key に対する reverse は線分とその名称の羅列の順序を逆にするものです。noreverse は初期状態に戻します。

samplen は key で表示されている線分の長さの調整に用います。

set key box linetype は、key の線分の様式や幅を変更するものです。最後に key を表示したくなければ、`unset key box` とします。

では, 各軸のラベルと key の設定例を以下に示しておきます.

```
set xlabel 'Time -sec-'  
set ylabel 'Height -mm-' 0,40  
set key outside  
set key reverse  
set key samplen 10  
plot sin(x),cos(x),sin(x)/x
```

この例では,X 軸のラベルを'Time -sec-',Y 軸のラベルを'Height -mm-'とし, Y 軸のラベルは 40 文字程上側に配置します. そして,key をグラフの外側に置き, この key の線分の長さを 10 とし, ここでの線分と曲線名の順番を reverse で逆にしています. この様子を図 10.47 に示しておきます.

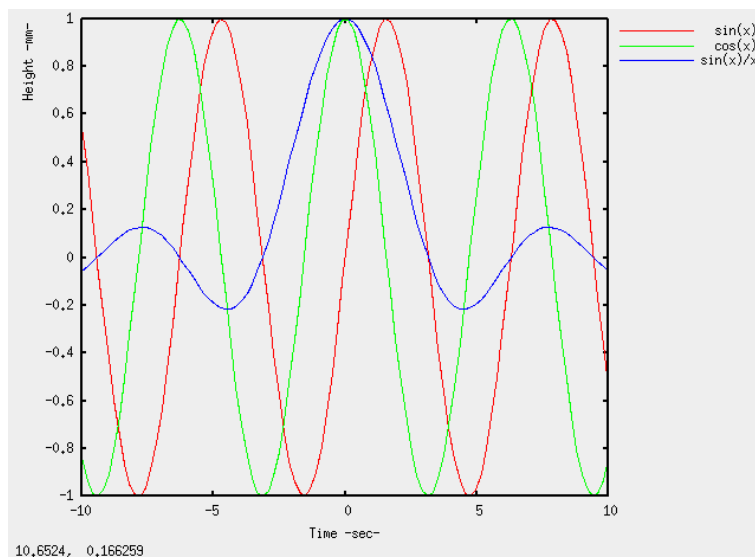


図 10.47: 各軸のラベルと key の設定例

注釈と矢印の追加

gnuplot ではグラフに注釈を入れる事が可能です. この注釈は `label` で設定します.

註釈の設定

```
set label < 文字列 >, < 注釈番号 > at < X 座標 >, < Y 座標 >
set label < 注釈番号 >, < 文字列 > at < X 座標 >, < Y 座標 >
set label < 文字列 >
set label < 注釈番号 > < 位置合せ >
set label < 注釈番号 > font < フォント名 >, < 大きさ >
set label < 注釈番号 > front
set label < 注釈番号 > back
set label < 注釈番号 > textcolor < 表示色 >
set label < 注釈番号 > rotate by < 角度 (deg) >
set label < 注釈番号 > norotate
set label < 注釈番号 > point pointsize < 数値 >
unset label < 注釈番号 >
unset label
show label
```

gnuplot では `set label` 文を < 注釈番号 > を指定せずに実行すると, 自動的に注釈に適切な番号を割当てます. その後の註釈の回転, 内容やフォントの変更等の処理はこの注釈番号を指定して行わなければなりません. 指定しない場合には新規の註釈が生成されるだけです.

ここで注釈の座標は先程のグラフの点の位置に対応します. 猶, 文字列の左寄, 中寄, 右寄といった位置決めは各々 `left`, `center`, `right` を指定する事で行います.

注釈を XY 面内で回転させる事も可能です. この場合, `rotate by` で行いますが, `by` の後に指定する角度の単位は度になります.

では実例として, 以下に示す処理を gnuplot でさせてみましょう.

```

set label 1 ' Comment at Origin' at 0,0
set label 2 ' Comment at (1,1)' at 1,1
set label 1 point pointsize 6
set label 2 point pointsize 2
set label 2 rotate by 90
plot sin(x)*2

```

最初に注釈を原点と (1,1) に配置しています. それから註釈番号 1 の註釈に対して大きさが 6 の点を配置しています. 同様に, 番号 2 の註釈には大きさが 2 の点を配置します. それから番号 2 の註釈を 90 度回転させています. 最後は, plot 命令で $2\sin x$ 関数の描画を行います.

この結果を図 10.48 に示しておきます.

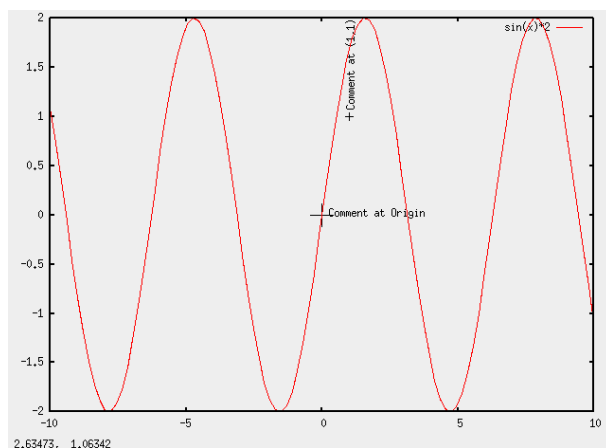


図 10.48: 注釈の設定例 (その 1)

或る注釈を非表示にしたければ, `unset label < 注釈番号 >` と入力する事で対処します. ここでは図 10.48 の例に対し, 番号 2 の註釈を `unset label 2` を実行した後に replot した結果を図 10.49 に示しておきます.

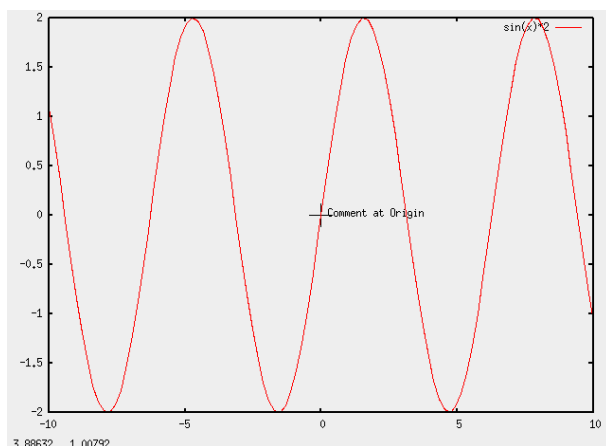


図 10.49: 注釈の設定例 (その 2)

番号 2 の註釈が実際に消えている事が分ります.

矢印の追加

gnuplot では矢印をグラフ中に追加する事も可能です. この矢印は註釈と併用する事で効果が上ります.

arrow による矢印の設定

```
set arrow < 番号 > from < X 座標 >, < Y 座標 > to < X 座標 >, < Y 座標 >
set arrow < 番号 > from < X 座標 >, < Y 座標 >
set arrow < 番号 > to < X 座標 >, < Y 座標 >
set arrow < 番号 > size < 矢の長さ >, < 矢の角度 >
set arrow < 番号 > head
set arrow < 番号 > heads
set arrow < 番号 > nohead
set arrow < 番号 > filled
set arrow < 番号 > nofilled
unset arrow
show arrow
```

やっている事は label を用いた註釈の設定に似ています. 先ず, 図形としての矢印の設定は

`set arrow 1 from 0,0 to 1,1` の様に始点と終点を決めるだけで生成されます. この時, 矢印の向きは from から to に向きます. 猶, from の点, 或いは to の点が原点の場合は省略しても構いません.

次に arrowsize で矢印の頭の部分を調整する事が可能です. ここでの〈矢の長さ〉は矢印の頭の矢の長さで, 〈矢の角度〉は頭の矢の角度になります. この角度は度数です. filled を指定すると, 矢印の矢の部分が塗り潰されます.

では実際に以下の様に入力してどの様なグラフになるか確認してみましょう.

```
set arrow 1 from 0.4,-1 to 1.1,-1
set arrow 2 from 0.9,1 to 1.1,1
set arrow 3 from 1,-1 to 1,1
set arrow 1 nohead
set arrow 2 nohead
set arrow 3 size 0.1,30 filled heads
set label 1 ' Width' at 1.05,0
plot cos(6.283*x);
set xrange [0:2]
set yrange [-2:2]
plot cos(6.283*x);
set arrow 3 filled
plot cos(6.283*x);
```

ここでは最初の三行で矢印を生成しています. 次の三行で各矢印の性質を指定しています. ここで, 第三の矢印のみ両端に矢を付け, その他は棒にします. そして, 第三の矢印の矢の部分を大きくして塗り潰してしまいます. この矢印の設定は, この様に一度に設定しなければ, 最後に入力した設定だけが有効になり, 他は初期化されてしまうので注意が必要です.

この結果を図 10.50 に示しておきます.

10.5.14 gnuplot の式

gnuplot では FORTRAN で用いられる数式が利用可能です. その他の関数も充実していますが, ここでは簡単に gnuplot の式について述べます.

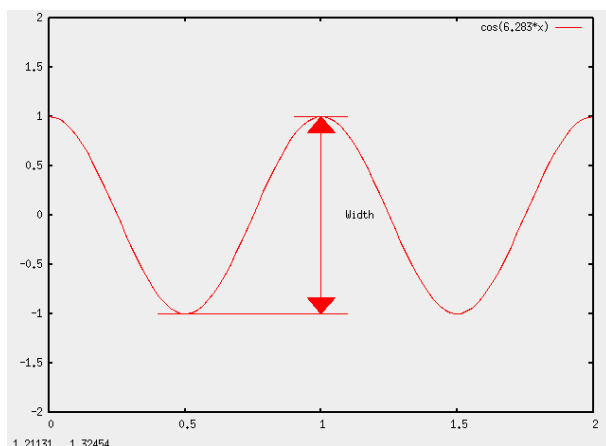


図 10.50: 矢印と註釈の設定例

先ず, 式の演算子は FORTRAN の書式に準じます. gnuplot の算術演算子を以下に示しておきます.

— gnuplot の算術項演算子 —

演算子	例	概要
+	a+b	和
-	a-b	差
*	a*b	積
**	a**b	冪
/	a/b	商
%	a%b	剰余
-	-a	負の演算子
+	+a	正の演算子
!	a!	階乗

この様に,gnuplot では一般的な算術演算子の殆どが使えます. 但し, 冪は Maxima 等の数式処理でよく用いられる[^]ではなく, FORTRAN 等で用いられる**2 を用います. この点は Maxima と併用する場合には特に間違え易いので注意しましょう.

更に,gnuplot では様々な数学関数が使えます. 以下に gnuplot で利用可能な函数の一覧を示しておきます.

数学函数

abs	絶対値
sqrt	平方根を計算
imag	複素数値の虚部を返す函数
real	複素数値の実部を返す函数
rand	乱数函数

三角函数

cos	余弦函数
sin	正弦函数
tan	正接函数
acos	逆余弦函数
asin	逆正弦函数
atan	逆正接函数
atan2	逆正接函数
cosh	双曲余弦函数
sinh	双曲正弦函数
tanh	双曲正接函数
acosh	逆双曲余接函数
asinh	逆双曲正弦函数
atanh	逆双曲正接函数
exp	指数函数
log	自然底の対数函数
log10	底が 10 の対数函数
arg	偏角を返す函数

— 特殊函数 —

ibeta	不完全 B 函数
gamma	Γ 函数
igamma	不完全 Γ 函数
besj0	第一種のベッセル函数 $J_0(x)$
besj1	第一種のベッセル函数 $J_1(x)$
besy0	第二種のベッセル函数 $Y_0(x)$
besy1	第二種のベッセル函数 $Y_1(x)$
erf	誤差函数
erfc	相補誤差函数 ($erfc(x) = 1 - erf(x)$)
inverf	逆誤差函数
norm	正規分布
invnorm	逆正規分布函数
lambertw	ランベルトの W 函数

— その他の函数 —

ceil	与えられた値を越えない整数に 1 を加えた値を返す. $ceil(x)$ は $floor(x) + 1$ と同じ
floor	与えられた値を越えない整数を返す
int	小数点の切り捨てを行う函数
sgn	符号函数. 引数が負の場合は-1, 正の場合は 1 を返す

これらの算術演算子と函数の他に論理演算も可能です.

gnuplot の論理演算子

演算子	例	概要
==	a==b	等号
!=	a != b	等しくない
>=	a>=b	以上
>	a>b	大なり
<=	a<=b	以下
<	a<b	小なり
&&	a&&b	論理積
		論理和
&	a&b	ビット単位の論理積
		ビット単位の論理和
~	~ a	補間
!	!a	否定

ここで独特なのが、ビット単位の論理積&と論理和|です。これらは与えられた整数に対し、それらを二進数で置換えて各桁で論理積や論理和を各々実行した結果を返す演算子です。

これらの演算子を活用する事で複雑な関数を gnuplot 内部で定義し、更に、その関数を使って描画する事も可能です。この場合、関数に与えられる式は $\sin(x)**1$ の様な gnuplot の関数と算術演算子を用いた式、そして、変数の範囲で関数を切り替える式の二種類が使えます。

関数の定義

〈 関数名 〉 (x) = 〈 gnuplot の算術演算子と関数の式 〉
 〈 関数名 〉 (x) = 〈 条件₁ 〉 ? 〈 式₁ 〉 : ... 〈 条件_n 〉 ? 〈 式_n 〉
 〈 関数名 〉 (x) = 〈 条件₁ 〉 ? 〈 式₁ 〉 : ... 〈 条件_n 〉 ? 〈 式_n 〉 : 〈 各条件を満たさない場合の値 〉

条件は gnuplot の論理式で構成されるもので、関数の定義域を定めるものです。例えば、-1 から 1 迄が x^2 で、それ以外を 1 とする関数を定義したければ、以下の様に設定します。

```
f(x) = x >= -1 && x <= 1 ? x**2 : 1
```

この様に、指定した区間以外の値は関数の定義の最後に値を設定すれば良いのです。更

に,gnuplot 上で定義する函数の変数は x,y,z に限定されません.

猶, 特定の区間のみの値のグラフ表示をしたければ, 特定の区間だけで式を定義しておいて, 最後に $1/0$ や $-1/0$ の様な極を設定すれば, グラフ表示では指定した区間だけが表示されます.

```
f(x)= x>=-5 && x<=1 ? cos(x): x>=1 && x<=5 ? sin(x):\
> x>=7 ?sqrt(x) : x<=-7?(x-x**2)/100: 1/0
plot f(x)
```

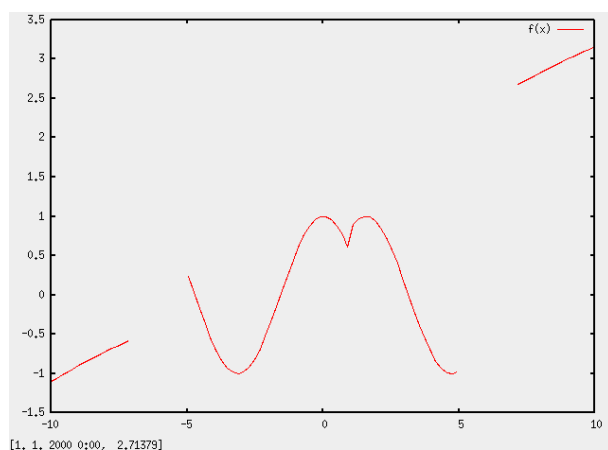


図 10.51: 不連続なグラフの描画

10.5.15 電卓としての gnuplot

この様に豊富な函数もあり, 自分で函数も定義出来るのであれば, Octave 迄はいかなくても, 簡単な卓上計算機として使いたいものです. ところが, Maxima と Octave とは異なる事は, 函数に値を入れたものを入力してもエラーが返される事でしょうか.

```
gnuplot> 1+1
~
invalid command
```

```
gnuplot> sin(0)
```

```
~  
invalid command
```

この事は gnuplot の式はグラフ描画の為に用いる事が第一であって、電卓代りにする事はあまり考えていなかった為でしょう。

とは言え、前述の様に C や FORTRAN の様な式が入力可能な程の能力を持っているので、このままにしておくのは非常に勿体無いので、電卓として使う方法について簡単に述べておきましょう。

とは言っても話は非常に簡単な事で、式の結果を何等かの変数に代入して、その変数を print 命令 に引渡すか、その式を直接 print 命令に引渡せば良いのです。

```
gnuplot> print 1+1  
2  
gnuplot> print cos(3.14)  
-0.99999873172754
```

で、電卓として使う場合に、この電卓で扱えるデータには何があるでしょうか？先程の式の解説では gnuplot で扱えるデータに関して頼被りしていましたが、gnuplot には論理演算を行う為の真偽値、整数値、浮動小数点、そして複素数値があります。

真偽値

gnuplot では真を 1, 偽を 0 とします。この値は他の数値と混在しても構いません。この事を利用すれば if 文を用いずに式を容易に定義する事が出来ます。例えば、x が 2 なら 10 を返し、それ以外は -9 を返す関数も以下の様に定義します。

```
gnuplot> g(x)= (x==2)*10 + (x!=2)*(-9)  
gnuplot> print g(2)  
10  
gnuplot> print g(4)  
-9
```

この方法は Octave でもお馴染の手応です。この機能を使えば、gnuplot でも長々と条件分岐を書かなくても済む訳です。

整数値, 浮動小数点 整数は小数点を持たない実数で、浮動小数点は小数点を持つ実数です。ここで整数同士の演算結果は整数になるので、整数同士の割算を行う場合は注意が必要です。


```
gnuplot> print 4/3, 4.0/3
1 1.333333333333333
```

この例では、最初の $4/3$ は分子、分母共に整数の為、整数の剰余が適用されています。但し、最後の $4.0/3$ では分子が浮動小数点の為、割算の結果は浮動小数点になります。この様に、実数式で浮動小数点を含む式の場合、結果は浮動小数点に、後述の複素数を含む式の場合、その結果は複素数になります。

複素数 複素数は大括弧 $\{ \}$ を用います。例えば、 $1 + 2i$ は $\{1, 2\}$ の様に数値の対で表現します。複素数の場合、実部と虚部は各々が浮動小数点になります。

```
gnuplot> print 1+{1,3}+{0,3}
{2.0, 6.0}
gnuplot> print 3*{1,4}/9
{0.3333333333333333, 1.333333333333333}
```

最初の例で示す様に、入力側の数値は整数でしたが、結果は実部と虚部が浮動小数点になっている事に注意して下さい。又、複素数に対しても通常の算術演算子が使えます。猶、実部を返す命令が `real` で、虚部を返す命令は `imag` です。

この様に、gnuplot は電卓としても十分な機能を持っています。この電卓には履歴機能に加えて、グラフ描画機能さえも付いて来るのです。素晴らしい事です。

猶、函数の定義で特定区間の描画を行う為に、函数の値に、 $1/0$ を設定すると申しましたが、`print` 文でこの式を表示させた場合には、`undefined value` となります。

10.5.16 プログラム言語としての gnuplot

gnuplot では変数が扱え、函数も定義出来ます。その際に Octave の様な処理も行える訳ですが、ではプログラム言語としてはどうでしょうか？

gnuplot では一応、`if` 文による条件分岐を持っていますが、`surf` で見られる様な `if-goto` による原始的な `loop` 文さえも無く、反復処理を行う為には、`reread` 命令 を用いてファイルを介して行う必要があります。

先ず、`if` 文の構文を以下に示しておきますが、基本的に C に似た構文になっており、文の区切はセミコロンで行います。

if 文の構文

```

if ((論理式)) <文1>; ... <文n>
if ((論理式)) <文1>; ... <文n>; else <文n+1>; ... ; <文n+m>;
if ((論理式1)) <文1>; ... ; <文n>; else if ((論理式2)) <文n+1>; ...

```

gnuplot の if 文は論理式や else の後に文をセミicolonで区切って並べます。但し、複雑な処理は行えません。if 文では条件式に合致する場合に、条件式の直後から else が出現する迄の文全てを処理し、そうで無い場合には、else の直後の文から行末の文迄の全ての文を実行する仕様の為です。

その為、gnuplot の if 文は文中に if 文を入れて階層的にせず、平面的に処理を行うのが妥当です。

又、処理言語の多くが条件分岐に加えて反復処理を行う制御文を持っています。ところが、gnuplot の場合は明示的な反復処理を行う制御文を持っていません。gnuplot では再帰的に reread 命令を用いて gnuplot の文を記述したファイルの再実行による反復処理だけしか行えません。

例えば、次の様に用います。

ファイル tama の内容

```

set isosamples 50;
if(amp<5) splot [-3:3] [-3:3] [-1:1] \
amp*sin(x*y)*exp(-sqrt(x**2+y**2));\
print "Hit any key!";pause -1;\
amp=amp+1;reread;else print "The End";

```

— 実行の様子 —

```
gnuplot> set pm3d at s; set hidden3d;\
> amp=1;load "tama"
Hit any key!

Hit any key!

Hit any key!

Hit any key!

The End
gnuplot>
```

この様に、ファイルを介した再帰によって反復処理が実現出来ます。猶、ファイルを介する為、反復処理の制御を行う変数の初期化は gnuplot 側で実行しておく必要があります。その為、現時点の Maxima でアニメーションを実行させる方法は、結局、maxout.gnuplot に書き込んで、それを gnuplot に引渡す方式の為、制御変数の初期化を maxout.gnuplot に書込む方法しか無いので、plot2d や plot3d を用いてスマートに処理させる事は難しくなります。

強引に実行させるのであれば、gnuplot の load 命令と save 命令を用いて、制御変数を別途臨時ファイルに保存と再読込を行う事で実現させる事が可能です。

— ファイル tama —

```
load "nekonekoxx";
set isosamples 50;
if(amp<5) splot [-3:3] [-3:3] [-1:1] \
amp*sin(x*y)*exp(-sqrt(x**2+y**2));\
print "Hit any key!";pause -1;\
amp=amp+1;save var "nekonekoxx";reread;else print "The End";
```

— nekonekoxx の内容 —

```
amp = 0
```

この例では予めファイル nekonekoxx に amp の値を設定しておきます。それから、gnuplot でファイル tama を load 命令を使って読み込みます。このファイル tama では、変数 amp

の値をファイル nekonekoxx から読込む事で gnuplot 内部に取り込み, それから amp の値が 5 より小であれば, グラフの表示と amp に 1 を加えてファイル nekonekoxx に gnuplot 内部の変数を保存して reread 命令を用いて自分自身を呼出します. amp が 5 以上であれば, The End と表示します.

この手法を Maxima で使えなくもありませんが, plot2d や plot3d では Maxima 側で数値データとして計算した結果を gnuplot に引渡すので, アニメーションにしたい Maxima の式を gnuplot_curve_titles 側に引渡すといった随分と込み入った方法になります. その為, 割り切って処理を行う必要があります.

MS-Windows 環境とそれ以外の環境では, Maxima は gnuplot の立ち上げ時のオプションが異なります. この立ち上げ時のオプションは大域変数 gnuplot_view_args で指定されています. MS-Windows 環境ではオプションが付きませんが, その他の環境では, -persist が付きます. このオプションは gnuplot を閉じてもグラフウィンドウを残すという gnuplot のオプションです.

以上で gnuplot の簡単な使い方の解説を終わります. 以降は, Maxima での使い方について幾つかの実例の解説をしましょう.

10.6 plot2d 関数と plot3d 関数の活用事例

10.6.1 gnuplot_preamble の使い方

この節では plot2d 関数と plot3d 関数で gnuplot を用いた描画例を幾つか示しておきます。

猶,gnuplot は非常に機能が高いアプリケーションで色々な事が出来ますが, 本質的には plot 命令と splot 命令で描画,set 命令で諸設定, そして replot で再描画を行う仕組みになっています。

その一方で,Maxima では plot2d 関数や plot3d 関数に与えられた式から数値データを生成し, 諸設定は,gnuplot の命令を単純にファイルやストリームに出力する方式を採用しています。

その為,gnuplot を Maxima 側から操作する事は,gnuplot にストリームとして送り込んだり,maxout.gnuplot に書き込んだりする gnuplot の命令文を, 如何に記述するかという問題に帰着されます。

ここで,gnuplot は非常に高機能のツールである為,Maxima に準備された plot_options の項目だけで全てを網羅する事は事実上不可能です。しかし, 大域変数 plot_options の項目の gnuplot_preamble に様々な gnuplot の制御文が設定可能となっているので, この特性を利用しないのは幾ら何でも勿体無い事です。但し,plot_options で設定可能な項目は各々位置が決っている為, 思い通りに利用する為にはそれなりの工夫が必要になります。その為,plot_format を gnuplot にして,maxout.gnuplot の内容を確認しながら進めて行く事は非常に有効な手段です。

さて, 今迄の復習になりますが, グラフの表題を mike,X 軸と Y 軸のラベルを各々 X,Y としたい場合,Maxima でどの様に入力すれば良いかを考えましょう。

ここで,gnuplot ではグラフの表題は set title で指定し, X 軸と Y 軸のラベルの設定は, 各々 set xlabel と set ylabel で行います。最終的には, これらの gnuplot の命令文をセミコロンで区切り, 全体を二重引用符で括ったものを gnuplot_preamble に設定します。

この処理と描画の実行を以下に示しておきます。

```
(%i5) nekoneko:"set title 'mike';set xlabel 'X';set ylabel 'Y';";
(%o5)          set title 'mike';set xlabel 'X';set ylabel 'Y';
(%i6) plot2d(sin(x)/x,[x,0,10],[gnuplot_preamble,nekoneko]);
```

この処理の結果得られるグラフを図 10.52 に示しておきます。

念の為, この例題の maxout.gnuplot の先頭部分を確認しておきましょう。

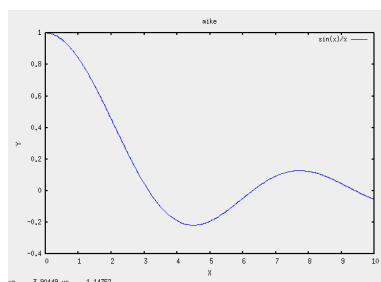


図 10.52: gnuplot_preamble で設定した結果

maxout.gnuplot のヘッダ部分

```
set title 'mike';set xlabel 'X';set ylabel 'Y';
plot '-' title 'sin(x)/x' with lines 3

2.441406250000E-4 0.9999999900658926
4.88281250000E-4 0.9999999602635706
---省略---
```

この様に,gnuplot_preamble の内容は gnuplot の描画命令である plot 命令や splot 命令の直前にそのまま埋込まれます. 又,Maxima の式は解釈されて,数値データとして gnuplot の描画命令の直下には書込まれます. 従って,plot 命令や splot 命令のオプションとして設定される曲線の表題や曲線の様式を gnuplot_preamble で設定したとしても,描画の際に,plot 命令や splot 命令のオプションが優先され,その上,式で gnuplot は描画を行わないので,sample や isosamples の設定は意味はありません. 曲線の表題や曲線の様式は各々,gnuplot_curve_titles と gnuplot_curve_style に設定すべきであり,描画する曲線や曲面の細かさは予め nticks や grid で設定すべきである事が理解されるでしょう.

では,以下の様な gnuplot_curve_style への設定を行って描画させてみましょう.

```
plot2d(sin(x),[x,-1,1],[gnuplot_curve_styles,"with impulse"])
```

この時,maxout.gnuplot の plot 命令付近は以下の様になっています.

gnuplot_curve_styles の設定

```
plot '-' title 'sin(x)' with impulse
-1.      -0.8414709848078965
-0.975   -0.8277018881672576
--- 略 ---
```

この結果は図 10.53 のようになります。

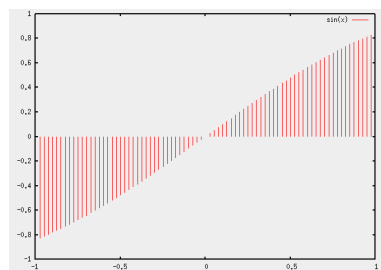


図 10.53: gnuplot_preamble で設定した結果

10.6.2 Maxima のバッチ処理

今度は,Maxima で複数のグラフを描く処理を行う為に,Maxima にバッチ処理を行わせる方法について簡単に解説しましょう。

先ず,gnuplot_preamble に記述する設定の内容は,高度なグラフ表示を行わなければならないならば,益々,膨大なものとなるでしょう。これを一々手入力する事は最初一枚のグラフを得る為に試行錯誤する場合は仕方が無いものとしても,書式が全て決っており,その決められた書式で出力すれば済む話であるのならば,自動化を図りたいところです。

ここで Maxima にはバッチ処理機能があります。この場合,Maxima に実行させるファイルを予め準備しておきますが,このファイルの中身は通常の Maxima の式をそのまま記述したものです。このファイルを Maxima に引渡すと,Maxima がファイルの内容に従って処理を行うもので,この処理は Maxima の通常の数式や数値の処理だけではなく,画像の生成にも使えます。

手始めに簡単な入力ファイル A を記述しておきましょう。

—— ファイル A の内容 ——

```
a: (x+1)^2;
b: expand(a);
plot2d(b, [x, -1, 1]);
plot2d(b, [x, -1, 1],
[gnuplot_preamble, "set term png;set output 'test.png'"]);
```

これは非常に簡単な処理です。最初の二つの式は通常の Maxima の処理の式です。それから plot2d の処理を行なっていますが、この場合は,gnuplot でグラフを生成し、グラフ表示ウィンドウにグラフを表示してものを出力します。そして、最後の処理では,gnuplot_preamble を利用して描画を行うものです。ここで gnuplot_preamble には、`set term png` と `set output 'test.png'` といった二つの gnuplot の設定文を纏めたものを入れています。ここで、最初の命令文は gnuplot の terminal を PNG に変更するもので、次の命令文が出力先をファイル 'test.png' に切替えています。即ち、最後の二つの命令文により,gnuplot はグラフを PNG データとして、ファイル test.png に書込む事になります。

ここで,gnuplot が出力可能な画像データ形式は MS-Windows か Linux 等の OS 環境によって異なります。一応,PNG であれば問題は無いと思いますが、確認したければ、gnuplot を立ち上げて、`? term` と gnuplot に入力してみましょう。すると色々と説明が表示されますが、末尾に一覧が表示される筈です。その中にある端末が利用可能なので、その一覧表にある画像データ形式を選択しましょう。

このファイルは Maxima の中から、`batch(A);` と入力する事で処理が出来ますが、一々,Maxima を立ち上げなくても、以下の構文で Maxima にファイルを引渡す事が可能です。

—— Maxima によりバッチ処理 ——

```
maxima -b <ファイル>
maxima -batch=<ファイル>
```

猶、この処理は MS-Windows でもコマンドプロンプト¹からの実行が可能です。因に、デスクトップに現われている Maxima のアイコンは wxMaxima のアイコンです。この wxMaxima は Maxima に被せる GUI 環境に過ぎません。その為、上に示した方法は wxMaxima には使えません。この場合は,batch 函数を使いますが、画像の生成を行いたければ,wxMaxima ではなく,maxima を起動させた方が良いでしょう。猶,maxima をコ

¹所謂 DOS 窓

マンドプロンプトから起動させる為には、MS-Window の環境変数の PATH に maxima への経路を追加する必要があります。この設定は、コントロールパネルのシステムから詳細の中にある環境変数を選び、システムの環境変数の Path に書き込みを行います。この場合、maxima にパスが通っている必要があります。そうでなければ、フルパスで指定する必要があります。更に、バッチファイルで term の設定を行っている場合、Maxima に付属の gnuplot では使えない term が存在するので、注意が必要です。例えば、jpeg は使えませんが、gif と png は利用可能なので、これらを選択すると良いでしょう。

Maxima によるバッチ処理の計算結果は、出力先をバッチファイル内部で指定しない限り、標準出力に出力されます。重要な計算を行う場合は、`maxima -b test>test.log` の様にリダイレクトを用いると良いでしょう。

又、一々、コマンドラインに入力するのが面倒であれば、この内容を含むシェルスクリプトを記述しておくのも良いでしょう。

参考迄に `maxima -b A` を実行した様子を以下に示しておきます。

```
yokota@Zuse:~/TEST> maxima -b A
Maxima 5.10.0 http://maxima.sourceforge.net
Using Lisp CLISP 2.37 (2006-01-02)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
This is a development version of Maxima. The function bug_report()
provides bug reporting information.
(%i1) batch(A)

batching /home/yokota/TEST/A
                                2
(%i2) a : (1 + x)
                                2
                                (x + 1)
(%o2)
(%i3) b : expand(a)
                                2
(%o3) x + 2 x + 1
(%i4) plot2d(b, [x, - 1, 1])
(%o4)
(%i5) plot2d(b, [x, - 1, 1], [gnuplot_preamble,
                                set term png;set output 'test.png'])
(%o5)
```

この処理では、gnuplot のグラフも表示されています。但し、このグラフは面白味が無いので、ここでは示しません。あしからず。

次に、もう少し複雑なグラフを描いてみましょう。その為に、次の内容のファイル A2 を記述しましょう。

—— ファイル A2 の内容 ——

```
nekoneko:"set pm3d at bs;set xrange [-2:2];\
set yrange [-2:2];set zrange [-10:20];\
set label 1 'top' at 0,0,20;\
set arrow 1 from 0,0,20 to 0,0,10;\
set arrow 1 size 5,30 filled head;\
set hidden3d;\
set output 'test1.png';set term png;"
plot3d(10*cos(x*y),[x,-2,2],[y,-2,2],[grid,50,50],
[gnuplot_preamble,nekoneko]);
```

このファイル A2 では,gnuplot_preamble に与える文字列が長くなるので, 適宜, 行が継続している事を示す\を入れて改行しています.

`maxima -b A2` の結果の text1.png の絵を図 10.54 に示しておきます.

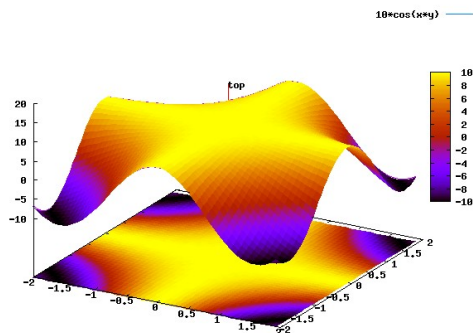


図 10.54: ファイル A2 の実行結果

ここで gnuplot で行う処理が複雑になればなる程,gnuplot_preamble の内容は必然的に長くなります. 画像 1 では設定 1, 画像 2 の場合に設定 2 を使う…等々とする場合, その設定を全て gnuplot_preamble に与える文字列として持たせるという方法は, 安易な方法ではありますが, 自動処理の事を考えた場合には, それ程, 効率的ではありません. 寧ろ, 共通の設定 1 があって, 後はケース毎に設定が追加される方が汎用性が高いでしょう. 又, ケース毎に画像ファイル等の保存先のファイルを切り替える事は普通に行われる事でもあります. ところで,gnuplot_preamble に与えるのは一つの文字列です.

そこで,Maxima で複数の文字列を纏めて一つの文字列にする操作を行えば良い事になります.Maxima には, 文字列の結合を行う処理を行う関数として, `concat` 関数があります. この関数は与えられた複数の文字列を一つの文字列に変換する関数です.

そこで今度は `concat` 関数を使ってみた例を示しましょう.

—— ファイル B1 の内容 ——

```
A1:"set pm3d at bs;";
R1:"set xrange [-10:10];\
set yrange [-10:10];set zrange [7:12];";
H1:"set hidden3d;set isosamples 50;";
O1:"set output ";
T1:"set term png;";
Ox:concat(O1,"'Fig1.png'",",");
nekoneko:concat(A1,R1,H1,Ox,T1);
assume(x^2+y^2>0);
fxy:10~realpart(log(sqrt(x^2+y^2+1)));
plot3d(fxy,[x,-10,10],[y,-10,10],[grid,20,20],
[gnuplot_curve_titles,"title 'fxy',5+x*y/10 title 'test'"],
[gnuplot_preamble,nekoneko]);
```

この例では,O1 に `set output` を割当てており,これを雛形として,出力先のファイルを O1 に `concat` 関数で追加した文字列 Ox を利用しています. この方法を使えば,条件文やバッチファイルを生成するスクリプトで,ファイル名の切替えが容易に行えます. 又,このファイルでは,spplot に gnuplot の式を `gnuplot_curve_titles` を使って組込ませる様にしています.

更に,今度は Maxima に B1 を処理させる以下に示すシェルスクリプトを記述します.

```
#!/usr/local/bin/maxima

maxima -b B1>B1.log
convert Fig1.png Batch2.eps
display Fig1.png
less B1.log
```

このシェルスクリプトでは,最初に Maxima でファイル B1 の処理を実行します. その結果,画像ファイル Fig1.png が生成されますが,次に `convert` 命令を使って,この

Fig1.png を Batch2.eps に変換して, Fig1.png を display 命令を使って閲覧します. そして最後にバッチ処理の記録ファイルの B1.log を見るというものです. このバッチ処理で生成された画像ファイルを図 10.55 に示しておきます.

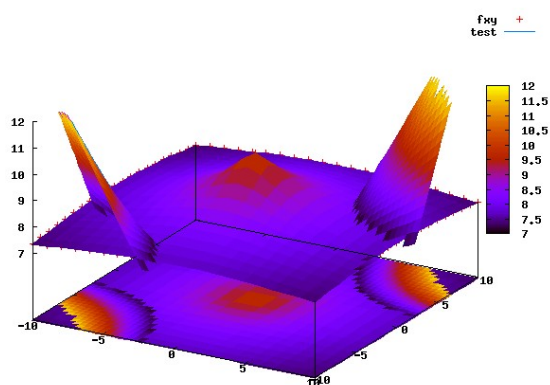


図 10.55: ファイル B1 の実行結果

Maxima のバッチ処理と gnuplot_preamble が揃えば, 後は, awk 等の適切な言語があれば十分複雑な処理が行えるでしょう.

10.7 draw パッケージ

10.7.1 draw パッケージの概要

Maxima では `plot2d` と `plot3d` 関数を主要なグラフ表示関数としていましたが, `gnuplot` を利用する関数として, `draw` パッケージが新たに追加されました. この `draw` パッケージは従来の `plot2d` 関数や `plot3d` 関数とは異なり, 外部アプリケーションとして利用可能なものは `gnuplot` のみ, それも 4.2 以降のものに対応しています.

`draw` パッケージには描画関数として `draw` 関数, `draw` 関数を簡易にした `draw2d` 関数と `draw3d` 関数があります. これらの関数は従来の `plot2d` 関数や `plot3d` 関数と違い, 式ではなく式等から構成された対象を表示する関数です.

`draw` パッケージで用いられる対象は, 表示する式の型によって様々な文を用いて構築されるものです. そして, これらの対象の表示, 例えば, 線分を実線, 或いは点線で表示するといった性質を変更させる為の指定, 即ち, 属性が指定可能となっています.

`draw2d` 関数と `draw3d` 関数は各々 2 次元と 3 次元グラフを描く為の関数ですが, これらの関数の実体は `draw` 関数です. この `draw` 関数を用いる事で, 一枚のウィンドウに複数の 2 次元や 3 次元の対象のグラフを同時に表示させる事が可能となっています.

`draw` パッケージではグラフの出力は `gnuplot` に任せますが, 基本的に, Lisp のストリームを用いる事で `gnuplot` の制御を行う方式を採用しています. その為, 中間ファイルは `plot2d` 関数や `plot3d` 関数と同様に, MS-Windows 以外は `maxout.gnuplot.pipes` になります. 猶, MS-Windows 環境であれば `maxout.gnuplot` となり, 名前から判る様に, Maxima 側からグラフデータだけではなく, `gnuplot` の全ての設定文や描画命令文を書出したファイルで, ストリームを用いない方式になっています. その為, MS-Windows 側から `gnuplot` を操作する事は出来ません.

10.7.2 対象と属性について

`draw` パッケージで実際の描画を行う関数である `draw`, `draw2d`, `draw3d` では, `plot2d` 関数の様に式をそのまま描画するのではなく, 式を対象として定義し, その対象を対象を描画する方式となっています.

ここでは従来の描画関数の `plot2d` や `plot3d` と対応のつく `draw2d` 関数や `draw3d` 関数を使って簡単な例題からこれらの関数の特徴を眺めてみましょう.

先ず, $\frac{\sin(x)}{x}$ を描画する時の `draw2d` の最も単純な構文を以下に示しておきましょう.

```
draw2d(explicit(sin(x)/x,x,-10,10));
```

この出力を以下の図 10.56 に示します。

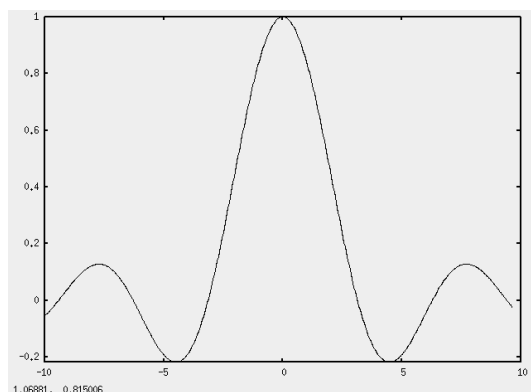


図 10.56: draw2d による単純なグラフ表示

この例では, draw2d の中に explicit 文の中に式が含まれています. この様に, 描画するものを含む文の事を対象と呼びます. この explicit 文には構文があり, 描画する式が $\frac{\sin(x)}{x}$ で, その式の変数が x で, この変数 x の定義域が $[-10, 10]$ である事を宣言しています. これだけだと, plot2d 関数に explicit を加えて複雑にした様にしか見えませんね.

では次の例を考えましょう.

```
draw2d(xrange=[-20,20],explicit(sin(x)/x,x,-10,10));
```

この結果を今度は図 10.57 に示しておきます.

この例では, 先程の explicit 文に加えて, xrange というものがありますね. こちらの xrange は explicit とは構文が異なり, $xrange = \langle \text{値} \rangle$ の書式になっています. この様な構文を持つ文をここでは属性と呼びます. 属性は演算子=の左辺に属性名を置き, 右辺にその値を指定します.

ここでの xrange 属性はグラフの枠に対し, その X 軸側の領域を定める属性です. draw2d 関数や draw3d 関数を利用する場合には, xrange 属性は大域的な属性の様に思えますが, draw 関数を用いると升目の様に複数のグラフを表示可能で, その様に表示したグラフの内の一つのグラフに対し, その枠に影響を及ぼす為, 局所的な属性になります. ところで, explicit で定めたグラフの定義域が $[-10, 10]$ なのに対して, xrange が $[-20, 20]$ となっている事に注意して下さい. その結果, 図 10.57 では, 式のグラフは X 軸側が $[-10, 10]$ を定義域とする値域のみしか描かれていませんが, グラフの枠は X 軸側が

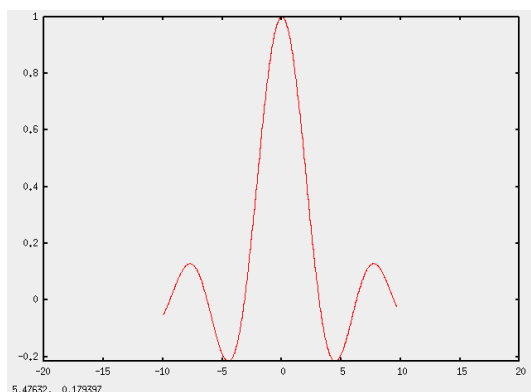


図 10.57: xrange による X 軸側の表示領域設定

$[-20, 20]$ に設定されています.

猶,xrange があるので, 二次元グラフの場合は yrange, 三次元の場合は zrange も当然ありますが, 例に示す様に, これらのグラフの枠の属性は Maxima 側に任せる自動設定にしても構いません.

上述の例では, 対象を定義する文や属性の式を draw2d 関数内部で記述していましたが, 対象や属性を一々描画関数内部で色々設定してやる必要はありません. 実際, 以下の様な方法でグラフを描画させる事も可能です.

```
(%i4) neko:explicit(sin(x)/x,x,-10,10);
                                sin(x)
(%o4)      explicit(-----, x, - 10, 10)
                                x

(%i5) X1:xrange=[-20,20];
(%o5)      xrange = [- 20, 20]
(%i6) Y1:yrange=[-0.5,1.5];
(%o6)      yrange = [- 0.5, 1.5]
(%i7) draw2d(X1,neko,Y1);
(%o7)      [gr2d(explicit)]
```

この例では,explicit といった対象に対し,xrange や yrange といった属性を Maxima 上で設定し, それらを纏めて,draw2d 関数を用いて表示したものです. この方法だと, 複数の対象を描く事が容易に行える事が理解出来るかと思います.

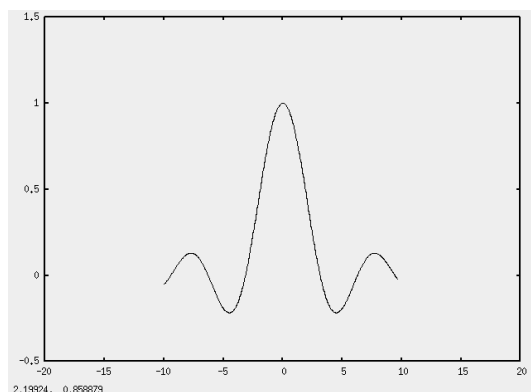


図 10.58: draw2d の表示例

さて, draw パッケージの属性には, 全体に影響を与える大域的な属性, グラフの枠や利用者定義の特定の対象に影響を及ぼす属性の二通りの属性があります. ここで, 大域的な属性に関しては, draw 関数, draw2d 関数や draw3d 関数の内部で, 他の対象や属性との順番は全く問題になりません. 次に, グラフの枠といった対象で, draw パッケージの描画関数で明示的に生成しない対象の属性の順番は比較的自由です. ここで比較的自由であるとは, draw2d 関数と draw3d 関数では自由に置けますが, draw 関数の場合には緩い制約が付く為です. これに対して, 曲線や曲面等の実際に描画される対象に影響を及ぼす属性は必ず対象の前に置かなければなりません. 即ち, 描画する対象とその属性は一つのグループとして扱う必要があります. これは draw パッケージの描画関数で, 前にある属性を解釈して gnuplot の設定文に翻訳し, 対象が現われた時点で, その対象を描画する命令と設定文を書込む為です. この方式を採用した理由としては, gnuplot の構文上の特性による影響が大きいと言えるでしょう.

では, 属性の対象に対する修飾の様子を以下の例で観察してみましょう.

```
(%i28) nekoneko:points([1,2,3],[1,2,3]);
(%o28)                points([1, 2, 3], [1, 2, 3])
(%i29) draw2d(xrange=[-1,6],
              yrange=[-1,6],
              nekoneko,
              point_size=3,
              point_type=diamant,
              points_joined=true,
```



```

        line_type=dots,
        color=red,
        nekoneko);
(%o29)          [gr2d(points, points)]

```

ここで用いられている対象 `points` は平面内の点列を定義する対象で, `point_size`, `point_type`, `points_joined`, `line_type` や `color` といった属性を持っています. これらの属性に関しては後の小節で詳細を述べますが, 各々, 点の大きさ, 点の型, 点列の繋ぎ型と色に関連する属性です. さて, 上記の入力から次の絵が得られます.

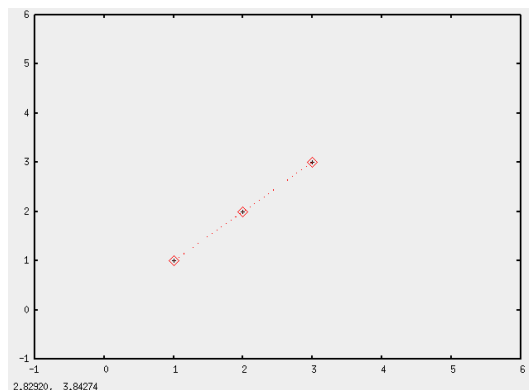


図 10.59: 点と属性の関係

この図に示す様に, 最初の点 `nekoneko` は後に続く対象 `points` の属性の影響を受けていませんね. ところが, 後の `nekoneko` は点の大きさが 3, 点の型が `diamant`, 各点を点線で結んでおり, 色も赤になっています. この様に, 最初の対象 `nekoneko` はデフォルトの属性のままですが, 次の対象 `nekoneko` には, 最初の `nekoneko` 以降に置かれた属性になっています.

10.7.3 draw 関数と draw2d 関数, draw3d 関数との関係

最初に `draw2d` 関数と `draw3d` 関数の実体は `draw` 関数であるとお話ししましたが, ここではもう少し詳細を述べる事にしましょう.

`draw2d` 関数と `draw3d` 関数は各々, 2次元と3次元の対象を専門に描く関数です. では, 2次元の対象と3次元の対象を画面を複数に分割して描く場合はどうすれば良い

でしょうか？残念ながら draw2d や draw3d では出来ません。しかし, draw 函数なら可能なのです。この理由は非常に単純な事です。実は, draw2d と draw3d は draw 函数に gr2d 文や gr3d 文を合成した書式になっています。

この様子を以下に纏めておきましょう。

— draw2d, draw3d と draw との関係 —

$\begin{aligned} \text{draw2d}(\text{対象}_1, \dots, \text{対象}_n) &\Rightarrow \text{draw}(\text{gr2d}(\text{対象}_1, \dots, \text{対象}_n)) \\ \text{draw3d}(\text{対象}_1, \dots, \text{対象}_n) &\Rightarrow \text{draw}(\text{gr3d}(\text{対象}_1, \dots, \text{対象}_n)) \end{aligned}$

ここで gr2d と gr3d は各々 2 次元と 3 次元の対象を纏めたものになります。先程の例で, explicit 文や points 文を持つ対象を描かせた時に, [gr2d(explicit)] や [gr2d(points, points)] と Maxima が返していましたが, その gr2d です。draw2d の場合は, gr2d 文, draw3d の場合は gr3d 文で対象を処理した事を示している訳です。

具体的には, Maxima の内部では draw2d 函数や draw3d 函数の引数に各々 (\$gr2d) や (\$gr3d) を Lisp の cons 函数で結合したものを draw 函数に引渡しています。そして, draw 函数内部では, その付加した情報を基に, make-scene-2d 函数や make-scene-3d 函数に引渡して個別のグラフ処理を行っています。

この draw2d 函数や draw3d 函数のみを用いていると, 大域的な属性と局所的な属性の扱いが少し見え難くなります。例えば, draw2d 函数や draw3d 函数で xrange の様なグラフの枠を定める属性は, 他の対象や属性とは独立して好きな場所に置けます。これは, draw2d では gr2d 文, draw3d で gr3d 文を用いて一つのグラフの枠が定義されているからで, その枠の内部で枠の設定を行うのは枠の中なら何処でも構わないという事に過ぎません。

ところが, draw2d や draw3d 函数では複数の gr2d 文や gr3d 文を扱う事は出来ません。これらの複数の文を処理する事が可能な函数が draw 函数で, この draw 函数を用いれば, xrange の様に見大域的な属性が, 実際は局所的な属性である事が明瞭になります。

10.7.4 draw 函数

さて, draw2d や draw3d の実体が draw 函数であると述べましたが, ここで, その詳細を述べます。

draw 函数は複数の gr2d 文や gr3d 文で構成された文の列を左から順番に表示する事が可能な函数です。この draw 函数では対象が 2 次元であるか 3 次元であるかに応じて, gr2d 文や gr3d 文を用います。但し, 同じ次元の対象を一つの gr2d 文や gr3d 文で記述するか, 複数の gr2d 文や gr3d 文で記述するかで結果が異なります。

最初に, 正弦函数と放物線を一つの gr2d 文で記述した場合の様子を以下に示しておきます.

```
(%i6) draw(gr2d(explicit(sin(x),x,-10,10),
                explicit(x^2+1,x,-2,2)));
(%o6)          [gr2d(explicit, explicit)]
```

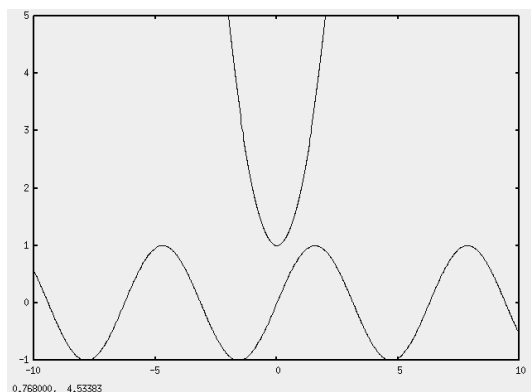


図 10.60: 一つの gr2d 文の場合

図 10.60 に示す様に, 一つのグラフの中に正弦函数と放物線が記述されていますね. 又, draw 函数の応答からも, 一つの gr2d 文の中に explicit が二つある事が判りますね. では次に, gr2d 文で正弦函数と放物線を別々にした場合を以下に示しましょう.

```
(%i7) draw(gr2d(explicit(sin(x),x,-10,10)),
            gr2d(explicit(x^2+1,x,-2,2)));
(%o7)          [gr2d(explicit), gr2d(explicit)]
```

図 10.61 から判る様に, 今度は別々にグラフが表示されていますね. 更に, draw 文の応答からも, gr2d 文が二つある事が判りますね. ここで, 最初の正弦函数のグラフが上で, 放物線のグラフが下に描かれていますね. draw 函数では gr2d 文や gr3d 文で構成される列を左から順番に描きます. 通常表示は一行にグラフを表示しますが, これを複数の列で表示させる事も可能です. この場合, 大域的属性の columns の値を変更すれば可能です.

では早速以下の例で解説しましょう.

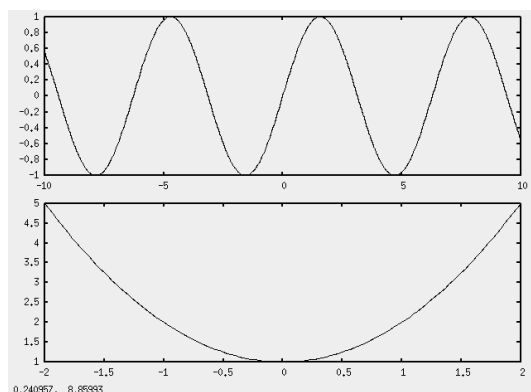


図 10.61: 二つの gr2d 文の場合

```
(%i15) draw(columns=1,gr2d(explicit(sin(x),x,-10,10)),
           gr3d(explicit(x*y,x,-2,2,y,-2,2)),
           gr2d(explicit(x^2-x+1,x,-2,2)));
(%o15) [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

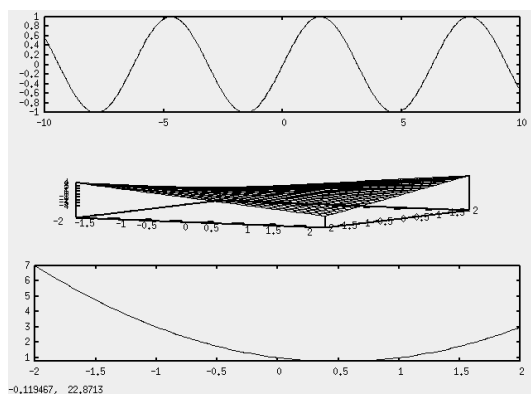


図 10.62: columns=1 の場合

ここでの例では、最初が正弦函数、次が xy 、最後が $x^2 - x + 1$ のグラフを描かせていますが、columns=1 の為、図 10.62 では正弦函数から順番に頭から一列に表示されていますね。では、columns を 3 にするとどうなるでしょうか？

```
(%i16) draw(columns=3,gr2d(explicit(sin(x),x,-10,10)),
           gr3d(explicit(x*y,x,-2,2,y,-2,2)),
           gr2d(explicit(x^2-x+1,x,-2,2)));
(%o16) [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

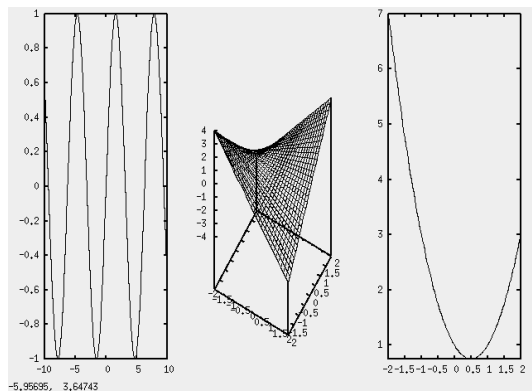


図 10.63: columns=3 の場合

図 10.63 で示す様に,3 列でグラフが左から順に表示されている事が判りますね.最後に,columns=2 とした場合の例を示しておきましょう.

```
(%i17) draw(columns=2,gr2d(explicit(sin(x),x,-10,10)),
           gr3d(explicit(x*y,x,-2,2,y,-2,2)),
           gr2d(explicit(x^2-x+1,x,-2,2)));
(%o17) [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

この様に,対象が 3 個の為,最初の 2 個が同じ行に表示されて,3 番目の対象が一段下に表示されていますね.猶,mouse が on 出来るのは,一番最後に描画される対象のみです.

この columns 文は gr2d や gr3d の外に,しかも,先頭に置っていますが,実際は何処にでも置ける文です.gr2d 文や gr3d 文の中に入れる事も可能です.この様に,draw 関数内部で何処にでも置ける属性の事を大域的な属性と呼びます.これに対し,個々の対象に関連する属性は,その対象が含まれている gr2d 文や gr3d 文の外に置く事は出来ません.例えば,xrange を draw 関数の中で columns の様に扱えません.

```
(%i6) draw(xrange=[-1,1],gr2d(explicit(sin(x)/x,x,-0.5,0.5)));
```

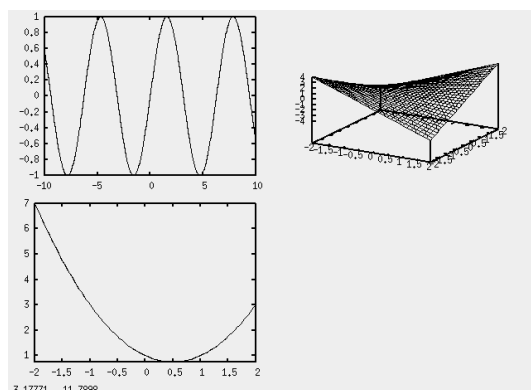


図 10.64: columns=2 の場合

Unknown global option xrange

-- an error. To debug this try debugmode(true);

```
(%i7) draw(gr2d(explicit(sin(x)/x,x,-0.5,0.5)),xrange=[-1,1]);
```

Unknown global option xrange

-- an error. To debug this try debugmode(true);

この例で判る様に, 属性 xrange は大域的な属性ではありません. この様に, 局所的な属性は draw 関数で記述した際に, gr2d 文や gr3d 文の外に出る事が出来ません.

さて, 以降の小節では draw パッケージで用意されている大域的な属性について述べましょう.

10.7.5 大域的な属性について

draw パッケージには以下の大域的な属性が用意されています.

大域的属性

columns	正整数値	1	グラフの列数
terminal			screen の値
pic_width	正整数値	640	画像ファイルの幅 (画素単位)
pic_height	正整数値	480	画像ファイルの高さ (画素単位)
eps_width	正整数値	12	画像ファイルの幅
eps_height	正整数値	8	画像ファイルの高さ
file_name	文字列	maxima_out	画像の出力ファイル名

columns は draw 関数で複数の gr2d 文や gr3d 文の列を描く時に、表示方法を指定するものです。通常、columns の初期値は 1 の為、columns に値を指定しなければ、draw 関数は gr2d 文と gr3d 文の列を左から順番に縦一列に表示します。より一般的に、columns を正整数値 n 、gr2d と gr3d の列を $a_1, a_2, \dots, a_m$ とすると、 $a_1, \dots, a_n$ を左から順番にグラフの 1 行目に、 $a_{n+1}, \dots, a_{2n}$ をグラフの 2 行目に描画してゆきます。猶、グラフの行数は $\frac{m}{n}$ を越える最小の整数が相当します。

猶、mouse が on になっているのは gr2d と gr3d の列を $a_1, a_2, \dots, a_m$ とする場合、 a_m のグラフのみです。その為、グラフの座標の読取、グラフの拡大・縮小、そして、グラフの回転は a_m のグラフに対してのみ実行され、グラフの拡大・縮小、回転によってグラフの再描画が行われた場合には、 a_m のグラフのみが再表示されて、他のグラフが消えてしまう事に注意しましょう。

terminal は gnuplot の terminal に対応する属性で、グラフ出力先の端末を指定する属性です。UNIX 環境でその初期値は通常 x11 となっていますが、この変数は計算機環境で著しく異なります。猶、JPEG、PNG、GIF や PostScript といった様々な画像形式でグラフを保存する場合、この変数を目的の画像形式に設定します。例えば、PNG 形式の画像ファイルにグラフを出力したければ、この terminal を `terminal=png` で PNG に指定します。この terminal の値は個々の環境の gnuplot で利用可能な端末に限定されます。MS-Windows に同梱されている wgnuplot では JPEG が扱えないので、MS-Word とかに利用したければ、PNG を用いると良いでしょう。又、その他にどのような値が設定可能かを知りたければ、gnuplot を立ち上げ、命令入力行に `? terminal` と入力すれば、terminal に関するオンラインマニュアルの内容が表示され、このオンラインマニュアルの末尾に利用可能な端末の一覧が表示されます。

例として、図 10.65 に `? terminal` と入力した時に表示されるオンラインマニュアルの末尾を示しておきます。

gnuplot の terminal にどの値が設定されているかを知りたければ、今度は gnuplot で

```

This document may describe terminal types that are not available to you
because they were not configured or installed on your system. To see a list of
terminals available on a particular gnuplot installation, type 'set terminal'
with no modifiers.

Subtopics available for terminal:
aeu512      aeu767      aifn          bitgraph
cgm         corel       dumb          dkf
eepic       emf         emtex         epslatex
epson-180dpi epon-60dpi  epson-1x800   fig
gif         gpdc        hp2623a       hp2648
hp500c      hpdl       hpdljll       hpdljll
hppj        lrasen     jpeg          kc-tek40xx
kn-tek40xx  latex      mf            nif
mp          nec-cp6    okidata       pbm
pcl5        png        pop            postscript
pslatex     pstex      psricks       push
qps         regis     selanar       stanc
svg         tandu-60dpi tek40xx       tek410x
texdraw     tgif      tkcanvas      tpic
vttk        x11       xlib

```

lines 4-26/26 (END)

図 10.65: ? terminal の結果

`show terminal` と入力します. すると terminal の値が表示されます. 例えば, Linux 上の gnuplot で show terminal を入力した様子を示しておきましょう.

```
gnuplot> show terminal
```

```
terminal type is x11
```

```
gnuplot>
```

因に MS-Windows 版に附属の wgnuplot で show terminal を実行すると, 結果として windows color nonenhanced が返されます.

pic_width, pic_height は EPS ファイル以外の書式でグラフをファイル出力する場合に, その画像の大きさを指定する属性値です. pic_width が画像の幅, pic_height が画像の縦方向の大きさを画素単位で指定します.

eps_width と eps_height は画像形式が PostScript や EPS の場合に設定します.

file_name は画像ファイルとして保存する場合, そのデータの保存先の名称です. 猶, 修飾子は Maxima 側でつけます. 例えば, terminal の値を PNG とし, file_name を tama にしていると, 画像ファイル名は tama.png になります.

以上が draw パッケージに含まれる大域変数です. 残りの属性は, グラフの枠に関連したものや, 描画される対象の見え方を指定するものになります. 次の小節では局所的な属性でグラフの枠に関連するものを解説しましょう.

10.7.6 グラフの枠に関連する属性

局所的な属性の特徴としては、draw 関数で描く際に、gr2d 文や gr3d 文の外に置く事が出来ない点が挙げられます。その中で、グラフの枠に関連する属性は gr2d 文や gr3d 文の中で、自由に置く事が出来る為、大域的な属性の様に見えるものです。しかし、その影響はその属性が置かれている gr2d や gr3d 内部を越える事はありません。

先ず、グラフの枠に関連する属性で大きさと表示方法に関連する属性を以下に示しておきます。

—— グラフの枠の大きさと表示に関連する属性 ——

xrange	二成分の実数リスト	自動	グラフの X 軸方向の枠を指定
yrange	二成分の実数リスト	自動	グラフの Y 軸方向の枠を指定
zrange	二成分の実数リスト	自動	グラフの Z 軸方向の枠を指定
axis_top	論理値	true	上側のグラフの枠表示を指定
axis_bottom	論理値	true	底のグラフの枠表示を指定
axis_right	論理値	true	右側のグラフの枠表示を指定
axis_left	論理値	true	左側のグラフの枠表示を指定
axis_3d	論理値	true	3 次元グラフの枠表示を指定

属性 xrange,yrange,zrange に関しては既に述べている様に、表示する領域、即ち枠の大きさを指定するものです。これらを実無指定にしておくと、関数側で自動的にその表示対象の大きさから指定を行います。例えば、対象 explicit や対象 explicit3d の場合、これらを定義する explicit から変数の定義域や値域を自動的に枠の領域として設定します。属性 axis_top,axis_bottom,axis_right,axis_left は二次元グラフでの上下、左右の枠の表示を行うかどうかを指定する属性です。例えば、axis_top が false の場合、グラフの上下、左右を囲む枠の上側が表示されません。同様に、axis_bottom が false であれば下側、以降、axis_right が false なら右側,axis_left が false なら左側の枠が表示されません。これと同様に、axis_3d が false の場合、三次元のグラフで枠が非表示になります。グラフの目盛に関連する属性もあります。これらの属性では目盛の有無や目盛を対数表示にするかどうか、そして、グラフ全体に網目を入れるかが指定出来ます。

—— グラフの軸の目盛に関する属性 ——

xtics	論理値	true	X 軸に目盛を入れる
ytics	論理値	true	Y 軸に目盛を入れる
ztics	論理値	true	Z 軸に目盛を入れる
logx	論理値	false	X 軸方向を対数目盛に設定
logy	論理値	false	Y 軸方向を対数目盛に設定
logz	論理値	false	Z 軸方向を対数目盛に設定
grid	論理値	false	網目の描画

属性 `xtics`, `ytics`, `ztics` は各々、グラフの枠に目盛を入れるかどうかを指定する属性です。true の場合に目盛を入れます。猶、各軸の目盛の属性は `axis_top` 等の枠の表示に関連する属性から独立しています。即ち、枠を非表示にしている場合、軸目盛の表示が true であれば、その軸の目盛は表示されます。

又、目盛の間隔を対数にする事も可能です。この場合、属性 `logx`, `logy`, `logz` の何れか必要な軸の属性を true にします。

属性 `grid` は 2 次元グラフの網目表示を指定する為の属性です。`grid` に true を指定すると、二次元グラフで目盛に対応する様に網目の表示が行われます。

グラフの表題や各軸のラベルに関連する属性を以下に示します。

—— グラフの表題等に関連する属性 ——

title	文字列	""	グラフの表題を指定
xlabel	文字列	""	グラフの X 軸のラベルを指定
ylabel	文字列	""	グラフの Y 軸のラベルを指定
zlabel	文字列	""	グラフの Z 軸のラベルを指定

ここでの属性は論理値ではなく、実際に表示する文字列になります。属性 `title` にグラフの表題を設定します。これは単純に `gnuplot` の `set title` 文に `title` の内容を引渡すだけです。

次の属性 `xlabel`, `ylabel` と `zlabel` は X, Y, Z 軸に設定するラベルを指定する属性です。これらも `gnuplot` の `set xlabel` 文等に各々の属性を引渡すだけです。

視点の変更やマウスで検出した座標値の保存に関連する属性もあります。

—— 視点の変更, 検出した座標値の保存に関連する属性 ——

rot_vertical	0 から 180 の間の数	60	Z 軸回りの回転角度
rot_horizontal	0 から 360 の間の数	30	水平の回転角度
xy_file	文字列	""	データ保存用のファイル名を指定

属性 rot_vertical や属性 rot_horizontal は gnuplot の set view 文に対応する属性です. 属性 rot_vertical は 0 から 180 の間の実数を設定することが可能です. 同様に, 属性 rot_horizontal には 0 から 360 の間の実数が設定可能です.

属性 xy_file はマウスによる座標の検出を行った結果を保存する為のファイルを指定します. このマウスによる座標の検出は, グラフが表示されているウィンドウ上の任意の点をマウスの左ボタンで押した時にウィンドウの左下に座標が表示されています. この値はキーボードから を押す事で, xy_files に指定したファイルの保存されます. 以上で, グラフの枠に関連する属性の説明を終えます. その他の局所的な属性に関しては, 対象に密接に関連する為, 対象の解説と共に, その属性の解説も行います.

10.7.7 対象について

描画関数で実際に表示されるものが対象です. 対象には様々なものがありますが, それは Maxima で計算した式は勿論の事として, その他に, 多角形や楕円もあります. 更に, これらの対象の色や塗り潰しといった, 各対象固有の属性があります.

ところで, 対象は各々固有の構文を持った文で生成されます. ここでは文毎に生成可能な対象を纏めて解説しましょう.

explicit 文

explicit 文で定義可能な対象は, 2 次元グラフの対象 explicit と 3 次元グラフの対象 explicit3d です.

対象 explicit は $y = f(x)$ の型の一変数の式のグラフを表現する対象です. この対象は以下に示す構文で記述されます.

—— 対象 explicit を表現する構文 ——

```
explicit(< 式 >, < 変数 >, < 最小値 >, < 最大値 >)
```

この場合, 第一引数に式, 第二引数に式の変数, それから, その変数の定義域を表現する最小値と最大値を各々第三引数と第四引数に指定します.

対象 explicit には以下の属性があります.

対象 explicit の属性

adapt_depth	正整数値	10	描画助変数
nticks	正整数値	30	点数
line_width	正整数値	1	曲線の太さ
line_type	solid, 又は dots	solid	曲線の型
color	文字列	black	曲線の色
filled_func	論理値	false	塗り潰しの指定の有無
fill_color	文字列	red	
key	文字列		凡例 (legend) を記述

属性 `adaptive_depth` は `explicit` のみで用いられる属性で, Common Lisp の関数 `adaptive-plot` で利用される属性です. ここでの値は `adaptive-plot` で描画を行う際の曲線分割の最大数で, 初期値は 10 です.

属性 `nticks` は曲線上の点数です. 曲線上の点は `explicit` の X 軸上の最大値と最小値を用いて, 始点を X 軸の最小値から間隔 $\delta = \frac{\text{最大値}-\text{最小値}}{\text{nticks}-1}$ で等間隔に配置され, これらの点を線分で結ぶ事でグラフが得られます. 従って, 複雑な曲線になればなる程, この `nticks` の値を大きくする必要がありますが, 直線に違い曲線であれば小さな値にしても構いません.

属性 `line_width` は直線の太さの属性になります. 初期値として 1 が設定されており, 大きな整数値を設定すると曲線が太くなります.

属性 `line_types` は曲線の型を属性です. この属性が取り得る値は `solid` と `dots` の二つだけです. 属性 `line_types` が `solid` の場合, 描画関数は実線を引きますが, `dots` の場合は破線になります. 猶, `gnuplot` には, `solid` であれば 1, `dots` であれば 0 を引渡しています. 属性 `color` は曲線の色属性です. 初期値として `black`(黒) が指定されています. 猶, 曲線と X 軸で囲まれた領域を塗り潰す場合には, 次の属性 `fill_color` や属性 `filled_func` の設定を行う必要があります.

属性 `fill_color` は曲線と X 軸で挟まれた領域を塗り潰す色の属性です. 初期値として `red`(赤) が指定されていますが, 実際に, 対象によって塗り潰しを行う為には, 別の属性 `filled_color` を `true` に設定しておく必要があります.

属性 `filled_color` は塗り潰しの指定を行う属性です. 属性 `filled_color` を `true` に設定した時に, 属性 `fill_color` で指定した色で塗り潰しが行われます. 猶, 赤で塗り潰す場合には, 属性 `filled_color` の指定をしなくても構いません.

属性 `key` は `gnuplot` の凡例 (legend) を設定する為の属性です. 要するに, `gnuplot` のグラフで右上側にどのグラフが何のグラフに対応するかが描かれる事がありますが, こ

れを設定する属性です.

例えば, 以下の様に入力した結果を図 10.66 に示しておきます.

```
(%i8) draw2d(line_type=solid,
             key="sin(x)/x",
             explicit(sin(x)/x,x,-10,10),
             line_type=dots,
             key="x^2-1",
             explicit(x^2-1,x,-3,3));
(%o8) [gr2d(explicit, explicit)]
```

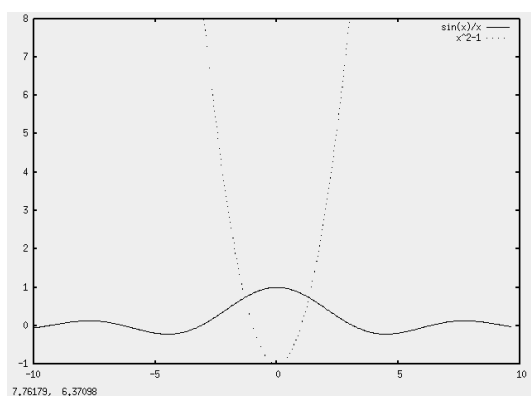


図 10.66: key による凡例

この図 10.66 の右上に, どれが $\frac{\sin(x)}{x}$ のグラフで, どれが $x^2 - 1$ のグラフであることを示す凡例が現われていますね. key はこの文字列を指定する属性になります.

対象 explicit3d は $z = f(x, y)$ の様な 2 変数の実数値関数として表現可能な式のグラフを表現します. 猶, この時の式の値域は, 絶対値が $1.75555970201398d+305$ を越えない領域でなければなりません.

—— 対象 explicit3d を生成する構文 ——

```
explicit(< 式 >, < 変数1 >, < 変数1 の最小値 >, < 変数1 の最大値 >, < 変数2 >, < 変数2
の最小値 >, < 変数2 の最大値 >)
```

explicit3d は基本的に対象 explicit を定義する explicit 文に第二変数に関連する引数が追加されただけです. 又, この対象 explicit3d の属性には以下のものがあります.

— 対象 explicit3d の属性 —

xu_grid	正整数値	30	X 軸方向の点数
yv_grid	正整数値	30	Z 軸方向の点数
contour	文字列		等高線の表示
contour_levels	正整数値	5	等高線の数
enhanced3d	文字列		
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線色
key	文字列		汎用 (legend) を記述

属性 xu_grid と属性 yv_grid は対象 explicit の属性 nticks に相当する属性で, 曲面の肌理の細やかさをここで指定する事になります.

属性 contour は等高線を描く位置を定める属性です. 属性 contour が取り得る値は, surface, base, both と map です. ここで, これらの値と gnuplot の命令との対応を以下に示しておきましょう.

— 属性 contour の値と gnuplot の命令文との関係 —

surface	set contour surface;set cntrparam levels contour_levels
base	set contour base;set cntrparam levels contour_levels
both	set contour both;set cntrparam levels contour_levels
map	set contour base;unset surface; set cntrparam levels contour_levels

猶, 対象 explicit の属性と同名のものは, 対象 explicit の同名の属性と同じです.

implicit 文

implicit 文は対象 implicit の生成で用います. この対象 implicit は $x^2 + y^2 - 1 = 0$ の様な 2 変数の陰関数のグラフの表示に用います.

— 対象 implicit の構文 —

```
implicit(< 式 >, < 変数1 >, < 変数1の最小値 >, < 変数1の最大値 >, < 変数2 >, < 変数2
の最小値 >, < 変数2の最大値 >)
```

ここで, 対象 `implicit` の属性を以下に示しておきます. 基本的に, 属性 `ip_grid` と属性 `ip_grid_in` 以外の属性は対象 `explicit` の同名の属性と同一です.

— `implicit` の属性 —

<code>ip_grid</code>	二成分の正整数リスト	<code>[50,50]</code>	点数
<code>ip_grid_in</code>	二成分の正整数リスト	<code>[5,5]</code>	
<code>line_width</code>	正整数	<code>1</code>	曲線の太さ
<code>line_type</code>	<code>solid</code> , 又は <code>dots</code>	<code>solid</code>	曲線の型
<code>color</code>	文字列	<code>black</code>	曲線色
<code>key</code>	文字列		凡例に用いる文字列

属性 `ip_grid` は対象 `implicit` で一次抽出で用いる網目を定め, 属性 `ip_grid_in` は二次抽出で用いる網目を定めます.

parametric 文

`parametric` 文は対象 `parametric` の生成で用います. ここで対象 `parametric` は媒介変数表示の平面曲線を描く際に用いる対象です.

— 対象 `parametric` の構文 —

```
parametric(< 式x>, < 式y>, < 媒介変数 >, < 媒介変数の最小値 >, < 媒介変数の最大値 >)
```

この対象 `parametric` の属性を以下に示しておきますが, 対象 `explicit` の属性と同名の属性と同じ働きをする為, ここでは詳細を述べません.

— `parametric` の属性 —

<code>nticks</code>	正整数値	<code>30</code>	点数
<code>line_width</code>	正整数値	<code>1</code>	曲線の太さ
<code>line_type</code>	<code>solid</code> 又は <code>dots</code>	<code>solid</code>	曲線の型
<code>color</code>	文字列	<code>black</code>	曲線の色
<code>key</code>	文字列		凡例 (legend) を記述

parametric3d 文

対象 `parametric3d` は媒介変数表示された空間内曲線の描画で用います. 構文そのものは, 対象 `parametric` を定義する `parametric` 文に Z 座標の媒介変数式が追加され

た書式になります.

—— 対象 parametric3d の構文 ——

```
parametric(⟨ 式x ⟩, ⟨ 式y ⟩, ⟨ 式z ⟩, ⟨ 媒介変数 ⟩, ⟨ 媒介変数の最小値 ⟩, ⟨ 媒介変数の最大値 ⟩)
```

対象 parametric3d の属性は対象 parametric と同じ属性を持っています.

—— parametric3d の属性 ——

nticks	正整数値	30	点数
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線の色
key	文字列		凡例 (legend) を記述

polar 文

polar 文は極座標表示された式のグラフを表現する対象 polar の定義で用いられます.

—— 対象 polar の構文 ——

```
polar(⟨ 動径の長さ ⟩, ⟨ 角度 ⟩, ⟨ 最小値 ⟩, ⟨ 最大値 ⟩)
```

対象 polar の属性は parametric の属性と同じものです.

—— polar の属性 ——

nticks	正整数値	30	点数
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線の色
key	文字列		凡例 (legend) を記述

parametric_surface 文

parametric_surface 文は媒介変数表示された空間曲面を表現する対象 parametric_surface を定義する際に用います.

この parametric_surface 文の構文を以下に示しておきます.

対象 parametric_surface の構文

```
parametric_surface(< 式x>, < 式y>, < 式z>, < 媒介変数1>, < 媒介変数1の最小値>, < 媒介変数1の最大値>, 媒介変数2>, < 媒介変数2の最小値>, < 媒介変数2の最大値>)
```

この parametric_surface の属性を以下に纏めておきますが, 対象 explicit3d に含まれるものです.

parametric_surface の属性

xu_grid	正整数値	30	X 軸方向の点数
yv_grid	正整数値	30	Z 軸方向の点数
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線色
key	文字列		汎用 (legend) を記述

points 文

points 文は平面上の点列を表現する対象 points と空間内の点列をを表現する対象 points3d を定義する為に用いられます.

points 文の構文

```
points(< 点列のリスト >)
```

対象 points を定義する場合, 点列リストの書式は $[[x_1, y_1], \dots, [x_n, y_n]]$ の様に点毎に定義するか, $[[x_1, \dots, x_n], [y_1, \dots, y_n]]$ の様に点列の X 座標と Y 座標のリストの二通りで表記する事が可能になっています.

これは対象 points3d も同様で, 対象 points3d の場合, 点列リストの書式として, $[[x_1, y_1, z_1], \dots, [x_n, y_n, z_n]]$ の様に三次元空間内の点として表現するか, $[[x_1, \dots, x_n], [y_1, \dots, y_n], [z_1, \dots, z_n]]$ の様に X, Y, Z 座標のリストで表現するか二通りが選べます.

対象 points 対象と points3d には以下の属性があります.

points と points3d の属性

point_size	正整数値		描画時の点の大きさを指定
point_type	文字列		描画時の点の型を指定
points_joined	論理値	false	描画時に点列を線分で繋ぐかどうかを指定
line_width	正整数値		描画時の点列を繋ぐ線分の太さを指定
line_type	solid 又は dots	solid	点列を結ぶ線分の型を指定
color	文字列	red	色を指定
key	文字列		凡例 (legend) を記述

polygon 文と rectangular 文

polygon 文は平面上の多角形を表現する対象 polygon の定義で用いられます. polygon 文では頂点を構成する点列を列記するだけなので, その表記方法は対象 points と全く同じです.

対象 polygon の構文

```
polygon(< 点列のリスト >)
```

又, 平面上の長方形は rectangular 文を使って, 対象 rectangular で表現する事も可能です.

rectangle 文の構文

```
rectangle([ $x_1, y_1$ ], [ $x_2, y_2$ ])
```

ここで, rectangular 文の構文で出てくる点 $[x_1, y_1], [x_2, y_2]$ の関係ですが, これらの点は定義する長方形の対角線上の点になります.

以下に対象 polygon と対象 rectangle の属性を以下に示しておきます.

polygon と rectangle の属性

transparent	論理値	false	透明にするかどうかを指定
border	論理値	true	描画時に境界を描くかどうかを指定
fill_color	文字列	red	塗り潰しの色を指定
line_width	正整数値	1	線分の太さを指定
line_type	solid 又は dots	dots	点列を結ぶ線分の型を指定
color	文字列	black	色を指定
key	文字列		凡例 (legend) を記述

transparent と border 以外は既に説明したものと同一属性の為、ここでは解説しません。先ず, transparent は多角形内部の塗り潰しを行うかどうかを指定するものです。次の border は多角形の境界を描くかどうかを指定するものです。基本的に多角形は fill_color が red で, transparent が false の為、何も指定しなければ赤で塗り潰されます。

ellipse 文

ellipse 文は楕円を表現する対象 ellipse を定義する為に用います。
ellipse 文の構文を以下に示しておきましょう。

ellipse 文の構文

```
ellipse(< 原点x>, < 原点y>, < 半径1>, < 半径2>, < 角度1>, < 角度2>)
```

対象 ellipse の属性

nticks	正整数値	30	総点数を指定
transparent	論理値	false	透明にするかどうかを指定
border	論理値	true	描画時に境界を描くかどうかを指定
fill_color	文字列	red	塗り潰しの色を指定
line_width	正整数値	1	線分の太さを指定
line_type	solid 又は dots	solid	点列を結ぶ線分の型を指定
color	文字列	black	色を指定
key	文字列		凡例 (legend) を記述

vector 文

vector 文は矢印を生成する為の構文です。平面上の矢印は draw パッケージで対象 vector で表現されており, 3次元空間内の矢印は対象 vector3d で表現されています。

先ず, 平面内の対象 `vector` の構文を以下に示しておきます.

2 次元空間内での対象 `vector` の構文

```
vector([< 基点の x 座標 >, < 基点の y 座標 >], [< 基点からの x 成分の増分 >, < 基点からの y 成分の増分 >])
```

対象 `vector` は 3 次元空間に置く事も可能です. この場合の構文は単純に平面の場合から, `z` 座標の部分を増やしただけです.

3 次元空間内部での対象 `vector` の構文

```
vector([< 基点の x 座標 >, < 基点の y 座標 >, < 基点の z 座標 >], [< 基点からの x 成分の増分 >, < 基点からの y 成分の増分 >, < 基点からの z 成分の増分 >])
```

猶, 3 次元空間でも, 対象 `vector` は 2 次元空間のものと同様に平面的に表示されます. 対象 `vector` と対象 `vector3d` は基点から増分で表現される向きに線分を引き, この増分側を向きにして, 矢印型を向けている事になります.

対象 `vector` の属性を以下に纏めておきます.

`vector` の属性

<code>head_angle</code>	正整数値	45	矢印の頭の角度
<code>head_both</code>	論理値	false	矢印の矢の設定
<code>head_length</code>	正整数値	2	矢印の柄の長さ
<code>head_type</code>	lilled, empty, nofilled	filled	矢印の頭の矢の型
<code>line_width</code>	正整数値	1	線分の太さを指定
<code>line_type</code>	solid 又は dots	solid	点列を結ぶ線分の型を指定
<code>color</code>	文字列	black	色を指定
<code>key</code>	文字列		凡例 (legend) を記述

対象 `explicit` の属性と同名の属性は同じ性質を持っています. それ以外の対象 `vector` 独自の属性は矢印の型に密接に関連するものばかりです.

image 文

`image` 文は対象 `image` を定義する為に用いられます.

— image 文の構文 —

```
image(<実数行列>, < $x_0$ >, < $y_0$ >, <幅>, <高さ>)  
image(<[ $r, g, b$ ] の行列>, < $x_0$ >, < $y_0$ >, <幅>, <高さ>)
```

— 対象 image の属性 —

colorbox	論理値	true	指定
palette	文字列		凡例 (legend) を記述

label 文

label 文は平面グラフ中の対象 label や 3 次元グラフ中の対象 label3d を定義する為に用います. 猶, label 文では複数の対象を同時に定義する事が可能です.

— 対象 label を定義する構文 —

```
label([<文字列1>, < $X$  座標値1>, < $Y$  座標値1>], ..., [<文字列n>, < $X$  座標値n>, < $Y$  座標値n>])
```

— 対象 label3d を定義する構文 —

```
label([<文字列1>, < $X$  座標値1>, < $Y$  座標値1>, < $Z$  座標値1>], ..., [<文字列n>, < $X$  座標値n>, < $Y$  座標値n>, < $Z$  座標値n>])
```

猶, 対象 label3d に含まれる文字列は常にグラフに対して正面を向く様に表示されます. その為, グラフを回しているとラベルが裏返ったりする事はありません.

— label の属性 —

属性名	取り得る値	初期値	概要
label_allnment	center, left, right	center	ラベルの位置合せ
label_orientation	horizontal, vertical	horizontal	ラベルの並び

— その他の関数 —

```
add_zeroes(<整数>)
```

10.7.8 環境の違いを調整する大域変数

draw パッケージは MS-Windows 環境でも UNIX 環境でも利用可能です。但し、MS-Windows 版では gnuplot ではなく、wgnuplot が同梱されていたり、ストリームを用いない方式となっている為、その他の環境との違いを吸収する際に、次の大域変数を用いています。

環境の違いを調整する大域変数

draw_pipes	true	Windows 環境の場合は false
draw_command	gnuplot	Windows 環境の場合は wgnuplot

先ず、大域変数 draw_pipes は plot2d や plot3d 函数でも採用された Lisp のストリームを用いる方式である事を示すものです。前述の様に、中間ファイルとして、maxout.gnuplot_pipes か maxout.gnuplot の何れかを用いる事が出来ますが、大域変数 draw_pipes が true であれば maxout.gnuplot_pipes を生成し、ストリームを用いて Maxima 側から gnuplot へ制御命令を送り込めます。しかし、大域変数 draw_pipes が false の場合、maxout.gnuplot ファイルを生成しますが、この中間ファイルを用いる場合、Maxima は gnuplot への描画データだけではなく、各種設定文や描画命令を含む maxout.gnuplot を生成し、そのファイルを gnuplot に引渡す方式となっています。その為、Maxima 側からは、データを生成した後の面倒は一切見ません。

UNIX 環境の場合は、true にも false にも出来ます。但し、false にすると、マウスを使った図形の拡大・縮小、三次元グラフの回転といったマウス操作が出来ません。更に、グラフ画面に `[m]` と入力しても、gnuplot の mouse が on になりません。その代わりに、別途、gnuplot を立ち上げて maxout.gnuplot ファイルを読込めば mouse を on にすればマウスによる操作が行えます。又、グラフ画面で `[m]` と入力する事で、mouse の on/off の切り替えが行えます。

猶、UNIX 環境でこの変数の値が true の場合、マウスによるグラフの操作も、gnuplot の mouse の on/off も可能です。

この変数の具体的な利用方法としては、グラフ描画の設定文を取り込んだ maxout.gnuplot ファイルを出力する為に使えます。この maxout.gnuplot ファイルは非常に便利なファイルで、このファイルさえあれば、ファイルを編集する事で、後は gnuplot 単体で色々な処理が行える為です。

次に、draw_command は MS-Windows では wgnuplot、その他の環境では gnuplot を用いる為に用いる大域変数です。独自の gnuplot を使いたい場合に、この変数にアプリケーション名を指定しておきます。その様な特殊な大域変数の為、この変数を弄る必要はまずないでしょう。

第11章 積分函数の動きを観察しよう

この章では非常に簡単ですが,Maxima の積分処理の流れに関して実際に Maxima がどのように解釈し,処理を行っているのか実例を示したいと思います. その為, 多少の LISP の知識 (この本の LISP についての内容) があれば良いでしょう. 計算機で数式を扱う事は一見人工知能に関係しそうですが, 実際は数値計算と同様に機械的な処理である事がなんとなく理解出来るかと思います.

11.1 積分函数ツアー募集要項

この章では Maxima の処理の流れを調べてみましょう. そこで, 何かと問題のある `integrate` 函数の動作を眺めてみる事にしましょう. 流石に, `integrate` 関数は色々と混み入っているので, 私が旗を持って観光旅行風に案内しましょう.

尚, このツアーに必要な事は以下の事です.

- `car`, `cdr`, `cons`, `append` 等のリスト操作の函数が理解出来る.
- `if` や `cond` が理解出来る.
- `lambda` 函数が理解出来る.

精々この程度ですが, 一番重要なものは好奇心です.

11.2 Maxima のソースファイル

Maxima のソースファイルは KNOPPIX/Math-2006 の場合, `/usr/share/maxima/5.9.2/src` にあります. Windows 版の場合, Maxima-5.9.2 を Maxima を Program Files にインストールしていれば, Program Files フォルダの Maxima-5.9.2 の中の `share\maxima\5.9.2` にある `src` フォルダに収録されています. この様に Maxima はバージョン番号のディレクトリ/フォルダの中に `src` ファイルが置かれています.

Maxima は Common Lisp で記述されてるので, ソースファイルは LISP のプログラムファイルです. このファイルは通常のテキストエディタで開いて閲覧する事が出来ます. 但し, ここでの作業では必ずソースファイルの複製を作って, 複製の方を書換える様にして下さい.

11.3 `integrate` 函数

Maxima の `integrate` 函数をこれから調べるのですが, `src` ディレクトリには沢山のファイルがあって, どのファイルに `integrate` 函数が定義されているか判らないでしょう. そこで, ファイル内部の検索を行います. この時, Linux や Cygwin 等の UNIX 環境であれば, `grep` 命令が使えます. この命令は指定した文字列が含まれる行をファイル名と一緒に表示する事が出来ます. そこで, 検索する文字列ですが, Maxima の表に出て来る函数や変数は Maxima 内部では頭に `$` が付いています. その為, `integrate` 函数の実体

は\$integrate 関数です. 尚,Maxima 側で直接操作しない関数や変数の頭に\$を付ける必要はありません.

そこで,`grep \ $integrate *.lisp`と入力しましょう. これで\$integrate を含む行と修飾子が.lisp のファイル名と一緒に表示します. その中で, 以下の表示がある筈です.

```
simp.lisp:(defmfund $integrate nargs
```

この grep では,:の左側にファイル名, 右側に\$integrate を含むファイル内の行が表示されています.defmfund は Common Lisp に無い関数ですが, 関数定義の命令の様ですね. 事実,\$integrate が含まれる他の行にこのような関数定義らしいものはありません. では,simp.lisp の該当箇所を抜出したものを以下に示します.

```
2156 (defmfund $integrate nargs
2157   (let ($ratfac)
2158     (cond ((= nargs 2)
2159           (with-new-context (context)
2160             (if (memq '%risch nounl) (rischint (arg 1) (arg 2))
2161                 (sinint (arg 1) (arg 2))))))
2162     ((= nargs 4) ($defint (arg 1) (arg 2) (arg 3) (arg 4)))
2163     (t (wna-err '%integrate))))
```

先程述べた様に Maxima 内部では Maxima の関数や変数には先頭に\$が付いています.Maxima で処理させる場合には,\$の事は全く考える必要はありませんが,LISP 側から処理する場合には, この事を忘れてはなりません.

先ず, 先頭が defmfund となっていますが, これは Maxima の関数定義に用いているマクロで, Common Lisp の defun と同じものと考えて構いません. 因に,Maxima では Macsyma が記述された LISP の構文に合せる為に色々なマクロを定義して用いています.

この\$integrate 関数の処理は上のリストを見ても判る様に cond 関数による場合分け以上の事をしていません. 尚,cond 関数は`cond((述語1)(関数1)... (述語n)(関数n)(T(関数n+1)))`の形で,述語_i ($1 \leq i \leq n$) の何れかが T であれば,関数_i を実行して cond 関数から抜けますが, 全ての述語が T にならない場合は最後の 関数_{n+1} を実行して終了する関数です. ここで,(T(関数_{n+1})) は不要であれば省略しても構いません.

では,\$integrate 関数の中にある cond 関数の内容を吟味してみましょう.

先ず,\$integrate 関数の引数が 2 個の場合に,rischint か sinint のどちらかを利用します. ここで, リスト nounl に '%risch が含まれている場合,Risch 積分を行う rischint 関数に引数が引渡されますが, それ以外は,sinint で積分が行われています. 変数が 4 個の場

合, 定積分を行う `defint` に引数を引き渡します. そして, 引数の個数が 2 個でも 4 個でもなければ, エラー処理が実行されます. 如何でしょうか?

さて, `integrate` 関数で Risch の積分を行う為には, リスト `nounl` に `'%risch` が含まれていなければなりませんね. では何処で `nounl` の設定が行われているのか調べてみましょう. この `nounl` の設定個所の探し方ですが, KNOPPIX や UNIX 環境があれば, `src` ディレクトリで `grep nounl *.lisp` と入力してみましょう. そうすると, 色々出てきますが, 変数に値の割当を行う `setq` 関数や `setf` 関数がある行に注目して下さい. すると以下の行があります.

```
mlisp.lisp:      (setq nounl (cons ($nounify fl) nounl)))
```

ここで左側がファイル名で, 右側に検索する文字列がある事を示しています. この事から `mlisp.lisp` に文字列に `nounl` を設定する行がある事を意味しています. そこで, ファイル `mlisp.lisp` を開いて該当個所を見ると, 実は, `ev` 関数の本体の処理である事が判ります. この `ev` 関数の定義は長いのでここでは示しません. 尚, `ev` 関数の詳細は 5.8.2 を参照して下さい.

因に, `ev` 関数での `nounl` の処理の意味は, Maxima の `nounify` 関数 (内部では `$notify`) を用いて関数名を名詞化した結果を LISP の `cons` 関数で `nounl` リストに追加する事です. この事から, `ev` 関数を用いると `nounl` に指定した名詞化された関数名が `cons` される事が判ります. この事から, `integrate` 関数を `ev` 関数を用いて評価する際に, `risch` を指定すれば `integrate` 関数は Risch 積分を実行すると判断されます.

そこで, 実際に動作を確認してみましょう. ここでの確認は単純に LISP の `print` 関数を `$integrate` 関数に埋め込んでみましょう. 最初に `src` ディレクトリから `simp.lisp` の複製を適当なディレクトリに置きます. それから `simp.lisp` の `$integrate` 関数の定義の個所に `print` 関数を二箇所埋め込みましょう. ここで表示させるのは `nounl` の内容とします. そこで, 具体的には `(print nounl)` を追加すれば良いのです.

以下に改造した `$integrate` 関数を示します.

```
(defmfun $integrate nargs
  (let ($ratfac)
    (print nounl)
    (cond ((= nargs 2)
      (with-new-context (context)
        (if (memq '%risch nounl) (and(print nounl)
                                       (rischint (arg 1) (arg 2)))
          (sinint (arg 1) (arg 2)))))))
```

```
((= nargs 4) ($defint (arg 1) (arg 2) (arg 3) (arg 4)))
(t (wna-err '$integrate))))))
```

次に、この関数の変更を Maxima に反映させる為には幾つかの処理が必要です。最も大袈裟な方法は Maxima をコンパイルして Maxima をインストールする事です。Maxima が C 等のコンパイラで記述されていればこうするしか方法はないでしょう。しかし、幾ら何でもこの方法は幾ら何でも大袈裟ですね。ところが、対話的処理が可能な Common Lisp で記述された Maxima ではもっと簡単に済ます方法があります。まず、一番簡単な方法は、Maxima を立ち上げて修正した simp.lisp を load 関数で読み込む方法です。

もう一つの安易ですが、場合によっては面倒な方法は、Maxima で `to_lisp();` と入力して Maxima の裏の LISP を表に出し、上記の `$integrate` 関数のリストを全て入力し、`(to-maxima)` と入力して Maxima に戻る方法です。こちらは Maxima で全て済ませる事が可能なのが長所ですが、長いプログラムの打ち込みを行うのであれば、ファイルの読み込みの方が楽です。

ここでは simp.lisp を書換えて、load 関数を使って読み込み、それから `ev(integrate(3^log(x),x),'risch);` と入力して Risch 積分が行われる事を確認してみましょう。

```
(%i11) load("./simp.lisp");
(%o11)                                     ./simp.lisp
(%i12) integrate(3^log(x),x);
```

NIL

```

              1
      (----- + 1) log(x)
      log(3)
3
-----
(%o12)              1
      (----- + 1) log(3)
      log(3)
```

```
(%i13) ev(integrate(3^log(x),x),'risch);
```

```
(%INTEGRATE %RISCH)
```

```
(%INTEGRATE %RISCH)
```

```

                                log(3) log(x)
                                x %e
(%o13) -----
                                log(3) + 1

```

この様に, ev 関数を利用しない場合, noun1 変数にはデフォルト値の NIL の為に sinint 関数が実行されている事が判りますね. 次に, ev 関数で 'risch を指定すると, noun1 にリスト (%INTEGRATE %RISCH) が割当てられています. noun1 に '%RISCH が含まれているので, (memq '%risch noun1) が T となるので rischint が実行されます.

ここでは内部変数の動きを見る為に, integrate 関数を改造しましたが, この様に Maxima では関数の修正や改造は簡単に出来てしまうのです.

では, もう一方の sinint 関数を調べてみましょう. この sinint 関数は sin.lisp の中で定義されています. 該当箇所を以下に示します.

```

(defmfun sinint (exp var)
  (find-function 'ratint) ; Make sure that RATINT is in core.
  (cond ((mnump var) (merror "Attempt to integrate wrt a number: ~:~M" var))
        (($ratp var) (sinint exp (ratdisrep var)))
        (($ratp exp) (sinint (ratdisrep exp) var))
        ((mxorlistp exp)
         (cons (car exp) (mapcar #'(lambda (y) (sinint y var)) (cdr exp)))))
  ((mequalp exp)
   (list (car exp) (sinint (cadr exp) var)
         (add2 (sinint (caddr exp) var)
               (concat '$integrationconstant
                       (setq $integration_constant_counter
                             (f1+ $integration_constant_counter)))))))
  ((and (atom var) (isinop exp var)) (list '(%integrate) exp var))
  ((let
    ((ans (simplify
            (let ($opsubst varlist genvar stack) (integrator exp var)))))
     (if (sum-of-intsp ans) (list '(%integrate) exp var) ans))))

```

この関数も基本的には cond 関数による分岐を持った関数です. 先ず, 最初の mnump 関数は変数 var が数値の場合に true を返す Maxima の真偽関数です. この箇所は変数が数値の場合のエラー処理を行う箇所です. 実際, 以下の様な場合の処理を行います.

```
(%i1) integrate(sin(x),3);
Attempt to integrate wrt a number: 3
```

次の二つの \$ratp 関数で変数 var と式 exp が各々 CRE 表現の場合の処理を定めます. \$ratp 関数は Maxima の ratp 関数です. この関数は引数が CRE 表現であれば true を返す真偽関数です.

そして mxorlistp 関数は exp がリストか行列の場合に NIL にならない関数です. ここでの処理では各成分に sinint 関数を作用させます. この個所で, lambda 関数を用いて, 変数に y に対して (sinint y var) を対応させる関数を定め, その関数を式 exp の内部表現の頭の部分を除いた行列やリストの各成分に mapcar 関数を使って作用させています. この処理の御陰で以下の様に関数をリストや行列で与えても計算が行えるのです.

```
(%i2) integrate([sin(x),cos(x),exp(x)],x);
                                     x
(%o2)          [- cos(x), sin(x), %e ]

(%i3) integrate(matrix([sin(x),cos(x),exp(x)], [tan(x),1/x,log(x)]),x);
                                     x
(%o3)          [  - cos(x)    sin(x)    %e      ]
               [                                     ]
               [ log(sec(x))  log(x)  x log(x) - x ]
```

mequalp 関数は式 exp に二項関係子=が含まれた場合に T となる関数です. ここでの処理は, exp の右辺と左辺に sinint を作用させ, 右辺に積分定数を加える処理を行っています. この積分定数の処理で用いられている concat 関数は, integrationconstant に大域変数 integration_constant_counter の値を付加するものです. 即ち, ここの処理で, 方程式の積分を行うと右辺に integrationconstant1 等の積分定数を意味する文字列を追加する処理を行っているのです. 随分と機械的ですね… それから, var がアトムで, (isinop exp var) が T の場合に積分を名詞形で返します. それ以外の場合は integrator 関数を用いて積分を実行します.

このことから, 問題の処理は integrator 関数で生じている事になります. ところで, integrator 関数は大きなプログラムになるので, 実際の式の積分で動きを見てみましょう

う. 式は $\sqrt{x + \frac{1}{x} - 2}$ とします. この式は, Maxima-5.9.2 でも上手く積分が出来ない式です.

```
(%i50) integrate(sqrt(x+1/x-2),x);
              3/2
          2 x  - 6 sqrt(x)
(%o50)  -----
              3

(%i51) integrate(sqrt(factor(x+1/x-2)),x);
          /
        [ abs(x - 1)
(%o51)  I ----- dx
        ] sqrt(x)
          /

(%i52) assume(x>1);
(%o52) [x > 1]
(%i53) integrate(sqrt(factor(x+1/x-2)),x);
              3/2
          2 x  - 6 sqrt(x)
(%o53)  -----
              3

(%i54) assume(x<1);
(%o54) [x < 1]
(%i55) integrate(sqrt(factor(x+1/x-2)),x);
              3/2
          2 x  - 6 sqrt(x)
(%o55)  -----
              3
```

この例で示す様に, 単純に積分すると $\frac{2x^{\frac{3}{2}} - 6\sqrt{x}}{3}$ を返しますが, この結果は $x \geq 1$ の場合は正しくても, $x < 1$ の場合は嘘になります.

与式は $\sqrt{x + \frac{1}{x} - 2} = \sqrt{\frac{(x-1)^2}{x}}$ となるので, この計算間違いの大きな原因は, $\sqrt{(x-1)^2}$ を安易に $x-1$ と置換えた為に生じたものと思われます.

先ず, $\sqrt{x + \frac{1}{x} - 2}$ と $\frac{|x-1|}{\sqrt{x}}$ の内部表現の違いを確認しておきましょう.

```
(%i69) exp1:sqrt(x+1/x-2);

                                1
                                sqrt(x + - - 2)
                                x

(%i70) exp2:sqrt(factor(x+1/x-2));

                                abs(x - 1)
                                -----
                                sqrt(x)

(%i71) :lisp $exp1
((MEXPT SIMP)((MPLUS SIMP) -2 ((MEXPT SIMP) $X -1) $X)((RAT SIMP) 1 2))
(%i71) :lisp $exp2
((MTIMES SIMP) ((MEXPT SIMP) $X ((RAT SIMP) -1 2))
 ((MABS SIMP) ((MPLUS SIMP) IRREDUCIBLE) -1 $X)))
```

ここでは、演算子:lisp を用いて Maxima の式の内部表現を表示させています。演算子:lisp は空行から後の改行までの入力を裏の LISP に流し込む演算子です。ここでは、式 exp1 と式 exp2 の LISP 内部の名前 (頭に\$が付きます) を入力して、そのエコーバックを調べます。すると exp1 の内部表現の caar が MEXPT となりますが、exp2 の内部表現の caar が MTIMES になる事に注意しましょう。

何故なら,integrator 函数には以下の処理があるからです。

```
224      (setq y (cond ((eq (caar exp) 'mtimes) (cdr exp)) (t (list exp))))
```

この処理は与式 exp の主演算子が mtimes であれば、その内部表現の cdr を取り、そうで無ければ、そのままをリストにするという処理です。この処理は integrator で実際の積分処理を行う函数の分岐に関係します。

11.4 integrate_use_rootsof を用いた積分

Maxima の有理式の記号積分で、大域変数 integrate_use_rootsof があります。この大域変数を true に設定すると、有理式の積分で、分母が一次式の積に分解出来ないものでも記号積分が出来てしまいます。ここでは実際の動きを見てみましょう。

```
(%i53) integrate_usr_rootsof:false$
(%i54) ans:integrate(1/(x^3-x^2-x+4),x);
      /
      [      1
(%o54)  I ----- dx
      ] 3    2
      / x  - x  - x + 4
(%i55) integrate_usr_rootsof:true$
(%i56) ans:integrate(1/(x^3-x^2-x+4),x);
      ====
      \      log(x - %r7)
(%o56)  > -----
      /      2
      ==== 3 %r7  - 2 %r7 - 1
            3    2
      %r7 in rootsof(x  - x  - x + 4)

integrate_usr_rootsof が false の場合,  $\frac{1}{x^3 - x^2 - x + 4}$  の積分は失敗しています. ところか, integrate_usr_rootsof が true の場合はちゃんと計算をしていますね. ここで, 計算結果 ans の内部表現を sum(x^2,x,0,2) と比較してみましょう.

(%i57) :lisp $ans
((%LSUM SIMP)
 ((MTIMES SIMP)
  ((MEXPT SIMP)
   ((MPLUS SIMP) -1 ((MTIMES SIMP RATSIMP) -2 $R7)
    ((MTIMES SIMP) 3 ((MEXPT SIMP RATSIMP) $R7 2)))
   -1)
  ((%LOG SIMP) ((MPLUS SIMP) ((MTIMES SIMP) -1 $R7) $X)))
 $R7
 (($ROOTSOFF SIMP)
  ((MPLUS SIMP) 4 ((MTIMES SIMP RATSIMP) -1 $X)
   ((MTIMES SIMP) -1 ((MEXPT SIMP RATSIMP) $X 2))
   ((MEXPT SIMP RATSIMP) $X 3))))
(%i57) test:'sum(x^2,x,0,4);
```



```

====
\      2
>    x
/
====
x = 0

(%i58) :lisp $test
((%SUM SIMP) ((MEXPT SIMP) $X 2) $X 0 4)

```

'sum(x²,x,0,4) の内部表現と計算結果を比較すると, 計算結果に (\$ROOTSOFF SIMP) が第一成分のリストが, 変数の上限と下限を設定する個所に置かれています. では,integrate-use-rootsof を true にするとどのような処理が実行されるのでしょうか調べてみましょう.

そこで, `grep integrate-use-rootsof *lisp` を src ディレクトリ上で実行してみましょう. すると, sinint.lisp に, この大域変数が含まれている事が判ります. では, 該当箇所を以下に抜き書きしてみます.

```

(defvar $integrate-use-rootsof nil
  "Use the rootsof form for integrals when denominator does not factor")

(defun integrate-use-rootsof (f q variable &aux qprime ff qq
  (dummy (make-param)) lead)
  ;; p2e is squarefree in polynomial in cre form p1e is lower degreee
  (setq lead (p-lc q))
  (setq qprime (disrep (pderivative q (p-var q))))
  (setq ff (disrep f) qq (disrep q))
  '((%lsum) ((mtimes)
    ,(div* (mul* lead (subst dummy variable ff))
      (subst dummy variable qprime))
    ((%log) ,(sub* variable dummy))) ,dummy
    (($rootsof) ,qq)
  )
)

```

integrate-use-rootsof が true の場合,integrate-use-rootsof が用いられます. ここでの処理は実の所, 決った雛形に式を当て嵌めているだけです.lead や qprime に ff の処理

はその当て嵌める為の式を生成しているだけで、それを ‘(%lsum) で開始するリストに当て嵌めています。このリストの中に、((\$rootsof), qq) がありますが、この関数 \$rootsof は他では定義されていないダミー関数です。又、Maxima の sum 関数には、リストの形式で与えられた助変数の集合に対する処理は行えません。その意味でも、ここでの処理は非常に形式的な処理です。

但し、内部の計算は出来ているので、以下の様な処理を行えば、最終的な結果を得る事が可能です。

```
(%i1) integrate_use_rootsof:true;
(%o1) true
(%i2) ans:integrate(1/(x^3-x^2-x+4),x);
====
\      log(x - %r1)
>  -----
/      2
====  3 %r1  - 2 %r1 - 1
      3      2
      %r1 in rootsof(x  - x  - x + 4)
(%i3) r1s:subst(%r1,x,solve(x^3-x^2-x+4,x))$
(%i4) ans1:substpart("+",map(lambda([z],ev(inpart(ans,1),z)),r1s),0)$
(%i5) diff(ans1,x)$
(%i6) trigsimp(%);

1
-----
3      2
x  - x  - x + 4
```

この方法では大域変数 integrate_use_rootsof を true にして式の積分を行います。その結果を ans に割当てて、sum の中身を part 関数を使って、part(ans,1) で取ります。次に、与式の分母の解を solve 関数を用いて計算します。この与式の分母は 3 次式の為に、厳密解を計算する事が出来ませんが、5 次以上の方程式の場合、algsys 関数を用いる事になるでしょう。ここで algsys 関数は厳密解が計算出来れば、厳密解を計算して返し、無理な場合は近似解を返す関数です。この際に変数 x を積分結果で用いられている解 %r1 に置換えます。それから、lambda 関数を使って、ev 関数による評価を行い、substpart でリストから和に変換させています。この結果 ans1 は長い式になるので非表示にしてい

ます. 次に, 微分を行い, `trigsimp` 函数を使って式を整理すると, 元の函数が出ているので, 兎に角出来ている事が判ります.

この様に, エレガントとは程遠い, とてもエレファントな計算をさせています. 勿論, この計算では式の極を考えていないので形式的な結果です,

如何でしょうか. 数式の積分は計算機の中の大勢の小人さん達が捻り鉢巻で計算している様に感じていた方も多いと思いますが, 実際の処理は, 数値計算で方程式を解く時に, 色々な定式化に基いて計算処理を行うのと同様に, 数式処理でも定式化に基づいて記号の処理を行っているのです. 計算機の中に小人が居るといったロマンティックな事は残念ながらありません.

第12章 Maximaの簡単な改造

12.1 使い勝手の向上を目指して

Maxima は非常に面白いシステムです. 特に面白い事が改造が容易に出来る事です. この安易な改造を行う為に, 貴方が LISP のエキスパート (LISPer) である必要は全くありません. 兎に角, Maxima のソースファイルを活用すれば良いのです.

実際, Maxima は単純に数式を処理するだけではなく, 様々な入出力を行っています. その中で, グラフィックスでは外部のアプリケーションを立ち上げたりもしている程です. その為, Maxima で何かを最初から作り上げる必要性は意外に無く, 似た処理を行う函数を観察して, その函数の仕組みを真似たり, 場合によってはその函数の複製を作ったり, 挙句の果てには, その函数を乗っ取ってしまっても構わないのです.

勿論, 乗っ取りを行えば, 他との互換性が崩れてしまうので, あまり勧められませんが, より単純には, 函数の複製を作ってしまうれば良いのです.

さて, Maxima の Help システムはそれなりの良く出来ています. しかし, 最近の数式処理システムでは HTML 等を利用するのが主流で, texinfo を用いる方法は事実上, 過去の方法です. そこで, 今風に HTML や PDF が表示出来れば如何でしょうか? 恐らく, 使い易くなる筈ですね. その様な動機で describe 函数を改造してみましょう.

12.2 describe 函数は何処にある?

先ず, describe 函数を改造したければ, describe 函数の所在を明確にし, それがどのようなものであるかを知っておく必要があります.

describe 函数の探し方は色々ありますが, UNIX 環境であれば, Maxima のソースファイルが収録されたディレクトリ src に移動して, `grep describe *.lisp|less` を実行してみましょう. 又, MS-Windows 版でも Maxima のソースファイルが付属しているので, 何らかのアプリケーションを使って探すのも良いでしょう.

では以下に結果を一部示しておきましょう.

```

ifactor.lisp;;; Elliptic curve factorization method as described in the paper
init-cl.lisp:(defvar $help "type 'describe(topic);' or 'example(topic);' or '? topic'"
limit.lisp;;; I believe a large portion of this file is described in Paul
macdes.lisp:(defmspec $describe (x)
macdes.lisp:(defmspec $describe (x)
mdebug.lisp:      ;; Process "?" lines. This is either a call to describe or a
mdebug.lisp:      '(((displayinput) nil (($describe) ,line $exact))))
mdebug.lisp:      '(((displayinput) nil (($describe) ,line $inexact))))
nparselisp;;; the comments in GRAM package, which describe them in reasonable detail.
nset.lisp;;; difficult to describe -- for an odd number of arguments, we favor the left side of the tree.
option.lisp:      ((eq '$describe (caar ans)) (mdescribe (decode (cadr ans))))
option.lisp:(subc $describe (c))
option.lisp:(subc $general-info () $describe $example $options $primer $apropos)
option.lisp:(subc $options (c) $down $up $back $describe $exit)
option.lisp:(subc $user-aids () $primer $describe $options $example $apropos $visual-aids)
rand-mt19937.lisp;;; The initialization routine is described below. Let s be the seed,
residu.lisp:      ;; described in bug 1073338.)
schatt.lisp;;; I think this is described in Chapter 3 of J. Moses' thesis,
lines 1-18

```

今回,Maxima-5.13.0 を利用しているので,他の版では多少出力が違っているかもしれませんが.しかし,探し方は単純で,Maxima の関数定義では defun や defmspec 等が用いられているので,\$describe と一緒にこれらの文字列が出ている箇所を捜せば良いのです.すると, `macdes.lisp:(defmspec $describe (x))` という行があるので,この事から describe 関数は macdes.lisp で定義されている事が判ります.

そこで今度は macdes.lisp の複製を作って,それを適当なエディタを用いて編集してみる事にしましょう.

12.3 describe 関数の動作

先ず,describe 関数の定義式を以下に示してしておきます.猶,Maxima-5.13.0 の describe 関数なので,他の版では多少異っているかもしれません.

describe 関数の定義

```

(defmspec $describe (x)
  (let
    ((topic ($sconcat (cadr x)))
     (exact-p (or (null (caddr x)) (eq (caddr x) '$exact))))
    (cl-info::*prompt-prefix* *prompt-prefix*)
    (cl-info::*prompt-suffix* *prompt-suffix*))
    (if exact-p
      (cl-info::info-exact topic)
      (cl-info::info topic))))

```

先ず,defmspec は maxmac.lisp で定義された Maxima のマクロで,Common Lisp で関数の定義に用いられる defun の様に \$describe 関数 (Maxima 上では describe 関数) を

定義するものです.

この describe 関数を定義する S 式の構造を以下に示しておきましょう.

describe 関数の構造

```
(defmspec 関数名 (変数1, ..., 変数n)
  (let ((内部変数1 値1) ... (内部変数m 値m))
    (処理1) ... (処理k)))
```

この describe 関数の処理を簡単に解説すると, 最初に与えられた変数 x に束縛された Maxima の文字列に対し, (`$sconcat (cadr x)`) を処理し, その結果を変数 `topic` に束縛させます. 次に, (`caddr x`) の値が空, 或いは '`$exact` に等しい場合に変数 `exact-p` を `T` にし, それ以外は `NIL` にします. それから `cl-info` パッケージの変数 `*prompt-prefix*` と変数 `*prompt-suffix*` に, `*prompt-prefix*` と `*prompt-suffix*` の値を束縛させます. 次が `if` による分岐処理になります. ここでは `exact-p` の値が `T` であれば, `cl-info` パッケージの `info-exact` 関数に `topic` の値を引き渡します. `NIL` であれば `cl-info` パッケージの `info` 関数に `topic` の値を引き渡します. 通常の処理では, `info-exact` に `topic` の値が引渡されて, `texinfo` によるマニュアルが表示される事になります.

12.4 関数 ponpoko の仕様

さて, describe 関数の動作は大体判りました. 今度はどの様に改造するかを考えてみましょう. 先ず, 最近のアプリケーションは困った時には, HTML ブラウザを使ってヘルプを表示する事が可能ですね. 又, PDF も表示出来れば良いですね.

そこで, describe 関数を基にして作る関数の名前は, そのオリジナリティの無さと, それにも関わらず「打ては響く」様な動作を期待して, ここでは狸らしく `ponpoko` としましょう. 又, この命令を書き込むファイル名を `ponpoko.lisp` としましょう.

この関数 `ponpoko` は, 関数名を指定した場合に通常の `info` ファイルを用いたヘルプを表示します. そして, 文字列として `html` を指定すると HTML のマニュアルを表示し, `pdf` を指定すると PDF のマニュアルを表示するものとしましょう. 猶, HTML マニュアルと PDF マニュアルは私のホームページに置いてあるものを用います¹. 又, `KNOPPIX/Math` の方は, `/usr/local/Math-ja/PDF` にあるマニュアルが使えます. 但し, `ponpoko.lisp` で指定するファイル名とパスは変更して下さい.

¹<http://www.bekkoame.ne.jp/~ponpoko/Math/books/ManualBook.tgz>,
<http://www.bekkoame.ne.jp/~ponpoko/Math/books/ManualBook.pdf>

12.5 大域変数の作り方

ここで HTML ブラウザとしては,Firefox,Opera に w3m, そして,Internet Explore と色々ありますね.PDF も同様に,acroread や xpdf に gv もあり,環境によって色々切り替えられるべきでしょう.又,HTML ファイルや PDF ファイルの格納先も別途定めておく必要がありますね.

この事から,必要となる大域変数はブラウザとファイルの格納先の 4 個になります.そこで,Maxima 側の変数名を hbrowser で HTML ブラウザ,pbrowser で PDF ブラウザ,jhtmlmdir で HTML ファイルの格納先,jpdfdir で PDF ファイルの格納先を指定する事にします.

そこで,他の関数でも用いられている大域変数で,これらのブラウザを切り替える事にします.ここで Maxima 内部で大域変数を定義したい場合,defvar マクロを用いて定義します.

defvar マクロ

```
(defvar 変数名 値)
```

ここで,defvar は LISP のマクロで,スペシャル変数として変数の宣言を行い,初期値の設定を行うものです.猶,似たマクロに defparameter がありますが,defvar の場合は,宣言した変数が既に値を持っていた場合には,その値が変更される事はありませんが,defparameter の場合は新しい値で置換えられます.Maxima では Maxima から扱う必要のある大域変数の宣言で,defvar が利用され,defparameter は Maxima 内部で利用者が設定する必要の無い,システムの環境変数の宣言で主に利用されています.

ここで,Maxima の変数の内部表現は,先頭に\$が付きます.その為,hbrowser は LISP 側では\$hbrowser になります.この事を留意して大域変数を定義しましょう.

ponpoko.lisp での大域変数の定義

```
(defvar $hbrowser "firefox")
(defvar $pbrowser "xpdf")
(defvar $jhtmlmdir "Maxima/ManualBook/")
(defvar $jpdfdir "Maxima/ManualBook/Book-New/")
```

ここで,\$jhtmlmdir と \$jpdfdir は私の環境の値を設定しています.この値は皆さんの環境に合わせて指定する様にして下さい.

ここで注意する事として,これらの変数には LISP の文字型のデータが束縛されている事です.注意すべき事として,Maxima の文字型は LISP の文字型では無い事が一番に

挙げられます。ここで Maxima の文字列から LISP の文字列に変換する場合, stringproc パッケージに含まれている関数 lstring が使えます。

さて, 次に関数の定義を行う事にしましょう。今回は describe 関数の実質的な乗取りが目的なので, 引数から html や pdf が得られた時に, こちらの処理を挿入すれば良いのです。その為, LISP の cond 関数を用いた処理が妥当でしょう。

即ち, 以下の様な関数のイメージになります。

—— 関数 ponpoko のイメージ ——

```
(defmspec $ponpoko (x)
  (let
    ((topic ($sconcat (cadr x)))
      (exact-p (or (null (caddr x)) (eq (caddr x) '$exact))))
    (cl-info::*prompt-prefix* *prompt-prefix*)
    (cl-info::*prompt-suffix* *prompt-suffix*))
  (cond
    ((topic に"html"の情報がある場合)
     (hbrowser を起動しろ))
    ((topic に"pdf"の情報がある場合)
     (pbrowser を起動しろ))
    (t
     (if exact-p
        (cl-info::info-exact topic)
        (cl-info::info topic))))))
```

本来の describe 関数にこちらの要求を取り込んだらこの様な形になります。すると, topic がどのような書式なのか。そして, 外部のアプリケーションに大域変数の値を組合せたパラメータを引き渡して起動させれば良いのかが問題になりますね。今度はこれらの課題を吟味してみましょう。

12.6 判別について

変数 topic には (\$sconcat (cadr x)) の値が束縛されます。ここで, 関数名に \$ が付いている事に注目しましょう。何故ならこれは Maxima の関数になるからです。従って, 後の (cadr x) は引渡された変数に束縛された値が Maxima のデータ型の為, LISP 内部で処理する為に不要な箇所を落す為の処理になるからです。この sconcat 関数は与え

られた複数の文字列を結合して、LISP の文字列型に変換する函数です。通常、describe は一つの文字列を入れる為に、ここでは単純に Maxima の文字列を LISP の文字列に変換する函数として用いている事が判ります。

この事から topic を用いた条件分岐では、topic に束縛された値が、html や pdf という文字列であるかどうかを判別すれば良い事になります。

この判別には LISP の equal 函数が利用可能です。例えば、topic が文字列"html" に等しいかどうかは、`(equal mantype "html")` で判別出来る訳です。

さて、判別の方はこれで出来た訳です。今度は外部のアプリケーションの立ち上げの方法を調べましょう。

12.7 外部アプリケーションの立ち上げ方

さて、Maxima では外部アプリケーションはどの様に立ち上げているのでしょうか？グラフを Maxima に描かせる場合、デフォルトでは gnuplot が立ち上がっていますが、これはどの様にしているのでしょうか？gnuplot をキーワードにソースファイルを調べると、plot.lisp で用いられている事が判ります。

そして、plot2d 函数の定義を眺めていると、末尾に、この様な記述があります。

```
($mgnuplot
  ($system (concatenate 'string *maxima-plotdir* "/" $mgnuplot_command)
    (format nil " -plot2d \"%a\" -title '%a' file plot-name)))
```

これは MS-Windows で付属の mgnuplot の起動を行う箇所です。先ず、\$system 函数は Maxima から外部のアプリケーションを立ち上げる際に用いられる函数です。この函数に、Maxima 外部のアプリケーション等を実行させたい処理を Maxima の文字列として引き渡せば良いのです。ここで \$system 函数の引数を見ると、concatenate 函数が用いられている事が判ります。この concatenate は LISP の函数で第一引数が string の場合に後続の文字列を結合して、新しい文字列を生成する函数です。

この事から、外部アプリケーションは \$system 函数を用い、ブラウザと開くファイルの指定は concatenate 函数を用いてしまえば良い事になります。従って、\$hbrowser で \$jhtmldir で指定したディレクトリにある index.html ファイルを開きたければ、以下の様に記述すれば良い事になります。

```
($system (concatenate 'string $hbrowser " "
  $jhtmldir "index.html" "&")))
```

更に,HTML ファイルや PDF ファイルの名前も大域変数にしておくとうとうでしょうか?随分と汎用性が出ますね,そこで,ブラウザで開く HTML ファイルと PDF ファイルを指定する変数名を各々hfile と pfile とし,初期値を index.html と ManualBook.pdf にしておきましょう.

これを纏めると,関数 ponpoko は完成です.

12.8 完成

以上の事を組合せて関数 ponpoko を完成させてみましょう.

関数 ponpoko

```
(defvar $hbrowser "firefox")
(defvar $pbrowser "xpdf")
(defvar $hfile "index.html")
(defvar $pfile "ManualBook.pdf")
(defvar $jhtmlmdir "Maxima/ManualBook/")
(defvar $jpdfdir "Maxima/ManualBook/")
(defmspec $ponpoko (x)
  (let
    ((topic ($sconcat (cadr x)))
      (exact-p (or (null (caddr x)) (eq (caddr x) '$exact)))
      (cl-info::*prompt-prefix* *prompt-prefix*)
      (cl-info::*prompt-suffix* *prompt-suffix*))
    (cond
      ((equal topic "html")
        ($system (concatenate 'string $hbrowser " "
                              $jhtmlmdir $hfile "&")))
      ((equal topic "pdf")
        ($system (concatenate 'string $pbrowser " "
                              $jpdfdir $pfile "&")))
      (t (if exact-p
              (cl-info::info-exact topic)
              (cl-info::info topic))))))
```

説明は特に不必要かと思いますが,jhtmlmdir と jpdfdir は貴方の環境に合わせて修正して下

さい。私の環境では、HTML ファイルをホームディレクトリ上の Maxima/ManualBook 以下に置き、PDF ファイルも同じディレクトリに置いています。

ここで、この関数の遊び方ですが、兎に角、Maxima 側から load 関数を用いて読み込んでしまえば良いのです。ponpoko.lisp をカレントディレクトリ上に置いているのであれば、ディレクトリの指定をする必要も無く、単純に `load("ponpoko.lisp");` と入力して実行すれば良いのです。これで関数 ponpoko が読み込まれます。後は、`ponpoko("html");` や `ponpoko("pdf");` を実行すれば良いのです。さて実際の実行例を示しておきましょう。

```
(%i42) load("ponpoko.lisp");
(%o42)                                ponpoko.lisp
(%i43) ponpoko("to_lisp");

-- Function: to_lisp ()
      Enters the Lisp system under Maxima. '(to-maxima)' returns to
      Maxima.

(%o43)                                true
(%i44) ponpoko("html");
(%o44)                                0
```

読込んだ後に、to_lisp 関数を調べています。この場合は describe 関数と同じ働きをします。次に、引数を "html" にすると、デフォルトで firefox を立ち上げます。ここで、system 関数に送り込んだ文字列の末尾に & を入れている為、Maxima は firefox を立ち上げて入力プロンプトに復帰しています。ここで、& を末尾に入れていなければ、起動した外部アプリケーションを終了する迄、処理が Maxima に戻りません。

ここで、関数 ponpoko を一々読込むのが面倒であれば、maxima-init.mac ファイルに予め `load("ponpoko.lisp");` を記入しておきます。例として、lstringproc パッケージも読込む様にした maxima-init.mac を以下に示しておきます。

----- maxima-init.mac の例 -----

```
load("ponpoko.lisp");
load("stringproc.lisp");
```

このファイルは UNIX の場合、ホームディレクトリに置けば、準備が完了です。

12.8.1 MS-Windows 環境の場合

MS-Windows 環境の場合は UNIX 環境と比べ、幾つか注意しなければならない事があります。まず、フォルダ (ディレクトリ) の区切は MS-Windows では逆スラッシュ (\) を用いていますが、Maxima 内部では UNIX と同様にスラッシュ (/) を用います。ところで、考えておかなければならないのは、ponpok.lisp と maxima-init.mac、そしてマニュアルを置くフォルダです。

MS-Windows を利用する方は、基本的にコマンドプロンプト (以下、DOS 窓とも略記) を開いて処理する人よりも、wxMaxima 等の GUI 環境を用いるの方が大多数でしょう。その為、利用する環境によって maxima-init.mac 等の置場を変更する必要があります。

まず、何があんでも DOS 窓を利用する方の場合、maxima-init.mac と ponpoko.lisp の置場は Maxima による作業を行うフォルダに入れておく必要があります。

これに対し、wxMaxima や XMaxima を利用する場合は勝手が異なります。まず、メニューバーのアプリケーションから Maxima や XMaxima を選んで立ち上げる場合には、C:\Documents and Settings フォルダにある利用者固有のフォルダに maxima-init.mac と ponpoko.lisp を入れておく必要があります。ところが、wxMaxima はやや勝手が異なります。wxMaxima の場合、通常は wxMaxima が置かれた場所がそのホームディレクトリとなるので、ここに maxima-init.mac ファイルを置かなければなりません。

勿論、wxMaxima を DOS 窓で C:\Documents and Settings フォルダにある個人フォルダに移動して立ち上げてしまえば良いのですが、そうする場合には、wxMaxima や maxima の実行ファイルのあるフォルダを環境変数 Path に追加しなければなりません。この様に、Windows では何かと面倒な事が生じるのです。そこで、何も考えずに出来る方法は兎に角、個人フォルダと wxMaxima のフォルダの二箇所に別々に置いてしまう事です。wxMaxima だけを利用するのであれば、wxMaxima のフォルダだけで十分です。

さて、この wxMaxima は Maxima のフォルダの直下にあります。Maxima はインストール時に特に設定を行わなければ、C:\Program Files\Maxima-5.13.0 (利用する Maxima が 5.13.0 の場合) にある筈です。そして、maxima-init.mac、ponpoko.lisp や Maxima マニュアルのフォルダも一緒に置きます。ここで、Maxima マニュアルのフォルダは下に ManulaBook フォルダを置き、その中に HTML ファイル、PDF ファイルや利用する画像ファイルを置きます。

又、firefox とかのアプリケーションも MS-Windows の場合は Path が通っていない事があるので、別途、環境変数 Path にアプリケーションへの経路を設定しなければなりま

せん. この環境変数 Path の設定はコントロールパネルのシステムをダブルクリックし, その中にある環境変数のボタンを押します. ユーザー環境変数とシステム環境を編集する為のウィンドウが出て来るので, システム環境変数から, Path を選んで編集ボタンを押すと, 入力用のウィンドウ欄が出るので, そこに, firefox 等のアプリケーションが置かれたフォルダの位置を追加します. それで使える様になる筈です.

猶, UNIX 環境と MS-Windows 環境で切り替えを行う際に, Maxima では, Maxima の内部関数の *autoconf-win32* を用いて判別を行う様になっています. 実際, グラフ処理を行う為に, 環境によってアプリケーションの切り替えを行う必要がある plot.lisp では動作

する環境が MS-Windows 環境であるかどうかを, `(string= *autoconf-win32* "true")` を用いて判別しています. 実際, この S 式を評価して true になる場合を MS-Windows 環境, そうでなければ UNIX 環境として処理を行っています.

12.9 大域変数の変更について

ところで, ponpoko.lisp で定義した大域変数の変更は少し面倒です. 何故なら, Maxima の文字列と LISP の文字列は別物だからです. そこで, この変換に stringproc パッケージに付属する lstring 関数を用います. 例えば, hbrowser を初期値の firefox から opera に変更してみましょう.

```
(%i1) load("stringproc.lisp");
(%o1) /usr/local/share/maxima/5.13.0/share/contrib/stringproc/stringproc.lisp
(%i2) hbrowser;
(%o2) firefox
(%i3) hbrowser:lstring("opera");
(%o3) opera
(%i4) ponpoko("html");
(%o4) 0
```

この方法は ponpoko.lisp で定義した大域変数全てで使えます. 猶, 注意する事として, ディレクトリを指定する場合, 末尾に必ず "/" を入れ忘れないで下さい. これが無いと, ファイル名と変に繋って, 恐らく存在しないファイルを指す事になり, エラーが返される事になる筈です.

さて, 如何でしょうか? 残念ながら, まだブラウザを別個に立ち上げる手間が省けただけです. 今後, 必要な機能としては, 指定した項目が載っている HTML ファイルを検索してちゃんと開くといった操作が必要になりますね. これは私にとってはとても大

変な事で, 安易な方法を目指したこの章の範囲を越えてしまいそうです. そこで, 以降の作業は皆さんに投げっておきます.

第13章 結び目の Alexander 多項式

この章では三次元空間内の結び目について簡単に解説し、この結び目から得られる結び目群の表現、最後にこの結び目群の不変量としての Alexander 多項式について簡単な解説をしています。

猶、この章の核心は Maxima に於ける演算子の定義方法と規則の適用です。従って、結び目に関して興味が無い場合は、演算子の定義と規則の適用の事例として参考にして下さい。

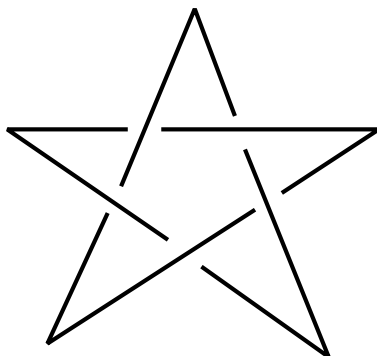
13.1 結び目の概要

始めに結び目理論から結び目の Alexander 多項式の計算について簡単なお話をします。ここで何故、結び目なのかという理由ですが、結び目理論はドイツ語では Knotten Theorie と呼びます。それにしても、Knotten!! 実に、KNOPPIX に似ていますねえ…。そこで、結び目愛好家の為に、Maxima で結び目の不変量を計算して、KNOPPIX/Math を Knotten Pics/Math と洒落込もうというのが目的です。

結び目理論の全般の話はクロウエル、フォックス [28]、本間 [33]、河内 [13]、村上 [36] 等を参照して下さい。猶、亀甲結びの結び方と言った別の方面の愛好家の方の要望には沿える内容ではありませんので、悪からず。

結び目には蝶々結びとか、色々な紐の結び方があります。ここで、結び目のある二つの紐が与えられた時、その結び目が同じ結び方で出来ているのかどうかを判別するにはどうすれば良いのでしょうか？現実問題としては、紐を引いたりして同じ形に変形出来るかどうかで判別する方法がありますが、数学では、どの様に表現すべきでしょうか？まず、結び目理論で扱う結び目は、紐の両端を繋いで輪にしています。丁度、図 13.1 に示す様に 3 次元空間内部の円周になっています。

こうするとどうでしょう、結び目の両端が切れたままだと、結び目の瘤の部分の引いて行けば結び目が消えてしまいますね。例えば、「トムとジェリー」で子鼠がテーブルから下したパスタを一気に吸引してテーブルに登るシーンの様な感じで結び目が消えて

図 13.1: 星型結び目 (5_1)

しまうのです. ところが, 両端を繋げて輪にしてしまうと, 端が無いので, そこから潰す事も出来ず. 引っ張っても結び目が解けて消えるとは限らなくなります.

この結び目は自分自身が交わる事の無い様に 3 次元空間 $\mathbb{R}^3$ に置かれています. この状態を円周 S^1 の 3 次元空間 $\mathbb{R}^3$ への埋め込みと呼びます. 更に, $\mathbb{R}^3$ から結び目を取り除いた空間 $\mathbb{R}^3 - K$ の事を結び目 K の補空間と呼び, $C(K)$ と記述します.

次に, 扱う結び目に現実的な制約を入れましょう. 先ず, ここで扱う結び目は, 有限個の折線で近似可能な結び目に限定しましょう. この有限個の折線で近似出来る結び目の事を順 (tame) な結び目と呼びます. 因に, 無限個の折線が必要な結び目を野性的 (wild) な結び目と呼びます. この様な野性的な結び目は, 力学系, 特にカオスの話に出て来る事がありますが, 基本的に, 帰納的な構成方法で得られるものです. . 次に, 結び目をゴム紐で出来たものの様に考えて, 結び目を $\mathbb{R}^3$ 内部で, 紐を切ったり, 紐を交点させずに変形して行く事で, 互いに移り合う事の出来る結び目を同値な結び目と呼びましょう. この事を数学では, 結び目 K_1 と K_2 に対し, ambient isotopy と呼ばれ, 図形が相互に連続的に変形して移り合える可能性を保証する写像が存在する事を意味します. そして, ここでは結び目 K_1 と K_2 の間に ambient isotopy が存在する事を $K_1 \Leftrightarrow K_2$ と表記しましょう. この ambient isotopy という写像が結び目 K_1 と K_2 の間に存在する事を簡単に言うのであれば, $t = 0$ [分] の時が, 結び目 K_1 で, それから結び目の紐が自分自身と交点する事もなく, 結び目の形を徐々に変形させながら, $t = 1$ [分] の時に結び目 K_2 に移って行く映画が作られる事を意味します.,

さて, 個々の結び目を特徴付けるものに何があるのでしょうか? 次の節では, 結び目の射影図を考えますが, そこで現れる結び目の交点の総数もその一つです, 勿論, 射影図の取り方で交点の総数は色々変化します. 例えば, 平面 $\mathbb{R}^2$ に置かれた $x^2 + y^2 - 1 = 0$

で定義される円周の事を自明な結び目と呼びますが, この自明な結び目を Y 軸を中心に捻じったものを XY 平面に射影すれば, 交点は幾らでも出来ます. しかし, 結び目の射影図が持ち得る最小の交点数を考えると如何でしょうか. 自明な結び目の場合は 0 個になります. 自明な結び目以外の一般の結び目の場合, 交点数の下限は 0 個になるので, 交点数が必ず最小限を持つ事が選択公理によって保証されます. そこで, 最小の交点数で結び目を分類する方法が考えられますね.

実際どうなるのでしょうか? 先ず, 交点が無い結び目は自明な結び目の他には存在しません. この場合は一つだけです. 何か使えそうですね. 実際, この交点で分類したものが, 数学辞典 [37] の付録, 公式 7 の結び糸等の文献にある結び目の表です. 表を見て頂くと判りますが, 交点が 3 個と 4 個の場合は一つだけですが, 交点数が 5 個以上になると同じ交点数を持つ結び目が複数個存在します. その為, 結び目を交点数でグループ分けは出来ませんが, 個々の結び目を特徴付けるには十分ではありません. 残念でした. では, どうすればよいでしょうか. 天下りの的になりますが, 結び目には結び目群というものがあり, この結び目群で結び目は特徴付けられます. この結び目群は結び目 K の補空間 $C(K)$ の基本群と呼ばれる群の一種の事です. 既に 3 次元空間内部の結び目はその補空間で分類が可能な事が知られています.

ここで, 結び目群の計算方法は, Dehn による Dehn 表示と, Wirtinger による Wirtinger 表示の二つの計算方法が代表的ですが, どちらも結び目の射影図を描いて求める方法です. ここでは, より機械的に計算が出来る Wirtinger による結び目群の表示を利用します.

この結び目群は非常に情報を沢山含んでいますが, そのままでは非常に扱い難い代物です. 実際, 群も自由群に関係子を入れたものになるので, 二つの群が与えられた時に, それらの群が同じものかどうか判断するのはそんなに簡単ではない事が殆どです. 一応, Tietze 変換と呼ばれる操作で移り合える群は同じものである事が知られていますが, それでも結構大変です. そこで, もっと簡単に結び目を比較出来る方法があります. それが, 結び目群の表示の関係子から得られる Alexander 多項式を計算する方法です. この方法では, 結び目に対してその Alexander 多項式を計算し, Alexander 多項式を使って結び目を比較します. 猶, 結び目群の表示から Alexander 多項式を求める時に, Fox の微分子という一風変わった演算子を用います.

この章では, Maxima の規則について詳細を述べる為に, 結び目群から Alexander 多項式を生成する方法について簡単に述べます. その為, ここで紹介するプログラムは計算効率があまり良いものではない事 (特に Alexander 多項式を計算する個所) を予め断っておきます.

13.2 結び目の射影図

最初に結び目の射影図というものを描きます. これは 3 次元空間 $\mathbb{R}^3$ 内部の結び目を平面 $\mathbb{R}^2$ に射影したもので, 図 13.2 に示す要領で, 光源 O (無限遠点でも構いません) から出る灯で結び目を射影したものです.

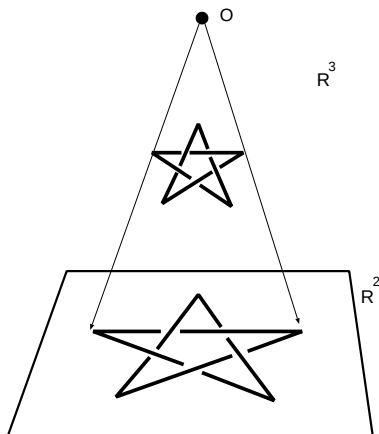


図 13.2: 結び目の射影図

ここで, 射影図上には結び目の紐自体が交点する個所が現われる事があります. この交点する個所を交点と呼びますが, この交点を十字で描くだけでは, どちらが上にあるか判らないので, 地図で高速道路のジャンクションを描く要領で, 上の道で下の道が切断される様に描きます. 即ち, 図 13.1 の様な絵を描きます. 猶, 交差点で上を通る道の事を上道, その下側の道を下道と呼びます.

ところが, 図 13.1 の様に綺麗な射影図が何時も描けるとは限りません. 図 13.3 の左側に示す様に, $n(\geq 2)$ 重点や直交しない交点が出来事もあります. ところが, 図 13.3 の様な交点は, 順な結び目には高々有限個しか出て来ません. 何故なら, 順な結び目は有限個の折線で近似可能なもので, その射影も有限個の折線で近似出来るからです. さて, それらの性格の悪い交点達は, 図 13.3 の右側に示す要領で紐を局所的に動かしてしまえば除去出来てしまいます. この処理を次々と射影図に施して行くと, 最終的に射影図で現われる交点を二重点だけに出来る事があります. この様にして, 有限個の交点で, それらが二重点のみとなった射影図の事を正則な射影図と呼びます.

ところで, 同じ結び目でも射影の方向の違いで形が違ふものに写る事があります. そこで, 射影図に対しては, 図 13.4 に示す Reidemeister 移動と呼ばれる変形で結び目の

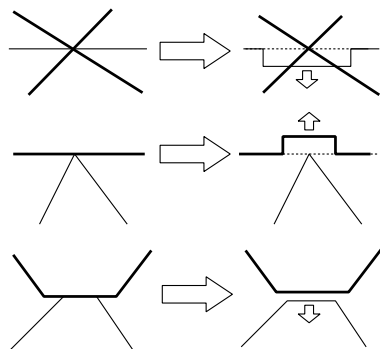


図 13.3: 変形可能な交点

射影図を変形する事も可能です.

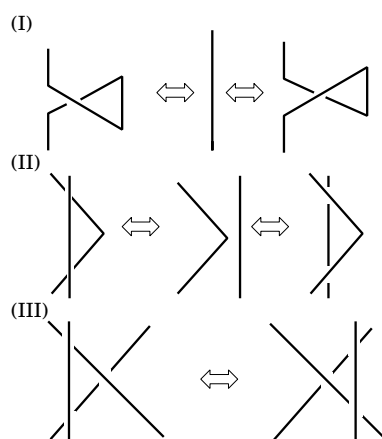


図 13.4: Reidemeister 変形

ここで, 同値な結び目は Reidemeister 移動を繰替えして行う事で, 相互に変形可能となる事が知られています.

13.3 結び目群の Wirtinger 表示

この節で, いよいよ結び目 K の結び目群 $G(K)$ の Wirtinger 表示を計算しましょう.

先ず, 綺麗な射影図が出来上がった状態で結び目の射影図は交点で幾つかの上道で分割されていますね. では, 次に結び目に向きを入れて, それから各上道に適当な変数を割当てましょう.

図 13.5 では星型結び目 5_1 の射影図の各曲線に変数 v, w, x, y, z を割当てた様子を示しています.

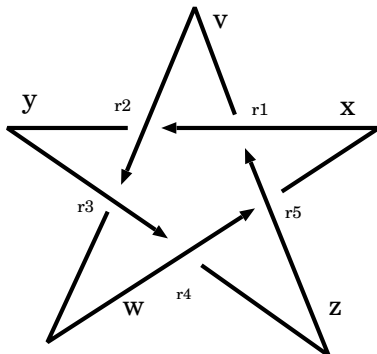


図 13.5: 星型結び目 5_1 と変数

先ず, Wirtinger 表示による結び目群では, 射影図上の曲線に割当てた変数が群の生成元となります. 次に, 結び目群の関係子は各交点で図 13.6 に示す方法で交点毎に決定されます.

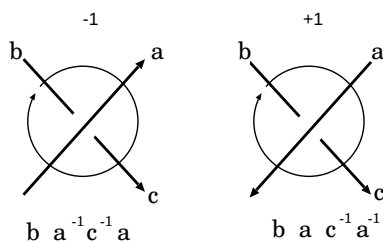


図 13.6: Wirtinger 表示

因に, 交点の上の $+1$ と -1 は交点の符号と呼ばれるものです. 結び目の交点の符号の総和は符号和と呼ばれるもので, これも重要な結び目の不変量の一つです.

以上から, Wirtinger による結び目群の表示は次の形になります.

$$\langle \text{上道}_1, \dots, \text{上道}_n \mid \text{交点}_1, \dots, \text{交点}_n \rangle$$

では何故, Wirtinger 表示で結び目群が表現出来たのでしょうか? この節も非常に天下りの的になりますが, その事を簡単に説明しましょう.

まず, 結び目の射影図の上道に付けた変数は図 13.7 に示す様に, $\mathbb{R}^3 - K$ 内部の点 P から出て, 結び目の上道の向きに対して右回りで上道を一回りし, 点 P に戻る閉じた道が対応します. この閉じた道は点 P に根本が固定されていますが, ゴム紐の様に伸縮自在で, 結び目を取り除いて出来た穴に邪魔されなければ, 点 P に潰れてしまう性質を持っています. ここで, 点 P を基点とする閉じた道 a_1 と a_2 が等しくなるのは, a_1 と a_2 が空間内部で連続的に変形する事で互いに移り合える場合とします.

閉じた道 a b による積 ab は点 P を出て a の道を辿って, 次に b の道を辿る閉じた道と定義します. この時, 点 P から動かない道は, この積の単位元となります. 更に, 生成元 a を閉じた道とすれば, a^{-1} を逆の同じ閉じた道で, 逆方向の左回りに上道を一周する閉じた道としましょう.

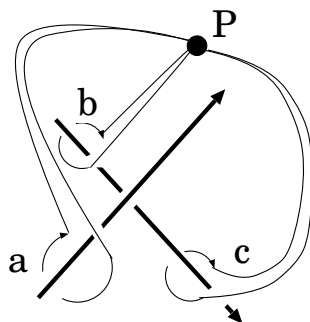


図 13.7: 結び目と閉じた道

ここで, a と a^{-1} の意味を説明するのが図 13.8 です.

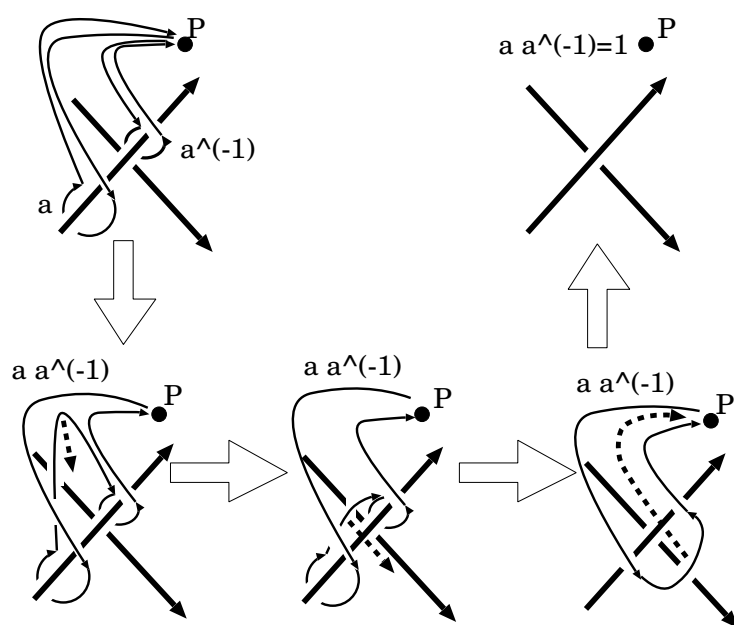


図 13.8: 閉じた道 a , a^{-1} と aa^{-1} の意味

この様に、通常の閉じた道 a と a^{-1} は各々結び目の上道に遮られているので、この上道を越えて点 P へと潰れる事は出来ません。

ところが、積 aa^{-1} は、一度、閉じた道 a を一回りしてから、次は a^{-1} の道、つまり、 a の道の逆方向に動く事を意味します。そこで、 aa^{-1} は図 13.8 の中段左の様な道になります。この道を矢印に沿って動かすと、結局、この閉道は結び目から外れてしまうので、点 P に潰れる事になります。すると、点 P から動かない点が単位元 1 となります。この事から、 a^{-1} の意味が明確になるかと思えます。

Wirtinger 表示の関係子の意味は、この図 13.8 でやっている事と似た処理を行う事で判ります。先ず、図 13.9 に示す様に、これらの閉じた道を関係子の順で繋ぐと、曲線から外れた点 P から出て点 P に戻る向き付けられた円になります。この円を少しずつ動かして行くと、結び目の射影図の曲線に引掛かからずに点 P に潰す事が出来ます。この様に、Wirtinger 表示の関係子は、関係子で表現される式が単位元 1 になる関係がある事を意味しているのです。

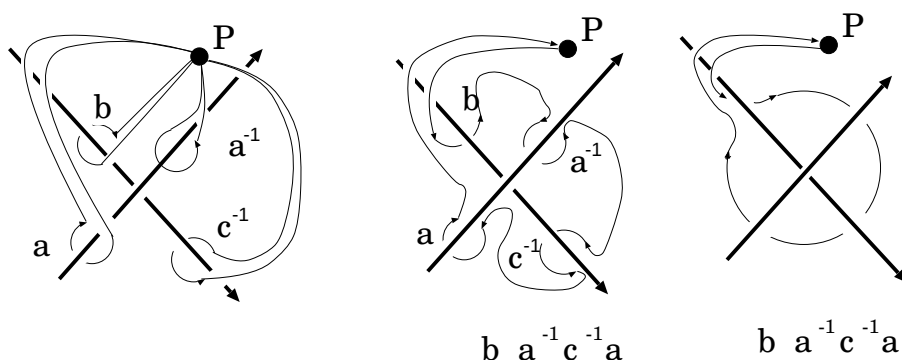


図 13.9: Wirtinger 表記の意味

この様に閉じた道が 1 になるという事は、閉じた道が空間上で邪魔される事無しに基準点 P に潰れる事を意味しましたが、同時に $\mathbb{R}^3 - K$ 内部 (結び目の補空間と呼びます) に埋め込まれた円盤、即ち、自分自身が交点した変な円盤ではなく、歪んでいるだけの円盤の境界になっている事も知られています (Dehn の補題)。

因に、3 次元空間内部の球面が平面上の円と同様に 1 点に潰れる場合、球面が球の境界になっているかどうかという問題があります。これは Poincare 予想) と同値な問題です。何故なら、もし、球面を境界に持つ、3 次元の球と同じ (同相) でない単連結な球の様なもの (偽物の球と呼びましょう) が存在すると、本物の球と偽物の球面を境界の球面で貼り合わせる事で偽物の 3 次元球面が出来るからです。

又, 自明な結び目の場合, 結び目の補空間はドーナツの様な空間になる為, その結び目群は $\langle x \rangle$, 即ち $\mathbb{Z}$ に等しい事が判ります.

猶, 結び目群は正確には結び目の補空間の基本群と呼ばれるものです. 一般的に, この基本群が 1, 即ち, 任意の閉道が潰れる空間の事を単連結な空間と呼びます. 近年証明された Poincaré 予想は「単連結な閉 3 次元多様体は 3 次元球と同相である」というものです. ここでいきなり多様体という言葉が出ましたが, これは物理学的な意味での宇宙空間と理解されていても問題は無いでしょう.

要するに, 3 次元多様体の場合は, 座標系が設定されたボールの集合として空間が表現され, ボールが重なり合う個所では, 座標の変換が上手く行く事が保証されている空間です. 物理学では様々な観測から考察する訳ですが, 観測結果が別の地点で一定の規則で変換出来てくれないと困りますね. 多様体はその様な事を保証する空間なのです. ここで, この変換規則が微分可能であれば, 可微分多様体, 連続写像であれば位相多様体と呼びます. ここで, 結び目は有限個の折線で近似可能なものに限定していますが, この仮定から, 結び目の補空間は PL 多様体と呼ばれるものになります. この PL 多様体と可微分多様体は同値なものである事が知られています.

さて, Wirtinger 表示の話に戻ります. 図 13.5 の星型結び目 5_1 の場合, 関係子の計算は図 13.7 の関係式を参照すれば, 結び目群の Wirtinger 表現として以下が得られます

星型結び目 5_1 の Wirtinger 表現

$$\langle v, w, x, y, z | xz^{-1}x^{-1}v, vx^{-1}v^{-1}y, yv^{-1}y^{-1}w, wy^{-1}w^{-1}z, zw^{-1}z^{-1}x \rangle$$

猶, この群では関係子が交点の数だけ出来ますが, 実際はこの関係子の内で一つは不要です. そこで一つを取り除いたものでも構いません.

Wirtinger 表示による結び目群は図 13.4 に示す Reidemeister 移動の不変量となります. この事を説明する為には語の Tietze 変換を用います. 詳細は クロウエル, フォックス [28] 等の結び目理論の本を参照して下さい.

13.4 群環と Fox の微分作用素

次に, 結び目群 $G(K)$ の生成元を $x_1, \dots, x_n$ とする時, これらで生成される自由群 $F = \langle x_1, \dots, x_n \rangle$ とその群環 $\mathbb{Z}F_n$ を考えましょう. 群環 $\mathbb{Z}F_n$ は群の成分を可換演算子 $+$ を使って形式的な和を取ったものになります. この群環は, 積 $*$ に対しては群になり, 和 $+$ を結合律や分配律を満す様に入れるので環になります.

Fox の微分子 Δ は次の性質を持っています.

————— Fox の微分子 Δ —————

- $\Delta(x + y) = \Delta(x) + \Delta(y)$
- $\Delta(xy) = t(y)\Delta(x) + x\Delta(y)$

先ず, Fox の微分子 Δ は $\mathbb{Z}F_n \rightarrow \mathbb{Z}F_n$ の線形写像 ($\mathbb{Z}$ 準同形写像) となります. それから, 語の積に対しては, 何処となく普通の微分の積公式に似たものですが, 函数 $t: \mathbb{Z}F_n \rightarrow \mathbb{Z}$ で, 群 F_n の各生成元に 1 を代入する線形函数 t が入っている事に注意して下さい. この基本性質を利用すると, Fox の微分子が次の性質を持つ事が容易に確認出来ます.

————— Fox の微分子 Δ の性質 —————

- $\Delta(n) = 0, n \in \mathbb{Z}$
- $\Delta(x^n) = \sum_{i=0}^{n-1} x^i \Delta(x), n \in \mathbb{N}$
- $\Delta(x^{-n}) = -\sum_{i=-n}^{-1} x^i \Delta(x), n \in \mathbb{N}$

では, Maxima で Fox の微分子を実際に表現してみましょう.

13.5 Maxima で遊ぶ Fox の微分子

Fox の微分子を Maxima で表現する為には, Fox の微分子で利用する各種の函数を最初に定義しなければなりません.

先ず, Fox 微分の積の性質で出て来た函数 t を定義しなければなりません. 但し, t は変数で使う事が多く, 紛らわしいので, ここでは函数名を $t1$ としましょう.

この函数は線形函数なので, 語に対してのみ函数を定義し, 後は declare 函数を使って線形性を持つ事を宣言してしまえば十分です. そこで, 以下の Maxima の函数として定義します.

————— 函数 $t1$ —————

```
t1(x):=block([vars:listofvars(x),n,i],
              n:length(vars),
              for i in vars do
                (x:subst(1,i,x)),
              return(x));
```

この函数では, 全ての変数に 1 を代入する為, listofvars 函数で語を構成する変数を全て取出し, subst 函数を使って変数に 1 を代入します. 実際は引数が語であれば 1 となる様に定義し, それから, declare 函数で函数 t1 が linear であると宣言しておけば十分ですが, ここでは subst 函数と listofvars 函数を使いたかったので, 敢て複雑にしています.

今度はこの函数 t1 の値域となる群環 $\mathbb{Z}F_n$ の Maxima での表現方法です. ここでは, 群 F_n の積は Maxima の非可換積である dot 積を用い, 冪はそれに対応する非可換積の冪^^を使います. 但し, 整数と語の積では, 可換積 * を用います.

従って, 非可換積と可換積が混入した式を扱う事になりますが, この様な式の計算を Maxima は難無く実行します.

```
(%i76) 2*x.y+3*x.y;
```

```
(%o76)                                5 (x . y)
```

```
(%i77) 2*x.y.z+3*x.y;
```

```
(%o77)                2 (x . y . z) + 3 (x . y)
```

これで群環の演算は一応大丈夫そうですね.

では, 与式が語であるかどうかを判別する真理函数を定義しましょう. 与えられた式を語として判断しても良いのは以下の 1 から 4 に限られます.

—— 与式が語となる条件 ——

1. 与式が原子 w
2. 与式が原子の積 $w_1.w_2 \cdots w_n$
3. 与式が整数と原子の可換積を含む式 $n * w_1.w_2 \cdots w_n$
4. 与式が原子や語の非可換冪 $x^{\wedge m}$

これらの状況の判定は, 式に含まれる演算子で出来てしまいます. 先ず, 最初の原子 w の場合は演算子が含まれていませんが, 真理函数 atom で判別出来ます. それから与式が $n * w_1.w_2 \cdots w_n$, $w_1.w_2 \cdots w_n$ と $x^{\wedge m}$ の場合, 各々の内部表現の先頭は, 可換積 *, 非可換積. や非可換積の冪^^の何れかになります.

以上から, 以下の判定条件で式を判別すれば良い事になります.

— w が語となる判定条件 —

1. `atom(w)` が true
2. `inpart(w,0)` が可換積*
3. `inpart(w,0)` が非可換積.
4. `inpart(w,0)` が非可換積の冪^^

定義すべき真理関数 `wordp` は, 最初に与式が原子であるかどうかを調べ, 原子であれば true, そうでなければ, 与式の内部表現の第一成分を取出して, それが非可換積積, 可換積, 冪の何れかであれば, true を返し, それ以外は false を返す関数になります.

この関数を Maxima で構築したのが次に示す真理関数 `wordp` となります.

— 真理関数 wordp —

```
wordp(w) := if atom(w) then true
            else if member(inpart(w,0),[".", "^", "*"]) then true
            else false;
```

この `word` 関数では, `inpart` 関数を用いて, `inpart(w,0)` で与式の内部表現の第一成分を取出し, `member` 関数を使って, その式の第一成分の値が, 非可換積, 非可換積の冪 ^^ か可換積* の何れかであれば true を返す仕様になっています.

では早速, この関数を試してみましょう.

```
(%i59) wordp(w);
(%o59)                                     true
(%i60) wordp(2*w.z);
(%o60)                                     true
(%i61) wordp(w.z);
(%o61)                                     true
(%i62) wordp(w.z+2*x.y);
(%o62)                                     false
```

これで, 与式が語であるかどうかを検証出来る真理関数が出来ましたね. 利用する前に `declare` 関数を使って, `declare(wordp, linear)` とすれば十分です. この真理関数の存在によって, Maxima にとって何が語となるかが判断出来る様になりました. この様な規則を定める対象を判別する真偽関数が出来た段階で, 初めて規則が定義可能となりました.

では, Fox の微分子 `D_fox:` を Maxima で表現しましょう. ここでは最初に演算子の属性を宣言しておきます. この関数は, 引数の前に置くので, `prefix` 関数を用います. 定義するだけであれば, `prefix("D_fox:")` だけでも十分です. 猶, Maxima の演算子には, 演算子と被演算子を結び付ける力が定義可能です. この演算子と被演算子を結び付ける力の事を演算子の束縛力と呼びます. 例えば, 式 $1 + 2 \cdot 3^4 - 4$ が与えられた時, 和, 積, 差と冪等の演算には順番があるので, この式は $1 + (2 \cdot (3^4)) - 4$ と解釈されますね. この様に演算子の隣の被演算子を引き付けておく力を Maxima では束縛力と呼んで, その強さを数値で表現しています. Maxima では和は 100, 可換積の左束縛力が 120, 非可換積の左束縛力が 130 で右束縛力が 129, 非可換積の冪の左束縛力が 140 で右束縛力が 139 となります. Maxima の関数を用いて演算子を定義した場合, 束縛力は初期値で 180 が設定されます. その為, `D_fox:x^5` が与えられた時, 演算子 `D_fox:` は非可換冪の束縛力よりも大きい為に, `(D_fox:x)^5` と Maxima は解釈します. そこで, `x^5` に `D_fox:` を作用させたいければ, `D_fox:(x^5)` の様に小括弧 `()` で括ります. 但し, 安全性を考えると, `D_fox:` の束縛力は `^` よりも小さい方が無難です. そこで, 右束縛力を 128 としましょう.

猶, ここでの宣言では, 具体的に関数を記述する必要は全くありません. Maxima は演算子に与えられた様々な性質や規則に基いて処理を行います. この宣言に続けて, 今度は線形性 $D(x+y) = D(x) + D(y)$ も追加しましょう. この線形性の宣言に `declare` 関数を用います. では以下の様に, 演算子 `D_fox:` の宣言を行いましょう.

```
(%i1) prefix("D_fox:",128);
(%o1)                                     D_fox:
(%i2) declare("D_fox:",linear);
(%o2)                                     done
```

これで, `D_fox` の演算子としての属性と線形性が付加されました.

次に, 微分の性質を付加します. この性質は Maxima の規則として与えます. その為に, 並び変数の宣言を予め行い, 規則の宣言では, この宣言しておいた並びの変数を用います.

```
(%i3) matchdeclare([_x,_y],wordp);
(%o3)                                     done
(%i4) defrule(Dfox_Prod,D_fox:(_x._y),
              D_fox:_x*t1(_y)+_x.D_fox:_y);
(%o4) Dfox_Prod : D_fox: (_x . _y) ->
              D_fox:_x t1(_y)+_x.D_fox:_y
```

この例では、最初に `matchdeclare` 函数を使って、規則を与える際に使用する並びの変数の定義を行っています。Maxima の規則は `matchdeclare` 函数で宣言した変数を用いて定義を行いますが、一般の式への適用では、後の真理函数を用いて、その式に適用する際に前提とした変数の条件を満たすものかどうかを Maxima は検証します。この宣言を行わなければ、Maxima は `defrule` 函数で用いた変数に対してのみ規則の適用を行い、一般の式に対して利用は出来なくなります。猶、`matchdeclare` 函数では並びの変数を判定する真理函数を設定しますが、ここで `true` を設定すると、全ての変数に対して `true` が返される事になるので、条件を付ける必要が無い場合は `true` を真理函数として指定する事が可能です。

次に、`defrule` 函数を用いて、規則名と、適用する式の並びと規則の適用後の変換式を宣言します。この例では、規則名を `Dfox_Prod` とし、その内容は `D_fox:(_x . _y)` を `D_fox: x*t1(_y)+x.D_fox:_y` で置換するものです。こうする事で、変数 `a` と `b` が語としての属性を持っていれば、`D_fox:(a.b)` は規則 `Dfox_Prod` を適用する事で `D_fox:a * t1(b) + a . D_fox:b` に置換される事になります。

以上で非可換積に対する微分の性質が付加されました。

ここで注意しなければならない事は、非可換積に影響を与える大域変数 `dotassoc` をデフォルトの `true` のままにしていると、Maxima が `(x.y).z` を自動的に `x.y.z` に平坦化してしまう事です。これは非可換積。自体は infix 型の中置演算子ですが、非可換積の規則によって、nary 型の演算の様に、自動的に内部表現が `(. x y z)` で置換えられます。ここでの規則は二項演算を前提にしているので、`(x.y).z` の様に平坦化される前の二項演算のままであれば処理が上手く出来ません。この処理を行わせない様に、結び目群を定義する前に `dotassoc:false` を設定していなければなりません。

最後に、与式が定数や非可換冪の場合の処理を組み込みましょう。先ず、与式が定数の場合、Fox の微分子は 0 を返します。ここで環は整数環 $\mathbb{Z}$ なので、定数は `integer` として仮定して構いません。従って、定数に対しては、次の規則を入れれば十分です。

```
(%i5) matchdeclare(_a,integerp);
(%o5)                                     done
(%i6) defrule(Dfox_const,D_fox:_a,0);
(%o6)                                     Dfox_const : D_fox: _a -> 0
```

次に、与式が非可換冪の場合を組み込みましょう。この場合、`matchdeclare` で宣言する整数には正の場合と負の場合を各々宣言する必要があります。そこで、`posintp` と `negintp` の二つの真偽函数を定義します。

posintp 関数と negintp 関数

```
posintp(x) := if featurep(x,integer) then if is(x>0) then true
negintp(x) := if featurep(x,integer) then if is(x<0) then true
```

posintp 関数と negintp 関数は基本的に同じ操作を行う関数です. 与えられた引数が整数であれば, 既に整数の属性を持っているので, featurep 関数は true を返し, 後は正負の判定を行えば良く, 一般の変数の場合, 先ず, declare 関数で整数の属性を持たせ, それから assume 関数で正負の性質を入れてあるものを扱います.

では, 関数の動作をここでも確認しておきましょう.

```
(%i11) declare(n1,integer);
(%o11)                                     done
(%i12) assume(n1>0);
(%o12)                                     [n1 > 0]
(%i13) posintp(n1);
(%o13)                                     true
(%i14) declare(n2,integer);
(%o14)                                     done
(%i15) assume(n2<0);
(%o15)                                     [n2 < 0]
(%i16) negintp(n2);
(%o16)                                     true
(%i17) negintp(n1);
(%o17)                                     false
```

これで道具が揃ったので, これらの真偽関数を用いて正の冪の場合と負の冪の場合の展開規則を D_fox: に付与しましょう. 規則の与え方は, 微分としての展開規則 Dfox_Prod と同様です. 最初に規則を与える変数を matchdeclare 関数を用いて宣言し, その変数を用いて defrule 関数で, 規則を宣言します.

```
(%i9) matchdeclare(_ap, posintp)$
(%i10) matchdeclare(_an, negintp)$
(%i11) defrule(Dfox_PPower, D_fox: (_x^(_ap)),
              sum(_x^(i), i, 0, _ap-1).D_fox: _x);
(%o11) Dfox_PPower : D_fox: (_x      )
              -> sum(_x      , i, 0, _ap - 1) . D_fox: _x
```



```
(%i12) defrule(Dfox_NPower, D_fox:(_x^(_an)),
               -sum(_x^(i), i, _an, -1).D_fox:_x);
(%o12) Dfox_NPower : D_fox: (_x
               -> - sum(_x      , i, _an, - 1) . D_fox: _x
```

これで, Fox の微分子を利用する為に必要な規則は全て揃いました. では試しに $r1 = x \cdot y \cdot z^{-1} \cdot y^{-1}$ に Fox の微分子を作用させた結果を計算してみましょう.

```
(%i9) dfr1:D_fox:(r1);
               <- 1>      <- 1>
(%o9)          D_fox: (x . (y . (z      . y      )))
```

入力しても別に展開されていません. 何故でしょう? Maxima では規則を定めるだけではなく, その定めた規則を式に対して適用する函数を必要とします. この様な函数には幾つかありますが, 今回は apply1 函数を用います.

必要な規則は, 語の非可換積の展開規則 Dfox_Prod と負の冪の規則 Dfox_NPower の二つです. この様に複数の規則を利用する場合, apply1 に適用する規則を順番に並べます. ここでは最初に展開規則を用い, それから負の冪の展開規則を用いるので, 以下の様に入力します.

```
(%i11) apply1(dfr1,Dfox_Prod,Dfox_NPower);
               <- 1>          <- 1>      <- 1>
(%o11) x . (y . (- z      . D_fox: z - z      . (y      . D_fox: y))
               + D_fox: y) + D_fox: x
```

確かに展開が出来ていますね.

以上の Fox の微分子の定義, 規則の宣言等を纏め, ファイル fox.mc に記入しておきましょう. このファイルの中に注釈を入れておくと後で読み易く, 管理を行うのも容易になります. Maxima では注釈は `/*` と `*/` の間に記述します. このファイル fox.mc の内容を以下に示しておきましょう.

```
/* MAXIMA */

/* 函数 t1

   x: 結び目群の語.
```

Fox 微分で現われる函数. 線形性を持ち, 語を 1 に写す函数.
 函数の定義では, 単純に語の変数に 1 を subst を用いて設定し,
 線形性は declare を利用します.

*/

```
t1(x):=block([vars:listofvars(x),n,i],
             n:length(vars),
             for i in vars do
               (x:subst(1,i,x)),
             return(x));
```

```
declare(t1,linear);
```

/* Fox の微分子の定義.

演算子は prefix 函数を用いて, 前置式の演算子である事を宣言します.
 猶, 被演算子の右束縛力は 128 とし, 非可換冪よりも小さな値とします.
 更に, 線形性は declare 函数を用います.

*/

```
prefix("D_fox:",128);
declare("D_fox:",linear);
```

/* Fox の微分子の置換規則で用いる変数宣言と真偽函数定義 */

/* _x と _y は語とします. */

```
matchdeclare([_x,_y],wordp);
```

/* 語の判定函数 */

```
wordp(w) :=
if atom(w) then true
else if member(inpart(w,0),[".", "^", "*"]) then true
else false;
```

```

/* _a は整数とします. 判定函数は Maxima 組込の integerp を用います. */
matchdeclare(_a,integerp);

/* _ap は正整数,_an は負の整数とします. ここで判定函数は,
正整数の場合は posintp, 負の整数の場合は negintp を用います.*/
matchdeclare(_ap,posintp);
matchdeclare(_an,negintp);

/* 語の判定函数 */
posintp(x):= if featurep(x,integer) then
              if is(x>0) then true;
negintp(x):= if featurep(x,integer) then
              if is(x<0) then true;

/* 積規則 Dfox_Prod */

defrule(Dfox_Prod,D_fox:(_x._y),D_fox:_x*t1(_y)+_x.D_fox:_y);

/* 定数に対する規則 Dfox_const */
defrule(Dfox_const,D_fox:_a,0);

/* 正の冪に対する規則 Dfox_PPower */
defrule(Dfox_PPower, D_fox:(_x^^(_ap)),
        sum(_x^^(i),i,0,_ap-1).D_fox:_x);

/* 負の冪に対する規則 Dfox_NPower */
defrule(Dfox_NPower, D_fox:(_x^^(_an)),
        -sum(_x^^(i),i,_an,-1).D_fox:_x);

/* 非可換積の結合律の適用を阻止. 積規則を利用する為に必要 */
dotassoc:false;

```

このファイルの演算子, 函数や規則を用いたければ, load 函数を用いて `load("fox.mc");` と入力します. 更に, Maxima の起動時に直ちに使いたければ, Maxima を起動するディレ

クトリに maxima-init.mac という名前のファイルを作り, その中に, 予め, `load("fox.mc");` を記入しておきます. すると, maxima-init.mac があるディレクトリ上で maxima を起動させると, 自動的に maxima-init.mac 内部が解釈されて fox.mc が読込まれます. これで結び目の Alexander 多項式を計算する為の道具が漸く揃いました.

13.6 Alexander 多項式

Alexander 多項式を計算する為には, Alexander 行列というものを計算しなければなりません. この Alexander 行列は結び目群 $G(K)$ の関係子に Fox の微分子を作用させる事で出来る行列です. 即ち, ベクトル $(r_1, \dots, r_n)$ から Fox の微分子による Jacobian が Alexander 行列になります.

ここで, 先程の節で定義した Fox の微分子は, 通常の微分と言えば, D に相当します. 普通, 変数 x に対する微分 $\frac{d}{dx}$ が存在しても良さそうですが, Fox の微分子も同様で

す. この場合, 結び目群の Wirtinger 表現の生成元 x_i に対して, $\frac{\partial}{\partial x_i}$ が定義され, 他の生成元 x_j に対して, $\frac{\partial x_j}{\partial x_i} = \delta_{ij}$ を満たします.

例えば, 星型結び目の場合, 生成元は x, y, z, v, w の 5 個があるので, Fox の微分子は具体的には, $\frac{\partial}{\partial x}, \dots, \frac{\partial}{\partial w}$ と生成元の数だけ存在します.

ここで Maxima 上で定義した Fox の微分子 D.fox:には, 生成元の情報がありません. 実はこうした方が便利なのです. 例えば, 星型の場合, 5 個の微分子を定義し, それらを使って行列を求めるたりする事や, 2 変数の関数を定義して処理を行うよりも, ここで行った様に, 一つの演算子を決めて, $\frac{\partial}{\partial x_j} x_i = \delta_{i,j}$ の関係を後で代入する方が, より処理を機械的に行えるだけではなく, 余計な定義や処理が不要になるからです.

それから, Fox の微分子を作用させた後に, 結び目群 $G(K)$ の Wirtinger 表示による生成元を全て変数 t で置換し, 非可換積とその冪も全て可換積とその冪に置換します. この処理によって, D.fox: の Jacobian は $\mathbb{Z}[t^{-1}, t]$ の元を成分を持つ正方行列になります. この正方行列を Alexander 行列と呼びます.

結び目群 $G(K)$ の Alexander 多項式と呼ばれる多項式は, この Alexander 行列の余因子行列の各成分の最大公約因子の事です. 正確にはこの最大公約因子は一次の Alexander 多項式と呼ばれるものですが, 結び目理論ではこの多項式の事を Alexander 行列と呼びます.

猶, Alexander 多項式は Laurant 多項式環 $\mathbb{Z}[t^{-1}, t]$ のイデアルの生成元になります. その為, Alexander 行列を計算した時に, $t^n, n \in \mathbb{Z}$ の違いが生じる事がありますが, Alexander 多項式として定数項が零にならない様に各項の次数が正となるものを選択します.

この処理を纏めたプログラム calcAlexanderPoly を収録したファイル AlexanderPoly.mc の内容を以下に示します.

```

/* MAXIMA */

/* calcAlexanderPoly

fox.mc と併用する事が前提です.

結び目 G はリストで表現します.
即ち,  $G = \langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$  を  $G: [[x_1, \dots, x_n], [r_1, \dots, r_m]]$ 
で表現します.

calcAlexanderMatrix では, Wirtinger 表示の結び目群に対して
Alexander 行列の計算を行います. その為, 関係子  $r_1, \dots, r_m$  の
個数  $m$  は  $n$  か  $n-1$  に等しくなければなりません. 従って, 関係子の
個数が  $n$  か  $n-1$  に等しくなければエラーになります.
*/
calcAlexanderPoly(G) :=
block(
  [vars:G[1],rels:G[2],amat,Alex:[],
   Ia,dfx,rdfx,rdfx,crdfx,mrow,n:length(G[1]),
   m:length(G[2]),AlexanderPoly,APolyData:false,
   tmp,lst:[]],
  /* 結び目群の Wirtinger 表示に限定 */
  if m=n or m=n-1 then
  (
    Ia:subst("[",matrix,ident(n)),
    for i from 1 thru n do append(lst,[i]),
    dfx:map("D_fox:",vars),
    /* 関係子リストに演算子を作用 */
    rdfx:map(lambda([x],
                  apply1(D_fox:x,Dfox_Prod,
                        Dfox_PPower,
                        Dfox_NPower))),

```

```

        rels),
/* 微分に 1 か 0 を設定 */
for i from 1 thru n do
  ( mrow:Ia[i],
    rdfx2:rdfx,
    for j from 1 thru n do
      (
        rdfx2:map(lambda([x],
          subst(mrow[j],dfx[j],x)),rdfx2)
      ),
/* 非可換積の冪を可換積の冪に変換 */
    crdfx:map(lambda([x],
      subst("^","^^",x)),rdfx2),
/* 非可換積を可換積に変換し、簡易化を実施 */
    amat[i]:ratsimp(map(lambda([x],
      subst("*",".",x)),crdfx))
  ),
/* Alexander 行列を配列として最初に構築. */
for i from 1 thru n do
  Alex:append(Alex,[amat[i]]),
/* 配列データから行列データに変換 */
Alex:substpart(matrix,Alex,0),
/* 変数を t に変換します */
for i in vars do
  Alex:subst(t,i,Alex),
Alex:ratsimp(Alex),
/* Wirtinger 表現で関係子の総数が生成元の個数 n よりも
  一つ少ない場合の処理.Maxima の組込函数では余因子
  行列の生成は正方行列に限定されます.
*/
if m=n-1 then
  (
    /* nx1 の零行列を n 列目に追加. */
    tmp:addcol(Alex,zeromatrix(n,1)),

```

```

        /* 小行列式を計算. */
        tmp:map(lambda([x],
            determinant(minor(tmp,x,n))),lst)
    )
    else
        tmp:expand(adjoint(Alex)),
        /* 余因子行列をリストに変換. */
        tmp:substpart("[",tmp,0),
        /* LGCD で Alexander 多項式を抽出 */
        AlexanderPoly:num(LGCD(map(LGCD,tmp))),
        /* Alexander 行列と多項式のリストを生成. */
        APolyData:[Alex,AlexanderPoly]
    )
    else
        /* エラー処理.Error!と表示するだけのシンプルなもの */
        print("Error!"),
        /* Alexander 行列を返します. */
        return(APolyData)
    )$
/* リストから最大公約因子を計算します.

Lp: 多項式, 或いは, 整数で構成されるリスト.

長さが 1 の場合, その成分をそのまま返します.
長さが 2 以上の場合, 先頭の二つの成分の最大公約因子を計算し,
その最大公約因子と頭の Lp の成分を二つ取り除いたリストを合せ
たリストを用いて, 再帰的に LGCD を呼出しています.
*/
LGCD(Lp):=
block(
    [a1,n,lgcd:false],
    /* Lp がリストであるかを確認. */
    if listp(Lp) then
        (

```

```

/* Lp の長さを求めます. */
n:length(Lp),
/* Lp の長さが 1 の場合, 成分をそのまま返します. */
if n=1 then
  lgcd:first(Lp)
else
  (
    /* Lp の先頭二つで GCD を計算. */
    a1:gcd(Lp[1],Lp[2]),
    /* LGCD を再帰的に呼出しています.
       猶,rest(Lp,2) で先頭の二つの成分を削除した
       リストを生成し,append で二つのリストを結合します.
    */
    lgcd:LGCD(append([a1],rest(Lp,2)))
  )
)
else
  lgcd,
/* lgcd を返して終わります. */
return(lgcd)
)$

```

この AlexanderPoly.mc ファイルは calcAlexanderPoly 関数と LGCD 関数の二つで構成されています. calcAlexanderPoly 関数は計算した Alexander 行列と多項式をリストの形式で返します. 内部で処理している事は, 与えられた結び目群から生成元と関係子を取り出し, 生成元に D.fox: を作用させたリストを生成します. このリストへの作用は map 関数 を用います. 次に, 生成元の個数 n から n 次の単位行列 I_n を生成します. この単位行列の i 番目の行は i 列のみが 1 となるので, $D.fox : x_j \Leftrightarrow \frac{dx_j}{dx_i} = \delta_{ij}$ への変換に用います. ここで, lambda 関数を用いて臨時の関数を構築し, その関数に対して subst 関数を用いて変換を行います.

それから, 非可換積を可換積, 非可換積の冪を可換積の冪へと変換します. この変換では substpart 関数を用います. Maxima の演算子は式の表現で 0 番目の位置に置かれる為, 0 番目の成分を入れ換える様にしています. ここでも相手がリストの為, map 関数を効果的に適用する為に, lambda 関数を使って, 臨時の関数を定義しています. それからリストの形式から行列データに substpart 関数を用いて変換し, 各変数を全て t に

置換えると Alexander 行列が出来ます.

次に Alexander 行列の余因子行列を計算しなければなりませんが, 余因子行列を計算する adjoint 関数や小行列を求める minor 関数も正方行列のみにしか使えません. Wirtinger 表示の場合, 変数よりも一つ関係子が少なくても問題が無い為, これでは余因子行列が計算出来ない事があります. そこで, 変数よりも関係子が一つ少ない場合には, Alexander 行列の n 列目に n 次の零ベクトルを addcol 関数を使って追加します. それから minor 関数で Alexander 行列 Alex の小行列 $\text{Alex}_{1 \leq i \leq n, n}$ を minor 関数を用いて計算し, determinant 関数 を map 関数を適用する事で作用させて必要な余因子行列を計算します. すると, 多項式成分の行列/リストが出来ますが, その成分の最大公約因子を求める為に, 新たに構築した LGCD 関数を使います.

この LGCD 関数はリストに含まれる各成分の最大公約因子を計算する関数で, リストの長さが 1 であれば, リストに含まれる式をそのまま返し, 長さが 2 以上であれば, 頭の二つに gcd 関数を使って最大公約因子を求め, 頭二つを削ったリストの先頭に結果を加えたリストを使って LGCD 関数を呼出すという再帰的な手法を用いています. この再帰的な手法は LISP で顕著な特徴の一つで, Maxima でもこの様に利用出来ます. 但し, 安易な再帰的な手法は速度的な面で問題が出易い傾向があります. 猶, ここでの処理では大きな行列 (例えば, 10×10 の行列) を扱う事は考えていない為, この処理で特に大きな問題にはならないでしょう. 但し, 非力な計算機 (Pentium 100MHz 以下) では問題があるかもしれません. この辺は工夫されると面白いかと思います.

それから, 余因子行列をリストに変更し, この LGCD 関数を用いて最大公約因子を求めます. この時, 最初に map 関数で余因子行列リストの行成分の最大公約因子を計算すれば, 行の最大公約因子のリストが得られます. そのリストに今度も LGCD 関数を適用すれば, 最終的に Alexander 多項式が得られます.

13.7 Alexander 多項式の意味

この Alexander 行列と Alexander 行列の意味をもう少し詳しく説明しましょう. 先ず, Fox の微分 ∂x_i は, 結び目群の Wirtinger 表現の群の生成元 $x_1, \dots, x_n$ で生成される自由群 F_n から構築される群環 $\mathbb{Z}F_n$ から $\mathbb{Z}F_n$ への環準同形写像となります. そして, 群環 $\mathbb{Z}F_n$ から結び目群の群環 $\mathbb{Z}G(K)$ への写像は, 包含写像 i から導かれる自然な包含写像 i があります. この包含写像 i と Fox の微分 ∂ を合成した関数を ∂ と記述しましょう. すると, 写像 $\partial: \mathbb{Z}F_n \rightarrow \mathbb{Z}G(K)$ が得られます.

それから, 群環 $\mathbb{Z}G(K)$ から $\mathbb{Z}[t^{-1}, t]$ の写像として, 生成元 x_i を全て t で置換えるものを考えましたが, 幾何学的には, t には結び目を一回りする閉道が対応します. 但

し, この閉道は基本群の閉道とは異なり, 点 P に束縛されないものです. 正確には, この t は結び目の補空間 $C(K)$ の一次の Homology 群 $H_1(C(K))$ の生成元になります.

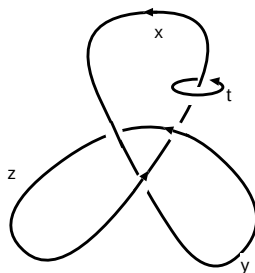


図 13.10: t の意味

因に, 群 $H_1(G(K))$ は, 基本群を可換化したものと同相になる事が知られています (Hurwitz の定理). 実際, 一点で拘束した輪を点から外してしまえば, 結び目の回りを好きな所に置く事が出来ますね. この一次の Homology 群で 0 になる元は, 結び目の補空間 $C(K)$ の内部で曲面を張る事の出来るものになります. 基本群の場合は, 円板に限定されていたので, その分, 条件としては緩いものになっています.

$\mathbb{Z}G(K)$ から $\mathbb{Z}[t^{-1}, t]$ への写像は全射になります. 以上の事実から, 次の写像の列が得られます.

$$\mathbb{Z}F_n \xrightarrow{\partial} \mathbb{Z}F_n \xrightarrow{i} \mathbb{Z}G(K) \xrightarrow{a} \mathbb{Z}[t^{-1}, t] \longrightarrow 0$$

ここで, 二つの写像の合成 $i \circ \partial$ で上記の列を置換えると, この写像の列は完全列と呼ばれるものになっています. ここで Alexander 行列は ∂ の個所に現れます. この Alexander 行列をかけたものが, 可換化写像 a の核になります.

この Alexander 行列と多項式には別の幾何学的な表現があります. これは非常に視覚的に強烈ですが, 普遍被覆空間等と準備するものが多くなるので, ここでは述べません. 但し, この事に関しては, Rolfsen の Publish or Perish (鹹が嫌なら論文を書け) という凄い名前前の出版社から出ている Rolfsen の名著 [52] に詳しく説明されています. 意味は判らなくても不思議な絵を図書館で眺めるだけの価値は十分あると思います.

13.8 Alexander 多項式で結び目を分類しよう

星型結び目 (5₁)

先程から出ている星型結び目の Alexander 多項式を計算しましょう. 先ず, 星型結び目の結び目群の Wirtinger 表示を以下に示します.

$$\langle v, w, x, y, z | xz^{-1}x^{-1}v, vx^{-1}v^{-1}y, yv^{-1}y^{-1}w, wy^{-1}w^{-1}z, zw^{-1}z^{-1}x \rangle$$

では, 早速計算しましょう. この例題では, 予め, `load("fox.mc")` と `load("AlexanderPoly.mc")` が実行されているものとします.

```
(%i14) star: [[x,y,z,v,w],
[x.z^(-1).x^(-1).v,v.x^(-1). v^(-1). y,
y.v^(-1).y^(-1). w,
w.y^(-1). w^(-1).z,
z.w^(-1).z^(-1).x]]$
(%i15) K5_1:calcAlexanderPoly(star)$
(%i16) K5_1[2];

              4      3      2
(%o16)          t  - t  + t  - t + 1
(%i17)tex(K5_1[1]);
$$\pmatrix{\frac{t-1}{t}&-1&0&0&\frac{1}{t}}\cr 0&\frac{1}{t}&\frac{t-1}{t}&-1&0}\cr -1&0&0&\frac{1}{t}&\frac{t-1}{t}}\cr \frac{1}{t}&\frac{t-1}{t}&-1&0&0}\cr 0&0&\frac{1}{t}&\frac{t-1}{t}&-1}\cr }$$
(%o17)                                     false
```

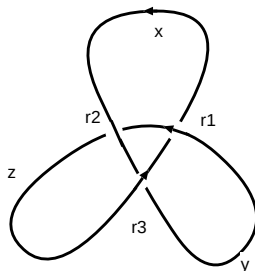
星型結び目の Alexander 多項式として $t^4 - t^3 + t^2 - t + 1$ が得られます, 従って, この結び目は自明な結び目と違う事が判りました.

この結び目の Alexander 行列は, `tex` 関数で出力した結果を $\text{T}_{\text{E}}\text{X}$ のソースファイルに入れば以下の様に表示されます.

$$\begin{pmatrix} \frac{t-1}{t} & -1 & 0 & 0 & \frac{1}{t} \\ 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 \\ -1 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} \\ \frac{1}{t} & \frac{t-1}{t} & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 \end{pmatrix}$$

クローバー結び目 (3_1)

クローバー結び目は三葉結び目 (trefoil) と呼ばれ, 図 13.11 に示す様に, 三枚の葉がある結び目です.

図 13.11: クローバー結び目 (3_1)

クローバー結び目の結び目群

$$\langle x, y, z \mid xyz^{-1}y^{-1}, yzx^{-1}z^{-1}, zxy^{-1}x^{-1} \rangle$$

```
(%i1) load("AlexanderPoly.mc");
(%o1) AlexanderPoly.mc
(%i2) Trefoil: [[x,y,z],
[x.y.z^(-1).y^(-1), y.z.x^(-1).z^(-1), z.x.y^(-1).x^(-1)]]$
(%i3) calcAlexanderPoly(Trefoil);
[ 1 - t t - 1 ]
[                ] 2
[ t - 1 1 - t ], t - t + 1
[                ]
[ - t t - 1 1 ]
```

この calcAlexanderPoly では Alexander 行列と Alexander 多項式のリストを返します. この結果から, クローバー結び目の Alexander 多項式は $t^2 - 1 + 1$ である事が判ります. この様に Alexander 多項式が 1 でない為, クローバー結び目は自明な結び目と等しくない事, 即ち, 本当に結ばれている事が判ります.

13.9 結び目の連結和と Alexander 多項式

二つの向き付けられた結び目が与えられた時, 連結和によって, 新しい結び目を生成する事が出来ます.

結び目の連結和

1. 先ず, 二つの結び目 K_1 と K_2 を用意し, 各結び目に向きを入れておきます.
2. 結び目 K_i 上の点 P_i を中心とする半径 r の球 $B_i(r), i \in \{1, 2\}$ を取ります. これらの球は結び目の一部を含みますが, この際, 半径 r を十分小さく取れば, $B_i(r)$ をドーナツの様な形状にする事が出来ます.
3. これらの球 $B_i(r)$ を結び目 $K_i, i \in \{1, 2\}$ から結び目の曲線を含めて削除します.
4. 取り除いた個所で, 結び目の向きが一致する様に結び目 K_1 と K_2 を繋ぎ合えます.

この結び目を結び目 K_1 と K_2 の連結和と呼び, $K_1 \# K_2$ と表記します.

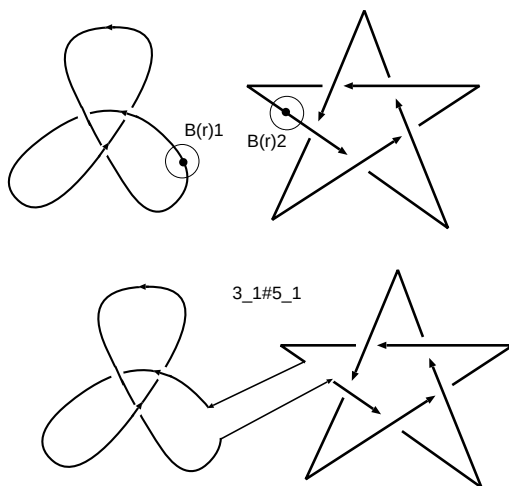


図 13.12: 結び目 3_1 と 5_1 との連結和

この時, 結び目の集合に, 連結和 $\#$ を入れる事で, 結び目の集合に半群の構造が入ります. 単位元は自明な結び目になります. 又, 与えられた結び目が連結和で生成された

ものでなければ既約な結び目と呼びます.

ここで, 3_1 と 5_1 の連結和の Alexander 多項式を計算してみましょう. この場合, $3_1 \# 5_1$ の Wirtinger 表示を直接計算しても構いませんが, 前節で計算した 3_1 と 5_1 の Wirtinger 表示があるので, それを利用しましょう.

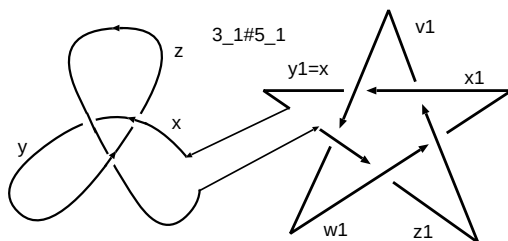


図 13.13: 結び目 3_1 と 5_1 との連結和の結び目群

$3_1 \# 5_1$ の場合, 図 13.13 の様に, K_1 側の結び目群はそのままにして, K_2 側の変数名を変更しておきます. そして, 繋げる個所の生成元に関係式を追加します. 例えば, この例では, x と y_1 の道を繋げるので $x = y_1$. 即ち, xy_1^{-1} を追加します. ここで `calcAlexanderPoly` は簡易的なプログラムなので関係子を一つ抜いておく必要があります. その為, k_2 側の本来の関係子を一つ抜いて, その代わりに xy_1^{-1} を入れておきます.

$$\left\langle x, y, z, x_1, y_1, z_1, v_1, w_1 \left| \begin{array}{l} xyz^{-1}y^{-1}, yzx^{-1}z^{-1}, \\ zxy^{-1}x^{-1}, v_1x_1^{-1}v_1^{-1}y_1, y_1v_1^{-1}y_1^{-1}w_1, \\ w_1y_1^{-1}w_1^{-1}z_1, xy_1^{-1} \end{array} \right. \right\rangle$$

```
(%i61) star1:subst(x1,x,star1)$
(%i62) star1:subst(y1,y,star1)$
(%i63) star1:subst(z1,z,star1)$
(%i64) star1:subst(v1,v,star1)$
(%i65) star1:subst(w1,w,star1)$
(%i66) ts:[append(Trefoil[1],star1[1]),
append(append(Trefoil[2],rest(star1[2],1)),[x.y1^(-1)])]$
(%i67) cs:calcAlexanderPoly(ts)$
```

```
(%i68) factor(cs[2]);
```

```
(%o68)          2          4    3    2
              (t  - t + 1) (t  - t  + t  - t + 1)
```

この様に連結和の Alexander 多項式は二つの結び目の Alexander 多項式の積になります.

これは, 何故でしょうか? 実際は非常に簡単な事で, 連結和の Alexander 行列を見ると判ります.

$$\text{--- } 3_1 \# 5_1 \text{ の Alexander 行列 ---}$$

$$\begin{pmatrix} 1 & -t & t-1 & 0 & 0 & 0 & 0 & 1 \\ t-1 & 1 & -t & 0 & 0 & 0 & 0 & 0 \\ -t & t-1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & \frac{1}{t} & 0 \\ 0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & 0 \\ 0 & 0 & 0 & \frac{t-1}{t} & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 \end{pmatrix}$$

如何でしょうか? クローバ結び目と星型結び目の Alexander 行列の成分が対角線上に二つのブロックで出現していますね. この行列から余因子行列を計算する訳ですが, 当然, 二つのブロックの行列の行列式の積になります.

この事から, 与えられた結び目の Alexander 多項式が既約でなければ, この結び目は Alexander 多項式の各既約因子に対応する結び目連結和で構成されている可能性があります. 逆に言えば, 結び目の Alexander 多項式が既約であれば, 結び目は既約である可能性があります. 残念な事ですが, 既約であるとは言えません. 何故なら, Alexander 多項式が 1 となる非自明な結び目が存在する事が知られているからです.

13.10 結び目の鏡像と Alexander 多項式

ここでは結び目 K の鏡像 $\bar{K}$ について考えましょう. 結び目を図 13.14 に示す

ここで, K_+ とその鏡像 K_- の違いは何でしょうか? それは交点の符号が逆になる事です. K_+ の場合, 交点の符号は全て $+1$ になりますが, K_- では全て -1 になっています. その為, K_- と K_+ は本質的に違う結び目の筈です. そこで, Alexander 多項式を計算してみましょう.

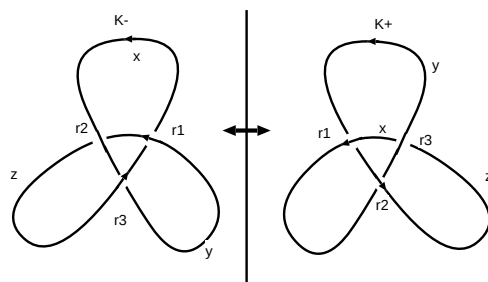


図 13.14: クローバー結び目の鏡像

————— K_- の結び目群 —————

$$\langle x, y, z \mid xzx^{-1}y^{-1}, yz^{-1}y^{-1}zy, yxy^{-1}z^{-1} \rangle$$

```
(%i20) TrefoilM: [[x,y,z],
[ x.z^(-1).x^(-1).y,z.y^(-1).
z^(-1).x,y.x^(-1).y^(-1).z]]$
(%i21) tm:calcAlexanderPoly(TrefoilM);

      [ t - 1      1          ]
      [ -----  -      - 1  ]
      [   t        t          ]
      [                               ]
      [   1          t - 1 ]    2
(%o21) [[ -      - 1  ----- ], t  - t + 1]
      [   t          t          ]
      [                               ]
      [          t - 1      1  ]
      [ - 1  -----  -      ]
      [          t        t          ]
```

ところが、 K_- の Alexander 多項式を計算して見ると、クローバー結び目の場合は鏡像の区別がつきません。

この様に Alexander 多項式は便利ですが残念な事に万能ではありません。

13.11 おまけ：スケイン多項式

Alexander 多項式の計算には, Wirtinger 表示等を用いた方法の他に, 幾何学的な手法で Seifert 曲面を貼って計算する方法, Dehn 手術と普遍被覆空間の構成で視覚的 (?) に求める方法があります. これらの方法も面白い方法ですが, これらとは全く別の方法で, 結び目の交点の局所的な入れ替えによるスケイン関係と呼ばれる関係式から多項式を計算する方法があります.

このスケイン関係で Alexander 多項式の他に, Conway 多項式, Kauffman 多項式, そして, Jones 多項式等の結び目の不変量となる多項式が色々計算出来ます.

ここでは簡単にスケイン関係を用いた結び目多項式の計算を解説します. ここで結び目の多項式は Laurant 多項式環 $\mathbb{Z}[A, A^{-1}, Z, Z^{-1}]$ のイデアルの生成元とします.

まず, スケイン関係式とは局所的に結び目の上下関係を入れ換えたものに対して結び目の多項式の間で成立する関係式の事です.

———— スケイン関係式 ————

$$1. \langle \bigcirc \rangle = 1$$

$$2. A^{-1} \langle \text{X} \rangle - A \langle \text{X} \rangle = Z \langle \text{ } \rangle \langle \text{ } \rangle$$

ここで, 上記の 1,2 を使うと, n 個の自明な結び目が並んでいる n 成分の自明な絡み目 $\bigcirc \cdots \bigcirc$ に対して, その絡み目の多項式として, $\langle \bigcirc \cdots \bigcirc \rangle = \left(\frac{A-A^{-1}}{Z} \right)^{n-1}$ が得られます.

この多項式の計算方法ですが, n 番目の結び目を捻じった $\bigcirc \cdots \bigcirc$ に上の 2 を適用すると以下の関係式が成立しますね.

$$A \langle \bigcirc \cdots \bigcirc \rangle - A^{-1} \langle \bigcirc \cdots \bigcirc \rangle = Z \langle \bigcirc \cdots \bigcirc \bigcirc \rangle$$

ここで, $\bigcirc \cdots \bigcirc$ と $\bigcirc \cdots \bigcirc$ は $\bigcirc \cdots \bigcirc$ に同値なので,

$K_n = \langle \bigcirc \cdots \bigcirc \rangle$ とすると, 以下の漸化式が得られます.

- $K_1 = 1$
- $(A - A^{-1})K_n = ZK_{n+1}$

この漸化式から計算出来ます.

猶, Skein 多項式の A と Z に変数を設定しなおす事で, 色々な結び目多項式が得られます.

———— スkein 多項式から得られる結び目多項式 ————

多項式名	A の値	Z の値
Alexander 多項式	1	$t^{1/2} - t^{1/2}$
Conway 多項式	1	Z
Jones 多項式	t	$t^{1/2} - t^{1/2}$

猶, Alexander 多項式は適当に $\pm t^{n/2}$ 倍する必要があります.

では, ここで図 13.15 に示すクローバー結び目の Skein 多項式を計算してみましょう.

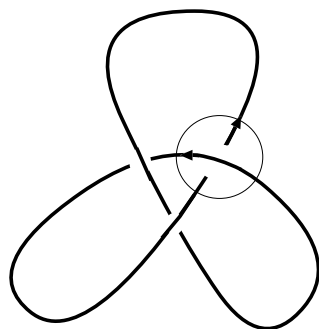


図 13.15: クローバー結び目

この図 13.15 で丸で囲った交点から Skein 関係を適用しましょう. この計算で本質的な事は, 図 13.16 に示す Skein 関係に関連する結び目や絡み目を描く事です.

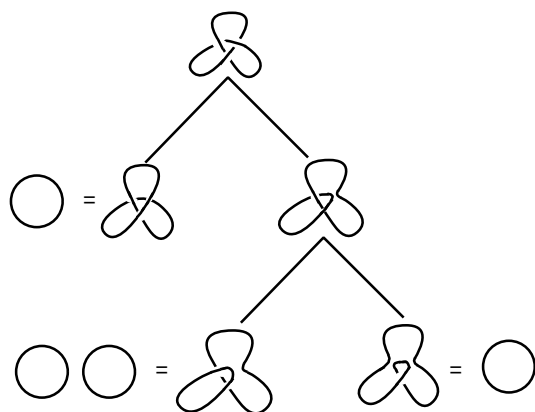


図 13.16: クローバー結び目のスケイン関係

このクローバー結び目の樹形図 13.16 の上 3 個から以下のスケイン関係式が得られます. $A^{-1} \langle \text{trefoil} \rangle - A \langle \text{trefoil} \rangle = Z \langle \text{trefoil} \rangle$

ここで, trefoil は自明な結び目と同値になる為, この結び目の多項式は 1 になります.

又, Hopf link は Hopf 絡み目と呼ばれる有名な絡み目です. そこで, Hopf 絡み目に対するスケイン関係式から多項式を計算します. 図 13.16 の下段がそれに対応します.

この下段から以下の関係式を得ます.

$$A^{-1} \langle \text{Hopf link} \rangle - A \langle \text{Hopf link} \rangle = Z \langle \text{Hopf link} \rangle$$

すると, Hopf link が自明な結び目と同値で Hopf link が二成分の自明な絡み目となるので,

Hopf 絡み目 Hopf link のスケイン多項式 $\langle \text{Hopf link} \rangle$ は $Z^{-1}A^{-2}(-AZ + A - A^{-1})$ とな

り, この結果から, trefoil のスケイン多項式 $\langle \text{trefoil} \rangle$ は $A^{-4}(A^2Z^2 + 2A^2 - 1)$ が得られます.

Alexander 多項式の場合, $A = 1, Z = t^{1/2} - t^{-1/2}$ を代入する事で, $t + t^{-1} - 1 \sim t^2 - t + 1$ が得られます.

このスケイン多項式で鏡像を計算する場合は, 単純に A を A^{-1} に, Z を $-Z$ に置換えるだけで計算が出来ます.

すると, 先程の三葉結び目の鏡像のスケイン多項式は, $A^2Z^2 - A^4 + 2A^2$ となります. これを Alexander 多項式に変換しても, $t^2 - t + 1$ となるので, 違いは判りません.

では, Jones 多項式で比較するとどうなるでしょうか?

—— 三葉結び目の鏡像で Jones 多項式を比較 ——

図 13.15 の Jones 多項式 $t^3 + t - 1$

図 13.15 の鏡像の Jones 多項式 $t^3 - t - 1$

この様に Jones 多項式では多項式自体が異なります. 即ち, Jones 多項式は Alexander 多項式や Conway 多項式よりも強力な結び目多項式なのです.

以上に示した様に, 基本群から Fox の微分子を使って計算した Alexander 多項式は, 中学生でも出来そうなお絵描きと多項式の計算になりました.

第14章 surfを使う話

この章では、外部アプリケーションを Maxima から利用する方法について、代数曲線や曲面を描画するアプリケーション surf を用いて解説します。

この解説では、surf を呼出して曲線や曲面を描画する surfplot という名前の関数を構築します。この関数は 2 変数の多項式を与えれば曲線、3 変数の多項式を与えると曲面を描きます。面白くて綺麗な絵が描けるので色々遊んでみて下さい。

14.1 代数曲面

標準的な Maxima で描く事の出来るグラフは $y = f(x)$ や $z = h(x, y)$ の書式や助変数を用いた関数のグラフに限定されます。その為、 $x^2 + y^2 - 1 = 0$ を満す点のグラフを描きたい場合、 $y = \sqrt{1 - x^2}$ と $y = -\sqrt{1 - x^2}$ のグラフを同時に描くか、 $x(t) = \frac{2t}{t^2+1}$ と $y(t) = \frac{t^2-1}{t^2+1}$ を描く等の工夫が必要になります。

そこで、2 変数や 3 変数の多項式の零点集合が簡単に描ける surf の登場となります。

14.2 surf の概要

surf は変数 x, y, z の多項式 p の零点集合 $V(p)$ を描くアプリケーションです。単純な GUI を持っており、直接スクリプトを編集して描画する事も容易に出来ます。C 風の簡単な処理言語を持っていますが、複雑な処理は出来ません。基本的に幾つかの基本的な設定を入力し、その設定に対して多項式の零点を描く事と多少の反復処理が出来る程度です。

surf の詳細はこの本では述べません。surf のサイト <http://surf.sourceforge.net/> のオンラインマニュアル <http://surf.sourceforge.net/doc.shtml> や、私の webpage ¹ を参照して下さい。

¹<http://www.bekkoame.ne.jp/ponpoko/Math/surf/SurfExamples.html>

猶, 残念な事です, surf は UNIX 環境でしか動作しないので, MS-Windows 上でどうしても使いたい方は, VirtualBox や VMware Player を使って, KNOPPIX/Math で遊んで下さい.

14.3 Maxima から surf を使う方法

Maxima から surf を使う最も簡単な方法は system 関数 を用いて surf を呼出す事です. 勿論, 単純に呼出しても Maxima 側の多項式が surf に引渡される事はありません. そこで, 中間ファイルを Maxima 側で生成し, それを surf に引き渡す方法を採用します. この時, surf の起動時のオプションの -x を使って生成したファイルを surf で処理させます. 即ち, `surf -x <ファイル>` を実行すると, surf は起動時に指定したファイルを読み込み, そのファイルに記述されたスクリプトを実行します.

そうとなれば Maxima で surf 用のスクリプトを生成し, ファイルに書出す関数を構築すれば良いのです. 最も簡単な方法は, 多項式の変数を完全に x, y, z に限定し, 変数が 2 個あれば曲線, 3 個ならば曲面を描く様に予め用意した surf の設定ファイルに曲線や曲面の方程式として多項式を引渡せば良いのです.

但し, この方法では x, y, z 以外の変数が使えません. 更に, 設定ファイルを準備するのもあまりエレガントではありませんね. それに設定ファイルの内容を必要に応じて Maxima から変更出来ないのも面白くありません. ここではもう少し Maxima に複雑な処理を行わせましょう.

14.4 surf の設定の取込み方法

surf では, 曲線や曲面を描く際に, 方程式が満す解を求める為に, 解法の設定, 解の精度や反復の回数, そして, 絵の大きさを各々, root_finder, epsilon, iterations, width と height で行います. これらを surf に引渡す必要があります. 猶, これらの値は以下の値で十分でしょう.

—— 求解に関するパラメータとウィンドウの大きさ ——

```
root_finder=d_chain_bisection;
epsilon=0.0000000001;
iterations=20000;
width=500; height=500;
set_size;
```

先ず, `root_finder` に指定可能な解法は色々ありますが, 自己交差を持つ曲面, 曲面と平面との断面が綺麗に描ける `d_chain_bisection` 法をここでは採用します. 絵の大きさはデフォルトの 200×200 では小さ過ぎるので, 見栄えの良い 500×500 で表示させる事にしましょう.

曲面の場合は上述の設定に加えて, 以下の設定を追加します.

—— 曲面の場合の設定 ——

```
do_background=yes;  
background_red=255;  
background_green=255;  
background_blue=255;  
rot_x=0.14; rot_y=-0.3;  
rot_z=0;  
scale_x=1.0; scale_y=1.0;  
scale_z=1.0;
```

先ず, `rot_x`, `rot_y` `rot_z` は曲面を X 軸, Y 軸と Z 軸回りで曲面を回転させるパラメータです. 次に, `scale_x`, `scale_y` `scale_z` は曲面の X 軸, Y 軸と Z 軸方向の拡大率になります. これら以外の設定は背景色の指定パラメータになります.

これらの値をファイルに書込むのが面倒であれば, 大域変数にしてしまえば便利です. しかし, Maxima の属性を用いるともっと扱い易くなります.

今回, Maxima の初期化ファイル `maxima-init.mac` に以下の設定を行います.

maxima-init.mac での属性設定

```
put(surfg, "d_chain_bisection",root_finder);
put(surfg, 20000,iterations);
put(surfg, 500,width);
put(surfg, 500,height);

put(surf, "yes",do_background);
put(surf, 255,background_red);
put(surf, 255,background_green);
put(surf, 255,background_blue);
put(surf, 0.14,rot_x);
put(surf, -0.3,rot_y);
put(surf, 0.0, rot_z);
put(surf, 1.0, scale_x);
put(surf, 1.0, scale_y);
put(surf, 1.0, scale_z);
```

ここでは Maxima の put 函数を用いて大域変数 surfg と大域変数 surf の属性として, surf の描画パラメータを指定します. 例えば, `put(surf, -0.3,rot_y)` で, 変数 surf の属性 rot_y に値-0.3 が設定されます.

初期設定の変更も同じ方法で値を変更すれば出来ます. これらの属性値を Maxima で取出す場合, get 函数を用います. 例えば, surf の background_red 属性に 255 を設定したければ, `put(surf, 255,background_red);` と入力します. 又, surf の background_red の属性値は `get(surf,background_red);` で得られます.

これで, 二次元と三次元の共用の設定が属性に含まれる大域変数 surfg と曲面専用の設定が含まれる大域変数 surf の属性が設定されます.

次に, surf の描画ファイルに大域変数 surf と surfg に記述した属性を書出を行う部分の構成を考えましょう. 先ず, surf と surfg の属性は 10 個あります. この属性を一々函数に書込むのは流石に面倒です. そこで, properties 函数を使って, surf と surfg の属性に何があるか一覧のグラフを出させ, その属性名と属性値の等式を作る方法を考えましょう.

要するに, 次の処理を自動的に行う様に do 文を組込めば良いのです.

```
(%i18) properties(surfg);
(%o18) [["user properties",height,width,iterations,root_finder]]
```



```
(%i19) properties(surf);
(%o19) [["user properties",scale_z, scale_y, scale_x, rot_z,rot_y,rot_x,
        background_blue,background_green,background_red,do_background]]
(%i20) %[1][6]=get(surf,%[1][6]);
(%o20) rot_y = -0.3
```

この例では,properties 関数で大域変数の属性リストを出させます. この属性リストは通常, 第一成分のリストに属性名が登録されています. 先頭は user properties なので, 第 2 番目以降の成分を演算子=の左側に置き, その成分の属性値を演算子=の右側に置けば surf 向けの等式が出来る訳です.

この処理で構築した等式 (上の例では,rot_y=-0.3) を Maxima の配列に入れて, stringout 関数でこの配列の内容をファイルに書出してしまえば良いのです. この様にしておけば, 別に変数名 (Maxima では属性) を憶えていなくても, 機械的に処理が行えるので, その属性を減らしたり増したりするのも容易になりますね.

14.5 多項式の処理

surf で用いる数値で $1.0e-3$ は使えますが,Maxima の様に bigfloat 等の数値はありません. そのまま Maxima の不動点少数を出力させると書式上の問題で,surf ではエラーになります. そこで, この様な余計な問題を避ける為, 多項式を expand 関数で一旦展開した後, ratsimp 関数で簡約化します.ratsimp 関数を使うと多項式の係数は有理数に近似されます. その為,surf の処理で問題は一切生じなくなります.

次に重要な事です, surf で利用可能な多項式は基本的に変数 x, y, z の多項式に限定されます. 勿論,surf は x, y, z 以外の変数も扱えますが, その様な変数は内部での補助的な変数として扱われ, あくまでも描かれるグラフは XY 平面, 或いは XYZ 空間に限定されます.

その為, 多項式の変数を 2 か 3 個に限定する必要があり, x, y, z 以外の変数が使われていれば, それらを x, y, z で置換える操作もあった方が良いでしょう.

ここで, 多項式の変数を返す関数として Maxima には showratvars 関数があります.

例えば, `showratvars(x+y^2+a^3);` と入力すると, `[a,x,y]` が返されます. このリストは Maxima の変数順序 $>_m$ に対して小さな順で変数が左から並びます. ここで式に `sqrt(2)` の様な関数が含まれていると, この関数を変数リストに追加してしまうので, この現象を避ける為に多項式に float 関数を予め作用させておきます.

Maxima の変数順序は基本的に逆アルファベット順 (a よりも z が小さい. 又, 小文字は大文字よりも小さい) となるので, 変数が全て小文字, 或いは大文字であれば, 順序

$>_m$ に対して小さいものから順番に並べてしまうと、実際は通常のアルファベット順になります。そこで、showratvars 関数で得られた変数リスト vars の左側から順番に x, y, z を対応させれば良い事になります。

具体的には vars[1] を x , vars[2] を y , vars[3] を z で置換えるので、subst 関数を使いそうです。ところが、subst 関数では、置換リストを与えると左から右へと代入が行われる為、`subst([a=x,x=y,y=z],a+x+y)` の結果は困った事に `3*z` になってしまっています。

そこで、置換を二段階で行います。関数に局所変数 surf_tmp_x, surf_tmp_y と surf_tmp_z に一旦置換え、これらを $[x, y, z]$ に最終的に置換え事で多項式を変数 x, y, z の多項式に変換します。以上で変数変換を終えます。

実際に surf で曲線や曲面を描く為には、その方程式に対応する surf の描画命令をスクリプトの末尾に追加しなければなりません。この処理は曲線と曲面で異なるので、曲線と曲面の場合に分けて surf の描画命令を追加します。

曲線の場合、多項式を変数 curve に割当てて、最後に draw_curve 命令にを追加します。又、曲面であれば、多項式を変数 surface に割当て、最後に描画命令である draw_curve 命令を追加します。

この描画命令を追加したスクリプトを、Maxima の system 関数から、surf に実行させる事が出来ます。

この考えで構築した Maxima の関数 surfplot を以下に示します。

```
/* Maxima */

/* 属性の設定.*/
/* surfg の属性設定

    surfg      平面曲線と空間曲面を描く際に用いる共通の設定.
    root_finder 零点を計算する際の、解法の指定.
                  d_chain_bisection を用いると自己交差も綺麗に
                  描きます.
    iterations: 零点を計算する際の繰返しの上限を設定
    width:      画像の横幅
    height:     画像の高さ
*/
put(surf, d_chain_bisection, root_finder);
put(surf, 0.0000000001, epsilon);
```

```
put(surfg, 20000, iterations);
put(surfg, 500, width);
put(surfg, 500, height);

/* surf の属性設定

surf には空間曲面を描く際に用いる設定を入れます.
surf で色の指定は 0 から 255 までの整数で RGB を指定します.
do_background:    背景色の設定を許可 (yes)
background_red:   背景色の指定 (赤)
background_green: 背景色の指定 (緑)
background_blue:  背景色の指定 (青)
rot_x:            X 軸回りの回転角度 (rad)
rot_y:            X 軸回りの回転角度 (rad)
rot_z:            Z 軸回りの回転角度 (rad)
scale_x:          X 軸方向の倍率
scale_y:          Y 軸方向の倍率
scale_z:          Z 軸方向の倍率

*/
put(surf, yes, do_background);
put(surf, 0, background_red);
put(surf, 0, background_green);
put(surf, 0, background_blue);
put(surf, 0.14, rot_x);
put(surf, -0.3, rot_y);
put(surf, 0.0, rot_z);
put(surf, 1.0, scale_x);
put(surf, 1.0, scale_y);
put(surf, 1.0, scale_z);

/* surfplot
```

引数は多項式. 多項式の変数は 2 個か 3 個でなければエラーに

なります。描画は `surf` を用いますが、この関数では、臨時ファイルとして `surf.tmp` に `surf` のスクリプトを書込み、`system` 関数で `surf -x surf.tmp` を実行させます。

```

*/

surfplot(f):=block(
[
poly,poly0,vars,lls1:0,lls2:0,
f:ratsimp(expand(f)),tmp,
str,target,j,sl,obj,
ls1:properties(surfg),
ls2:properties(surf)
],
vars:showratvars(float(f)),
n:length(vars),
display2d:false,
if n=2 or n=3 then
  (lls1:length(ls1[1])-1,
   for i:1 thru lls1 do
     (str:ls1[1][i+1],
      (if str=epsilon then tmp:rat(get(surfg,str))
       else tmp:get(surfg,str)),
      surf_settings[i]:str=tmp
     ),
   if n=3 then
     (lls2:length(ls2[1])-1,
      for i:1 thru lls2 do
        (str:ls2[1][i+1],
         j:i+lls1,
         surf_settings[j]:str=get(surf,str)
        ),
      /* 変数の入換を行います。*/
      poly0:subst([vars[1]=surf_tmp_x,vars[2]=surf_tmp_y,
                   vars[3]=surf_tmp_z],f),

```

```

        poly:subst([surf_tmp_x=x,surf_tmp_y=y,surf_tmp_z=z],
                    poly0),
        /* 曲面を描く為の設定 */
        target:surface,
        obj:draw_surface)
else
(
    /* 変数の入換を行います.*/
    poly0:subst([vars[1]=surf_tmp_x,vars[2]=surf_tmp_y],f),
    poly:subst([surf_tmp_x=x,surf_tmp_y=y],poly0),
    /* 曲線を描く為の設定 */
    target:curve,
    obj:draw_curve),
/* 配列 s1 の定義. 描画設定と曲線/曲面の方程式+描画命令を割当 */
j:11s1+11s2,
array(s1,j+2),
(for i:0 thru j-1 do
    s1[i]:surf_settings[i+1]),
s1[j]:target=poly,
s1[j+1]:obj,
/* 只今, 描画中... */
print("Surf is now drawing ", poly, ". Please wait ...."),
/* stringout 関数で, スクリプトの式を書込みます. */
if n=2 then
    (stringout("surf.tmp",clear_screen,s1[0],s1[1],s1[2],s1[3],
               s1[4],s1[5],s1[6]))
else
    (stringout("surf.tmp",clear_screen,s1[0],s1[1],s1[2],s1[3],
               s1[4],s1[5],s1[6],s1[7],s1[8],s1[9],
               s1[10],s1[11],s1[12],s1[13],s1[14],
               s1[15],s1[16])),

/* system 関数で surf を起動します.XView 版の surf の場合は,
-x オプションを外すと動く様です */

```

```

    system("surf -x surf.tmp>surf.log&")
)
else
/* 多項式が,2変数でも3変数でもなければエラーを表示します. */
print("Error!"))$

```

この函数を使うと 2 変数多項式や 3 変数多項式の零点のグラフが描けます. `surfplot` を実行すると, 直ちに `surf` が立ち上ります. 描画は計算機的能力と式の複雑さに依存する為, どの程度かかるかは一概に言えませんが, 最近の Pentium 3 の 1GHz 程度でもあれば, 余程複雑な曲面でもない限り, 30 秒もあれば描画が完了します.

描いた絵は `surf` の側で保存も出来ます. この様な操作は描画ウィンドウ上で, マウスの右ボタンを押すと, 制御用のウィンドウが出るので, そこで色々操作を行います. 詳細は `surf` のマニュアルを参照して下さい.

`surf` の終了は描画ウィンドウ上でキーボードから `q` を入力すれば, `surf` を終了して Maxima に制御が戻ります.

猶, ここでは敢てしませんでした, `status` 函数, 場合によっては `sstatus` 函数を併用して, UNIX 環境だけでパッケージが使える様にする事が可能です. 具体的には `textttstatus(feature,unix)` の値が `true` になれば, 描画プログラムを実行させ, それ以外では「UNIX 環境でなければ使えません」等と表示させる様にすれば良いのです. では, 色々曲面や曲線を描いて遊んでみましょう.

14.6 簡単な例

最初に半径が 1 の球面を描いてみましょう. 球面の方程式は $x^2 + y^2 + z^2 - 1 = 0$ で与えられます. 従って, `surfplot(x^2+y^2+z^2-1)` と入力しましょう. すると `surf` が立ち上がって球面を描きます. これだけでは面白く無いので, もう少し捻りの効いた方程式を描いてみましょう. では, 次の 3 個の球面の方程式の積はどのような曲面を描くでしょうか?

— 3 個の球面の方程式の積 —

$$(x^2 + y^2 + z^2 - 4) ((x - 2)^2 + (y - 2)^2 + z^2 - \sqrt{2}) ((x + 2)^2 + (y - 2)^2 + z^2 - \sqrt{2})$$

この 3 個の球面の方程式の積で表現される式は図 14.1 に示す 3 個の球面で構成された曲面になります. それにしても, この曲面は何処かで見た事のある様な気がしますね.

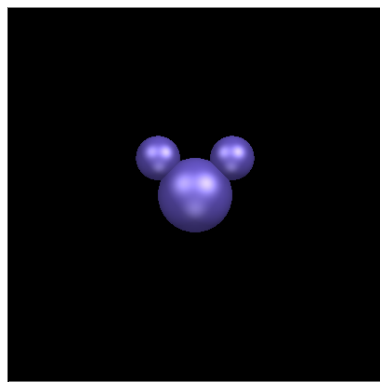


図 14.1: ? 曲面

この曲面は原点を中心とする半径 2 の球面を中心に, $(2, 2, 0)$ と $(-2, 2, 0)$ を中心とする半径 $\sqrt{2}$ の球面で出来ています. この様に複数の曲面の方程式の積で出来た方程式から得られる曲面は各曲面の和集合になります.

正確に言えば, 多項式 $f \in K[x_1, \dots, x_n]$ に対し, $V(f)$ を多項式 f の零点集合とします. この時, $V(f_1 \cdot f_2 \cdots f_n)$ は $V(f_1) \cup V(f_2) \cup \dots \cup V(f_n)$ となります.

従って, 多項式の零点集合を描く事は, 多項式の各因子の零点集合の和集合を描く事を意味します. この事から, 多項式の既約因子分解が出来ると, 後は各既約な多項式を調べてしまえば十分な事が判ります. 更に, 既約な元は多項式環 $K[x_1, \dots, x_n]$ の素イデアルの生成元となるので, 結局, 曲線や曲面は素イデアルと密接に関連する事になります.

この事を怪しい絵を描く方法に適用するのであれば, 片っ端から描きたい方程式の積を surf で描かせれば, 怪しい絵が作られる事になります. 例えば, 以下の 3 個の円と楕円の積を描いたものは図 14.2 に示すアヒルの顔の様なものになりますね.

— 3 個の円の方程式と楕円の積 —

$$(x^2 + y^2 - 9) * ((x-2)^2 + (y-1)^2 - 1) * ((x+2)^2 + (y-1)^2 - 1) * (x^2/9 + 2*(y+1)^2 - 1)$$

この様な巫山戯た使い方だけではなく, 座標を付けて曲線を描きたい場合にも使えます. この場合, 座標軸の方程式と曲線の方程式の積を描けば良いのです. 例えば, 原点を通る X 軸と Y 軸を描きたいのであれば, X 軸の方程式が $y = 0$, Y 軸の方程式が $x = 0$ となるので, 描きたい曲線に $x * y$ をかけたものを描けば良いのです. 例えば, `surfplot((x^2*(9-x^2)-4*y^2)*x*y);` と入力した結果を図 14.3 に示します.

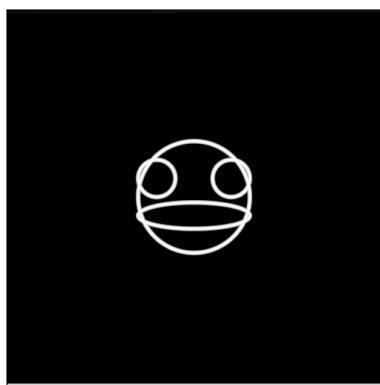


図 14.2: アヒル?

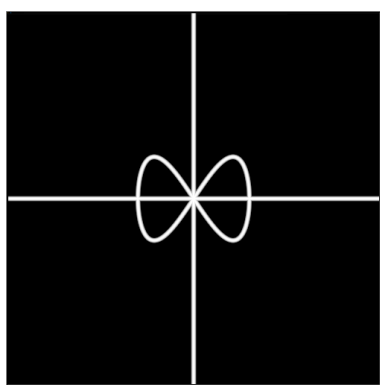


図 14.3: レムニスケート

この様にマジメな使い方も出来ます. が, 色々と巫山戯た使い方をするのも楽しいかと思います.

14.7 Barth Diec

さて, `surfplot` を使って派手な曲面を描いてみましょう. そこで, 以下に示す式の零点集合の曲面を `surfplot` で描いてみましょう.

Barth Diec

$$8(x^2 - \tau^4 y^2)(y^2 - \tau^4 z^2)(z^2 - \tau^4 x^2)(x^4 + y^4 + z^4 - 2(x^2 y^2 + y^2 z^2 + z^2 x^2)) \\ + (3 + 5\tau)(x^2 + y^2 + z^2 - 1)^2(x^2 + y^2 + z^2 - (2 - \tau))^2$$

$$\tau = \frac{1+\sqrt{5}}{2}$$

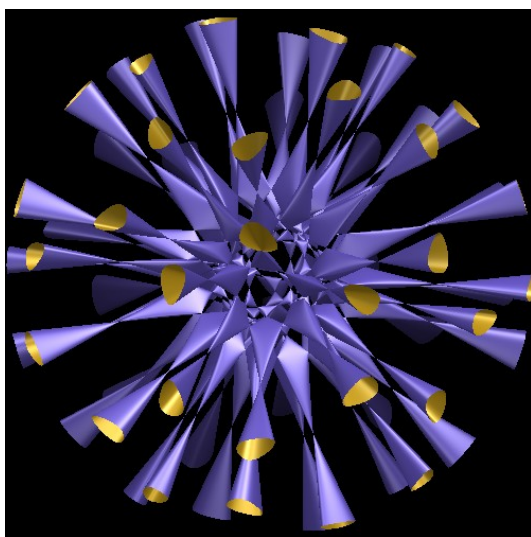


図 14.4: Barth Diec

この式の零点集合は図 14.4 に示す派手派手な曲面になります. ここで, τ を代数的数ではなく, 浮動小数点にするとどうなるでしょうか? その結果を図 14.5 に示しておきましょう.

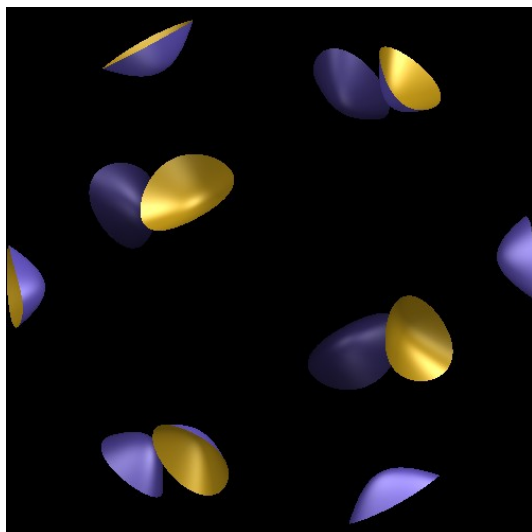


図 14.5: Barth Diec

この様に図 14.5 の結果は図 14.4 とは随分と異なる代物になります.

この現象が生じるのも, $\tau = \frac{1+\sqrt{5}}{2}$ を代数的数ではなく近似値 $\tau = 1.618033988749895$ を用いた為です. この様に近似多項式を用いた場合, 代数曲面の特異点の計算はあまり期待出来ません.

14.8 Steiner のローマ曲面ギャラリー

今度は Steiner のローマ曲面を描いてみましょう. ローマ曲面は $x^2y^2 + x^2z^2 + y^2z^2 - 17xyz = 0$ を満す点の集合で, この曲面は写像 $f: (x, y, z) \rightarrow (xy, yz, zx)$ で原点中心の半径 1 の球面を写したものです.

この曲面は二次元の実射影空間の三次元空間への嵌め込みと呼ばれる平面の三次元空間での置かれ方の一つになります. 因に, 二次元の射影空間は Moebius の帯の境界となる円周に円盤を貼って出来る曲面に同相になりますが, この曲面は通常の三次元空間では実現出来ません. どうしても, 自分自身が交わる点が出来てしまいます. その辺が面白い曲面です.

最初に, 上記のファイル (surfplot.mc とします) を読み込みましょう. 猶, Maxima の利用者定義の関数を含むファイルは load 関数で読みめます. surfplot.mc が Maxima を起動

したディレクトリ上にある場合, `load("surfplot.mc");` で Maxima に読み込みます. これでエラーも何も出なければ問題ありません. カレントディレクトリに `surfplot.mc` が無ければ, `surfplot.mc` を置いたディレクトリを直接指定します. この場合, 相対パス (カレントディレクトリを基準とするパス) でも絶対パス (ルートディレクトリを基準とするパス) でもどちらでも構いません.

では, 早速描いてみましょう.

`surfplot(x^2*y^2+x^2*z^2+y^2*z^2-17*x*y*z);` と入力して下さい.

すると, この関数はカレントディレクトリ上にデータファイル `surf.tmp` を生成し, `surf` に引渡し, `surf` が起動します. その結果, 図 14.6 に示す絵が得られます.

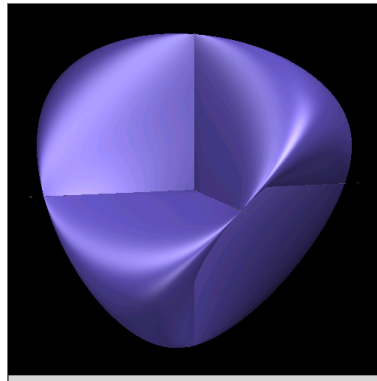


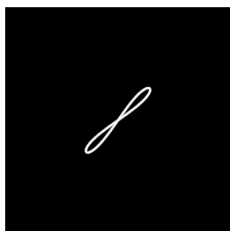
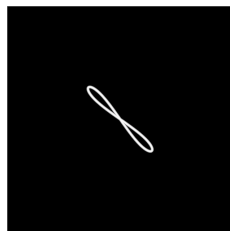
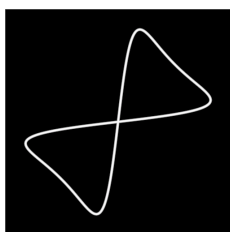
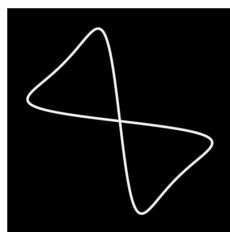
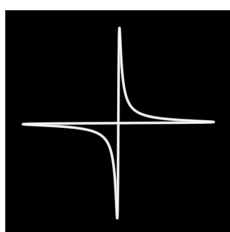
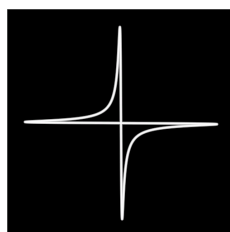
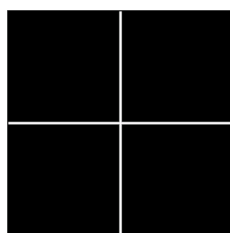
図 14.6: Steiner のローマ曲面

この絵で判る様に, ローマ曲面は X, Y, Z 軸で潰れた球面の様な形をしています. これだけでは何がどうなっているのか判りませんね. では, 曲面を平面で切った断面を見てみましょう. この平面による曲面の断面は Maxima+surf で簡単に見られます.

その為には, 断面の方程式を Maxima で計算しなければなりません. これは実は簡単な事です. 平面の方程式は $(0,0,0)$ と異なる 3 個の実数 (a, b, c) と実数 d を用いて, $ax + by + cz - d = 0$ で表現されます. ここで, a, b, c のどれか一つは 0 でないので, ここでは c が零でないとしましょう. すると, 方程式を c で割る事で, $z + ax + by - c = 0$ の形式の方程式が得られます. そこで, z に $c - ax - by$ を代入した式を Maxima で計算し, その式を描けば良いのです.

具体的に XY 平面 $z = 0$ による断面はどうなるのでしょうか? この場合, 断面の方程式として x^2y^2 が得られるので, 零点の集合として, X 軸と Y 軸が描かれてしまいます.

以下に, z を $[-8, -2, -0.1, 0, 0.1, 2, 8]$ で描いた結果を以降に示しておきます.

Steiner のローマ曲面, $Z=8$ Steiner のローマ曲面, $Z=-8$ Steiner のローマ曲面, $Z=2$ Steiner のローマ曲面, $Z=-2$ Steiner のローマ曲面, $Z=0.1$ Steiner のローマ曲面, $Z=-0.1$ 図 14.7: Steiner のローマ曲面, $Z=0$

これらの断面図は単純に Z 軸を法線とする平面で描いたものですが、この曲面の面白さが出ています。 $z = 0$ を除くと、断面は 8 の字になっています。 $z = 0$ に近づくにつれて、8 の字の上下が潰れて、交差点付近が XY 軸に貼り付き、 $z = 0$ を通り過ぎると、再び 8 の字が出現しますが、今度は潰れた 8 の字が 45 度回転した形で出現します。この曲線で自分自身が交わる点で、接ベクトルの次元は 2 になります。

14.9 SINGULAR を使ってみよう

SINGULAR は可換代数、代数幾何と特異点理論に特化した計算機代数システムです。SINGULAR は C 風の処理言語があり、この処理言語で記述したライブラリで機能を拡張して行く事が出来ます。SINGULAR 自体には GUI やエディタ、更にはグラフ表示機能を持っていませんが、Emacs や Surf といった外部プログラムで代用が出来ます。

14.9.1 SINGULAR 初歩

SINGULAR の式の入力は、Maxima と同様に、式の末尾にセミコロン ; を必ず付けます。セミコロンが無ければ、Maxima と同様に、SINGULAR は入力が継続していると判断して入力行の評価を行わず、プロンプトを ">" から "." にして入力待ちになります。SINGULAR の対象には全て型があります。整数の和、差、積のみは、そのまま入力しても、結果が帰って来ますが、それ以外は最初に基礎環 (Base Ring) を定義しなければなりません。

ここで基礎環の定義は以下の様に行います。

基礎環の定義方法

```
ring < 環の名前 > = < 係数体, (< 変数1>, ..., < 変数n>), < 順序 >;
```

ここで、基礎環の係数体には実数 $\mathbb{R}$ 、複素数 $\mathbb{C}$ 、有理数 $\mathbb{Q}$ 、整数 $\mathbb{Z}$ 等が選べ、標数 p は 2147483629 以下の整数が指定出来ます。更に、指定可能な順序の代表的なものには、辞書式順序 (lp)、斉次辞書式順序 (Dp)、斉次逆辞書式順序 (dp) と重み付き逆辞書式順序 (wp) と重み付き辞書式順序 (Wp)、負辞書式順序 (ls)、負斉次逆辞書式順序 (ds)、負斉次辞書式順序 (Ds)、負重み付き逆辞書式順序 (ws) と負重み付き辞書式順序 (Ws) 等が選べます。この様に、Maxima と比べ変数順序は非常に充実しているので、順序の特徴を利用したプログラムが書き易くなっています。

基礎環を定義すれば、その基礎環上の多項式 (poly) やイデアル (ideal)、更に商環 (qring) が扱えるようになります。以下に多項式、イデアルと商環の定義方法を示しましょう。

多項式, イdeal, 商環の定義方法

```

poly < 多項式名 > = < 多項式 >;
ideal < イdeal名 > = < 多項式1 >, ..., < 多項式n >;
ideal < 商環名 > = < イdeal >;

```

次に, 商環 R/I の定義で指定するイdeal I は単純に `ideal` で定義したものも使えますが, その場合はイdealが標準基底で無いと警告します. イdealの標準基底は `std` 命令を使って `std(< イdeal >);` で得られるので, この結果を用いましょう. 猶, 標準基底を計算する命令に `groebner` もありますが, `std` 命令と違って `groebner` 命令は環の順序に制約があります.

SINGULAR は入力された大文字と小文字を判別します. 一度定義した対象は, 名前を SINGULAR に入力すればその内容が表示されます. この点は Maxima と同じです.

次に各対象の定義例を示しておきましょう.

```

> ring r=0,(x,y,z),dp;
> poly unit_circle_eq=x^2+y^2-1;
> ideal unit_circle=x^2+y^2-1;
> unit_circle_eq;
x2+y2-1
> unit_circle;
unit_circle[1]=x2+y2-1
> ideal points=unit_circle_eq,y-x;
> points;
points[1]=x2+y2-1
points[2]=-x+y
> points[1];
x2+y2-1

```

この例では基礎環として $r = \mathbb{Q}[x, y, z]$ を定義し, それから多項式の `unit_circle_eq` とイdealの `points` を定義しています.

この様に SINGULAR は Maxima と違い, 値の割当では `=` を使います. 対象の内容は単純に対象名を入力すれば多項式 `unit_circle_eq` の様に返されていますね. ここで, イdealは番号付で返されるので, イdealを生成する多項式を取り出したい時は, この例の `points[1]` の様にすれば一番目の生成元が取り出せます. 猶, 取り出した生成元の型は多項式 (poly) 型になります.

次に, 商環の定義例を示します.

```

> qring ri=std(x^2+y^2-1);
> ri;
// characteristic : 0
// number of vars : 3
//      block 1 : ordering dp
//      : names x y z
//      block 2 : ordering C
// quotient ring from ideal
_[1]=x2+y2-1
> r;
// characteristic : 0
// number of vars : 3
//      block 1 : ordering dp
//      : names x y z
//      block 2 : ordering C

```

この例では前の基礎環 $\mathbb{Q}[x, y, z]$ 上で、商環の定義に必要なイデアルを $\text{std}(x^2 + y^2 - 1)$; で与え、商環 ri を $\mathbb{Q}[x, y, z] / \langle x^2 + y^2 - 1 \rangle$ で定義しています。

SINGULAR では環を複数定義する事が出来ますが、定義する毎にポインタが新規に定義された環に移動します。多項式、イデアル、写像等はポインタがある環上で定義されるので、必要に応じてポインタを戻す必要があります。ポインタをある環に移す場合には、`setring` 命令を使います。その使い方は `setring <環の名前>;` と環の名前を直接指定するだけです。

SINGULAR では写像が扱えます。この写像の定義では名前が他の数式処理のものとは紛らわしい名前ですが、`map` 函数を用います。この場合、対象が含まれる環を基礎環にした状態で写像の定義を行います。以下に写像の定義例を示しましょう。

```

> ring r1=0,(x,y,z),dp;
> ring r2=0,(a,b),dp;
> map f=r1,a,b,0;
> f;
f[1]=a
f[2]=b
f[3]=0

```

この例では、環 $r1$ と環 $r2$ を各々 $\mathbb{Q}[x, y, z], \mathbb{Z}[a, b]$ とし、環 $r1$ から環 $r2$ への写像 $f(x, y, z) \rightarrow (a, b, 0)$ を写像の値域となる基礎環の変数に対して一つ一つ定めています。

す. 基本的に, SINGULAR の写像の定義方法は, 基礎環の変数がどの様に写されるかを定めるだけです.

面白いのは, 定義した写像によるイデアルの逆像が計算可能な事でしょう. 例えば, 上の

例の環 r_2 のイデアル i_2 の写像 f による逆像は `preimage` 命令を使って `preimage(r1, f, i2);` で計算出来ます.

次の例ではイデアル i_2 を零イデアル (0) として, 写像 f の核を計算するものです.

```
> ring r1=0,(x,y,z),dp;
> ring r2=0,(a,b),dp;
> map f=r1,a,b,0;
> ideal i2=0;
> setring r1;
> preimage(r2,f,i2);
_[1]=z
```

猶, この例では零イデアルを `ideal i2 = 0;` で定義していますが, 単純に `ideal i2` としても同じ意味になります.

14.9.2 SINGULAR で surf を使ってみよう

SINGULAR はその処理言語を用いてライブラリが構築出来ます. このライブラリの読込は `LIB` 命令 (大文字) を用います. SINGULAR には様々なライブラリが附属していますが, 標準で附属するライブラリで, `surfplot.mc` の様に, SINGULAR で定義した関数を `surf` を用いて可視化する `surf.lib` があります. 実は, `surfplot.mc` の構成は, この `surf.lib` の内容を参考になっています.

この `surf.lib` を用いたローマ曲面の描き方の例を以下に示しておきます. 但し, Maxima と同じ事をしては面白く無いので, ここでは写像の定義と, その逆像を用いて表示する例を示しておきましょう.

```
> ring r=0,(x,y,z),dp;
> poly sp4=x2+y2+z2-16;
> ideal i1=sp4;
> map f=r,xy,yz,zx;
> ideal steiner=preimage(r,f,i1);
> steiner;
steiner[1]=x2y2+x2z2+y2z2-16xyz
```



```
> plot(steiner,"background_red=0;background_green=0;
. background_blue=0;rot_x=2;rot_y=0.5;");
```

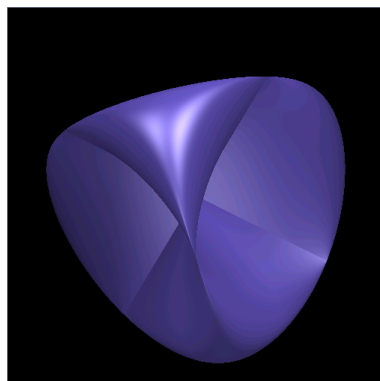


図 14.8: Steiner のローマ曲面

この例では, 写像 $f : (x, y, z) \rightarrow (xy, yz, zx)$ による半径 4 の三次元球面の逆像を preimage 命令で求め, plot に surf の背景色を設定する命令と一緒に引き渡しています. 猶, この背景色の設定が長い為に途中で改行を入れているが, SINGULAR は行末のセミコロンが未入力の為に, 行が継続中であると判断して, ピリオド. を出しています. この様に, Steiner のローマ曲面は, 球面の写像 f による逆像としても得られます.

では, 今度は SINGULAR の方から, 曲線や曲面の新しい描き方を提案しましょう. 曲線や曲面が, 助変数表示されている場合, surf で絵を描く事は出来ません. この場合は, Maxima の描画函数の方が上手く描くかもしれません.

ところが, SINGULAR の eliminate 命令を併用して描く事が可能な場合があります. SINGULAR の eliminate 命令は不要なイデアルの変数を削除する命令です. 以下に, eliminate 命令を使って図 14.9 に示すデカルトの葉状曲線を描く例を示しましょう.

```
> ring r1=0,(x,y,t),dp;
> ring r2=0,(x,y),dp;
> map f=r1,x,y,0;
> setring r1;
> ideal i1=(1+t^3)*x-3*t,(1+t^3)*y-3*t^2;
> ideal i2=eliminate(i1,t);
> poly c2=i2[1];
```

```

> c2;
x3+y3-3xy
> setring r2;
> plot(f(c2));

```

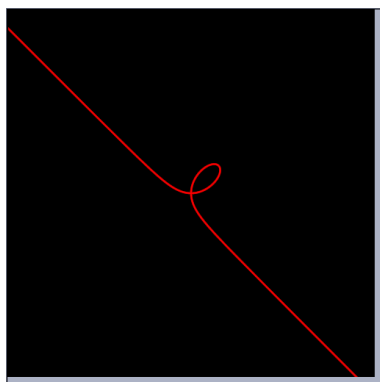


図 14.9: デカルトの葉状曲線

この例では、助変数を含めた環 $r1$ とそれを除いた環 $r2$, 更に、環 $r1$ から環 $r2$ への写像 f を定義し、次に、 $r1$ 上でイデアル $i1$ を定義します。この葉状曲線は助変数 t を用いて $x = \frac{3t}{1+t^3}, y = \frac{3t^2}{1+t^3}$ で表現されていますが、基礎環 $r1$ が $\mathbb{Q}[x, y, t]$ の為、イデアル $i1$ の定義で分子をかけた形にしています。次の `eliminate` 命令では、助変数 t を削ったイデアル $i1$ の表現を求めています。この `eliminate` 命令でイデアルが返され、それをイデアル $i2$ と定義しています。次に、イデアルは生成元を $c1$ に示す様に取り出せます。それから基礎環を `setring` 命令で環 $r2$ にし、写像 f による $c1$ の像を `plot` 命令で描いた結果が図 14.9 になります。ここで環 $r1$ ではなく環 $r2$ 上で曲線を描いた理由は単純に `surf.lib` で、環の変数の総数で曲線と曲面を判別する為、環 $r1$ の変数が (x, y, t) の三変数の為、曲面が描かれるからです。

14.10 Maxima で終結式を使ってみよう

ここで話を SINGULAR から Maxima に戻します。先程の例では、パラメータ表示されたデカルトの葉状曲線を SINGULAR の `eliminate` 命令を用いて、 x, y の多項式に変換しました。これと同じ事を Maxima でするにはどうすればよいでしょうか？

Maxima で同じ事を行う為に eliminate 函数を用いれば良いのですが、これでは幾ら何でも呆気ないので、eliminate 函数で利用されている多項式の終結式を使ってみましょう。Maxima では終結式の計算に resultant 函数が使えます。又、bezout 函数を使って行列を求め、その行列の determinant を計算する方法もあります。ここでは色々と試してみしましょう。

—— 終結式を求める resultant 函数 ——

resultant($\langle$ 多項式₁ $\rangle$, $\langle$ 多項式₂ $\rangle$, $\langle$ 変数 $\rangle$)

resultant 函数は引数として、二つの多項式と一つの変数を必要とします。ここで指定する変数は、二つの多項式から消去したい変数になります。この終結式は、二つの多項式を指定した変数の多項式として並び換えを行います。

Maxima では resultant 函数は bezout 函数と determinant 函数の合成で表現可能です。ここではデカルトの葉状曲線を用いて順番に作業を追ってみましょう。

```
(%i1) p1:(1+t^3)*x-3*t;
(%o1) (t^3 + 1) x - 3 t
(%i2) p2:(1+t^3)*y-3*t^2;
(%o2) (t^3 + 1) y - 3 t^2
(%i3) expand(p1);
(%o3) t^3 x + x - 3 t
(%i4) expand(p2);
(%o4) t^3 y + y - 3 t^2
(%i5) m1:matrix([x,0,-3,x,0,0],
                [0,x,0,-3,x,0],
                [0,0,x,0,-3,x],
                [y,-3,0,y,0,0],
                [0,y,-3,0,y,0],
                [0,0,y,-3,0,y]);
                [ x  0  - 3  x  0  0 ]
                [
                [ 0  x  0  - 3  x  0 ]
```

```

[
[ 0  0  x  0 - 3  x ]
(%o5) [
[ y - 3  0  y  0  0 ]
[
[ 0  y - 3  0  y  0 ]
[
[ 0  0  y - 3  0  y ]
(%i6) expand(determinant(m1));
3 3
(%o6) - 27 y + 81 x y - 27 x

```

この例では、最初に二つの多項式多項式 $p1 = (1+t^3)x - 3t$ と $p2 = (1+t^3)y - 3t^2$ を定義しています。ここで、多項式 $p1$ と多項式 $p2$ を展開すると、各々、 $p1 = xt^3 + 0t^2 - 3t + x$ と $p1 = yt^3 - 3t^2 + 0t + y$ となります。多項式 $p1$ と $p2$ の終結式は、この t の多項式と看做した場合の係数を並べたもので得られます。両方の多項式の次数が 3 の為、構築する行列 $m1$ は次の 6 次の正方行列になります。

$$m1 = \begin{bmatrix} x & 0 & -3 & x & 0 & 0 \\ 0 & x & 0 & -3 & x & 0 \\ 0 & 0 & x & 0 & -3 & x \\ y & -3 & 0 & y & 0 & 0 \\ 0 & y & -3 & 0 & y & 0 \\ 0 & 0 & y & -3 & 0 & y \end{bmatrix}$$

この行列の行列式を計算したものが終結式になります。

ここで、この終結式には面白い性質があります。それは終結式と二つの多項式の解の関係を示すものです。

一般的には、変数 x を主変数とする多項式 f と g が以下で表現されるものとします。

$$f = \sum_{i=0}^m a_i x^i$$

$$g = \sum_{i=0}^n b_i x^i$$

この二つの多項式の終結式は次で表現されます。

$$\text{resultant}(f, g, x) = \det \begin{pmatrix} a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 \\ b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 \end{pmatrix}$$

ここで、多項式 f と g の解を各々 α_i, β_j とすると、多項式 f と g の終結式は解 α_i, β_j を用いて以下の式に等しくなります。

$$\text{resultant}(f, g, x) = a_m^n b_n^m \prod_{1 \leq i \leq m, 1 \leq j \leq n} (\alpha_i - \beta_j)$$

この事は、多項式 f と g が共通の解を持てば終結式が常に 0 になり、同時に、終結式が 0 になるのは多項式 f と g が共通の解を持つ時に限る事を意味します。ここでは、零点集合を表現する多項式を計算するのが目的なので、終結式の零点集合と、多項式 p_1 と p_2 が同時に 0 になる零点集合は一致しなければなりません。この性質を利用して、終結式を使って助変数を排除しているのです。

猶、Maxima ではこの行列を効率良い行列で書き出す bezout 函数もあります。この函数で、上記の多項式 p_1 と p_2 の係数から構成される行列と、determinant 函数による結果を示しておきます。

```
(%i7) bezout(p1,p2,t);
      [ 0    3 x  - 3 y ]
      [                ]
(%o7)  [ 3 y   - 9    3 x ]
      [                ]
      [ - 3 x  3 y    0  ]
(%i8) expand(determinant(%));
      3                3
(%o8)  - 27 y  + 81 x y - 27 x
```

この計算を resultant 命令で一度に実行したものを以下に示します。

```
(%i5) p1:(1+t^3)*x-3*t;
```

```
(%o5)          (t + 1) x - 3 t
(%i6) p2:(1+t^3)*y-3*t^2;
              3          2
(%o6)          (t + 1) y - 3 t
(%i7) i1:resultant(p1,p2,t);
              3          3
(%o7)          - 27 (y - 3 x y + x )
```

これで, SINGULAR と定数項を除いて等しい多項式が得られました. ここで定数項は無視しても構いません. 実際, a を定数, $f(x_1, \dots, x_n)$ を多項式とする時, $f(x_1, \dots, x_n)$ の零点集合と, $af(x_1, \dots, x_n)$ の零点集合は一致するからです.

一寸, 脇道に逸れますが, 多項式 f の零点集合 $V(f)$ と多項式 $g(x_1, \dots, x_n)$ と多項式 $f(x_1, \dots, x_n)$ の積の零点集合 $V(fg)$ の関係を思い出して下さい. $V(fg)$ は多項式 f と g の零点の両方を含み, $V(f) \cup V(g)$ になります. 更に, f で割切れる多項式 h の零点集合 $V(h)$ に対しては, $V(h) \supset V(f)$ が成立します.

ここで, 多項式 f で生成されるイデアル (f) を考えると, $V(f)$ がイデアル (f) に含まれる多項式の零点集合の中で最小の集合になります. このイデアルで考えてしまえば, 多項式が定数倍である事は大きな問題となりません. ですから, 多項式の零点集合を考える場合, そのイデアルを考えるのが最も手軽な手段になります.

次に, 応用で猿の腰掛と呼ばれる曲面を表示してみましょう. この曲面は助変数表示では以下の関係式を満たすものです.

猿の腰掛の助変数表示

$$\begin{aligned} x - u &= 0 \\ y - v &= 0 \\ z - u^3 + 3uv^2 &= 0 \end{aligned}$$

本当は, 最後の式の変数 u, v を x と y で各々置換えてしまえば済む話なのですが, ここでは, 終結式を使って猿の腰掛の変数 x, y, z の式に変換してみましょう. ここで, 注意する事は, 迂闊に, $\text{resultant}(x - u, y - v, u)$ の様に, 片方の式にしか存在しない変数を使ってはいけません. 無意味な式が返って来るだけです. この例では, 変数 u と v が含まれているのは, $z - u^3 + 3uv^2 = 0$ のみなので, resultant はこの多項式を中心に, 例えば, $x - u$ から開始し, $y - v$ で終える手順になります.

```
(%i1) p1:x-u;
(%o1)          x - u
```

```
(%i2) p2:y-v;
(%o2)          y - v
(%i3) p3:z-u^3+3*u*v^2;
(%o3)          2      3
          z + 3 u v - u
(%i4) a1:resultant(p1,p3,u);
(%o4)          3      2
        - z + x - 3 v x
(%i5) a2:resultant(a1,p2,v);
(%o5)          2      3
        - z - 3 x y + x
(%i6) surfplot(a2);
```

これで多項式として、 $-z - 3xy^2 + x^3$ が得られました。このグラフを surf で描いたものが図 14.10 に示す、尻尾の生えた猿に丁度良さそうな腰掛みたいな曲面が描かれます。

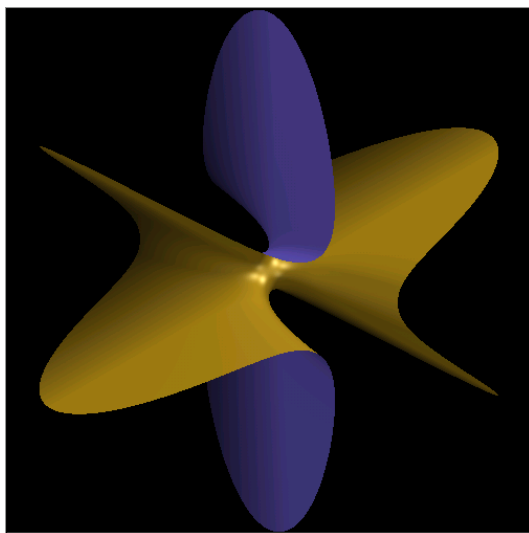


図 14.10: 猿の腰掛

猶,Maxima には eliminate 関数が存在するので, 実際は,
 $\text{eliminate}([x - u, y - v, z - u^3 + 3 * u * v^2], [u, v]);$ で十分です.

試しに、動作を確認してみましょう。

```
(%i13) eliminate([x-u,y-v,z-u^3+3*u*v^2],[u,v]);
                                     2    3
(%o13)          [- z - 3 x y  + x ]
(%i14) %[1];
                                     2    3
(%o14)          - z - 3 x y  + x
```

長々とした計算を行っていましたが,eliminate 函数を使えば,簡単に出来る事でした. ご苦労さまでした.

14.11 デモファイルでも書いてみよう

さて,surfplot.mc で色々遊んでみたので,デモ用のファイルを作ってみましょう. これはバッチファイルでも良いのですが,Maxima には函数やパッケージのデモ用に demo 函数があります. このデモ用のファイルの内容は batch ファイルと基本的に同じもので,要するに,Maxima が実行する内容を並べておけば良いのです.

但し,demo 函数は Maxima の一行を実行するとプロンプト_を出して一旦停止します. ここで,セミコロン; や Enter キーを押せば,次の行の処理に移ります. その為,入力行に対する処理がどの様に実行されるかが判り易くなっています.

このデモファイルの修飾子は dm,dm1,dm2,dm3,dmt,dem が用いられます. 更に,デモファイルが大域変数 file_search_path に記述されたディレクトリ上にある場合,ファイル名の修飾子を外した名称だけを引数にする事が可能です. 即ち,デモファイル名が surfplot.dem の場合, `demo(surfplot);` の様に入力するだけでデモが実行される事を意味します.

但し,大域変数 file_search_path に登録されていないディレクトリ上にデモファイルがある場合は, `demo("Desktop/surfplot.dem");` の様に,カレントディレクトリ上に無いファイルに対してはパスも含めて二重引用符でファイル名を括る必要があります. 猶,大域変数 file_search_path に強引に自分のデモファイルを登録したディレクトリを登録する事も可能ですが,Maxima のバージョンによって文字列の扱いが異なる為,注意が必要になります.

先ず,Maxima-5.13.0 以前では Maxima の文字列と LISP の文字列は別のデータ型になります. 従って,file_search_path には Maxima の文字列を LISP の文字列に変換して追加する必要があります. その為,to_lisp で \$file_search_path に強引にパスを追加す

る方法もあります. 但し,Maxima-5.14.0 では Maxima の文字列と LISP の文字列が共通になる為, この変換は不要になり,Maxima 上で追加処理が済ませられます.

猶,追加する LISP の文字列は" $\langle PATH \rangle/###.\{dem,dmt\}$ " の形になります. 猶, $\langle PATH \rangle$ はディレクトリ, $\{dem,dmt\}$ がファイルの修飾子を指定するものです.

さて,dem ファイルの内容は Decartes の葉状曲線, レムニスケート, お遊びのアヒル, を入れておきましょう. 但し, それだけでは寂しいので, ローマン曲面の断面を描いたものや Bath Diec も入れてみましょう.

ここでローマン曲面の断面を描く為に細工を行います. それは surf に曲線の透明度を最初に指定し, それからローマン曲面に交差する平面と, その平面との交わりの色を指定する事, そして,surf でそれらを描く surf のスクリプトファイルを生成する事です. つまり, 最初に標準的なローマン曲面を表示させます. すると,surfplot は surf.tmp ファイルを描画の為に生成します. そこで, 透明度や平面と断面の指定を surf.tmp に書き込むのです. ここでは簡単に済ませる為に,system 関数を使って,UNIX の mv 命令と shell の echo 命令のみを用いて実現させています.

```
/* decartes curve */
delta:-1/4;
poly:x^3+y^3-3*x*y -delta;
surfplot(poly);

/* lemniscate(1)*/
poly:(x^2*(9-x^2)-4*y^2);
surfplot(poly);
/* lemniscate(1)*/
poly:(x^2*(9-x^2)-4*y^2)*x*y;
surfplot(poly);

/* Ducky */
poly:(x^2+y^2-9)*((x-2)^2+(y-1)^2-1)*
((x+2)^2+(y-1)^2-1)*(x^2/9+2*(y+1)^2-1);
surfplot(poly);

put(surf,0.6,rot_x);
put(surf,0.6,rot_y);
put(surf,0.6,rot_x);
```

```
put(surf,0.6,rot_z);
put(surf,0.06,scale_x);
put(surf,0.06,scale_y);
put(surf,0.06,scale_z);
poly:x^2*y^2+x^2*z^2+y^2*z^2-x*y*z;
surfplot(poly);
/* 透明度の指定
一旦,surf.tmpx ファイルに透明度の指定等を書き込んでおきます.
*/
system("echo \"transparence=50;
illumination=ambient_light+diffuse_light+
reflected_light+transmitted_light;\" >surf.tmpx");
/* surf.tmp の内容を surf.tmpx の末尾に追加します */
system("cat surf.tmp>>surf.tmpx");
/* surf.tmpx を surf.tmp に名前を変えます */
system("mv surf.tmpx surf.tmp");
/* 平面 x+y+z-0.005=0 による断面を指定 */
system("echo \"plane=x+y+z-0.05;\">>surf.tmp");
/* 以降,断面の色を指定 */
system("echo \"curve_red=255;\">>surf.tmp");
system("echo \"curve_green=0;\">>surf.tmp");
system("echo \"curve_blue=0;\">>surf.tmp");
/* 断面を描く事を指定 */
system("echo \"cut_with_plane;\">>surf.tmp");
/* 平面 x+y+z+0.05=0 による断面を指定 */
system("echo \"plane=x+y+z+0.05;\">>surf.tmp");
/* 以降,断面の色を指定 */
system("echo \"curve_red=0;\">>surf.tmp");
system("echo \"curve_green=255;\">>surf.tmp");
system("echo \"curve_blue=0;\">>surf.tmp");
/* 断面を描く事を指定 */
system("echo \"cut_with_plane;\">>surf.tmp");
/* surf に surf.tmp 内容を読み込んで実行 */
system("surf -x surf.tmp>surf.log&");
```

```

/* Barth Diec */
/* set rot_x,y,z and scale_x,y,z */
put(surf,0.6,rot_x);
put(surf,0.2,rot_y);
put(surf,0.6,rot_z);
put(surf,0.3,scale_x);
put(surf,0.3,scale_y);
put(surf,0.3,scale_z);
/* set tau */
tau:(1+sqrt(5))/2;
/* polynomial */
poly:8*(x^2-tau^4*y^2)*(y^2-tau^4*z^2)*(z^2-tau^4*x^2)*
(x^4+y^4+z^4-2*(x^2*y^2+y^2*z^2+z^2*x^2))+
(3+5*tau)*(x^2+y^2+z^2-1)^2*(x^2+y^2+z^2-(2-tau))^2;
surfplot(poly);

/* Barth Diec */
/* set rot_x,y,z and scale_x,y,z */
put(surf,0.6,rot_x);
put(surf,0.2,rot_y);
put(surf,0.6,rot_z);
put(surf,0.3,scale_x);
put(surf,0.3,scale_y);
put(surf,0.3,scale_z);
/* set tau */
tau:float((1+sqrt(5))/2);
/* polynomial */
poly:8*(x^2-tau^4*y^2)*(y^2-tau^4*z^2)*(z^2-tau^4*x^2)*
(x^4+y^4+z^4-2*(x^2*y^2+y^2*z^2+z^2*x^2))+
(3+5*tau)*(x^2+y^2+z^2-1)^2*(x^2+y^2+z^2-(2-tau))^2;
surfplot(poly);

```

さて、これでデモファイルが出来ました。では実行してみましょう。

その前に、ここではカレントディレクトリ上に,surfplot.mc と surfplot.dem が置かれて

いるものとします.そして,注意になりますが,このデモファイルには surfplot.mc ファイルの読み込みを記述していないので,デモを実行する前に, `load("surfplot.mc");` で surfplot.mc の読み込みを実行し,それから, `demo("surfplot.dem");` を実行しましょう.

実行の様子を頭の数行だけ示しておきます.

```
(%i2) demo("surfplot.dem");
```

```
batching /home/yokota/Desktop/surfplot.dem
```

```
At the _ prompt, type ';' followed by enter to get next demo
```

```

                                - 1
(%i3)                        delta : ---
                                4

-
                                3    3
(%i4)                poly : - delta - 3 x y + y  + x

```

Enter キーを押して行くとグラフが出たりすれば成功です.

14.12 例題ファイルもついでに…

さて,demo 関数用のデモファイルを作ったので,今度は example 関数向けの例題ファイルを作ってみましょう.

例題ファイルの在処は,LISP の変数*maxima-demodir*に束縛された値を演算子:lisp で調べると判ります.又,ファイル名は Maxima の大域変数 manual_]demo に割り当てられています.

```

(%i54) :lisp *maxima-demodir*
/usr/local/share/maxima/5.13.0/demo
(%i54) manual_demo;
(%o54)                                manual.demo

```

以上から,この Maxima の例題ファイルは, /usr/local/share/maxima/5.13.0/demo にある manual.demo ファイルである事が判ります.

ここで件の manual.demo ファイルの内容を覗いてみましょう.

```

/* This file is to be run by the EXAMPLE command, and may not
   otherwise work.

   The following are either acceptable lines to maxima, or they are
   two successive '&' characters immediately following a '$' or ';'
   and then followed by the name of the section of examples, and then followed by
   a sequence of maxima forms.

*/
&&
additive declare(f,additive);
f(2*a+3*b);&&
algsys f1:2*x*(1-11)-2*(x-1)*12$
f2:12-11$
f3:11*(1-x**2-y)$
f4:12*(y-(x-1)**2)$
algsys([f1,f2,f3,f4],[x,y,11,12]);
f1:x**2-y**2$
f2:x**2-x+2*y**2-y-1$
algsys([f1,f2],[x,y]);&&

... 中略 ...

zeroequiv zeroequiv(sin(2*x)-2*sin(x)*cos(x),x);
zeroequiv(%e^x+x,x);
zeroequiv(log(a*b)-log(a)-log(b),a);

```

ここで、例題ファイルの注釈は/* */で括る事が判りますね。書式は Maxima のバッチファイルに似ていますが、末尾に&&がある行や、`additive declare(f,additive);` の様にラベルと Maxima の文が記述されている箇所がありますね。

例題ファイルの書式は、ラベルを置いた場合、空白を空けてから Maxima の入力文を記述し、一通り終了すると、末尾に&&を置く書式になっているのです。ここでラベルは自由です。だからといって出鱈目なもの、例えば、`surfplot` の様な函数名ではなく、”三毛猫大王”の様なラベルにすると、内容が判らず後で苦勞する事になるでしょう。さて、`example` 函数は与えられた引数に合致するラベルを検出し、そこから&&が検出される迄の行を処理する仕組になっています。

従って、以下の様に改造する事が可能です。

```

... 略 ...

zeroequiv(%e^x+x,x);
zeroequiv(log(a*b)-log(a)-log(b),a);&&
/* 三毛猫大王
これでどうかな？ */&&

```

```
surfplot print("Decartes!!");
surfplot(x^3+y^3-3*x*y+1/4);
```

この様に改造した例題ファイルを所定の位置に置くか,*maxima-demodir*の値の変更を予め行ない,肝心の surfplot.mc を読み込んでいれば何時でも `example(surfplot);` を実行すると例題が処理されます.

```
(%i56) example(surfplot);
```

```
(%i57) print("Decartes!!")
```

```
Decartes!!
```

```
(%i58) surfplot(1/4-3*x*y+y^3+x^3)
```

```
'rat' replaced 9.999999999999999E-11 by 1/10000000000 = 9.999999999999999E-11
```

```
Surf is now drawing (4*y^3-12*x*y+4*x^3+1)/4 . Please wait ....
```

```
(%o58) 0
```

```
(%i59) false
```

```
(%o59) done
```

```
(%i60)
```

第15章 Octaveで遊ぶ話

この章では,Maximaではなく GNU の MATLAB クローンである Octave を使った数値行列の処理,特に並びの照合等の MATLAB 風の言語に特有の処理について解説を行います.

猶,MATLAB は数値行列の処理を目的とした The Mathworks,Inc. の非常に高機能で,高価な製品です.Octave はこの MATLAB と非常に互換性が高い GNU のアプリケーションです.

15.1 MATLAB クローン色々

この本は Maxima の解説書ですが,ここで何故 MATLAB クローンの話となる理由を説明しておきましょう. 先ず,数式処理システムの多くが任意精度による数値計算が可能であり,行列も比較的扱い易いという長所を持っていますが,残念な事に処理速度自体は速いものではなく,特に行列の処理では遅さが目立ちます. 更に,Maxima は LISP で記述されている為,Maxima の処理言語でプログラムを記述すると,その実行は,一旦,LISP に変換されてから処理を行う為に一層不利になります. この辺を回避する為には,LISP で直接処理プログラムを書く方法もありますが,数値計算だけで済むのであれば,行列データを扱う事を主な目的とした MATLAB の様なツールを使うのも一手です. その上,この方が行列の処理が楽でもあります.

この MATLAB に似たものには次のものがあります.

- Octave
- SciLab
- yorick

この中で Octave が最も MATLAB と互換性が非常に高く,MATLAB クローンと呼んでも差し支え無い程です. 但し,グラフ出力では Octave は機能的に MATLAB とかなり劣ります. それに対し SciLab は MATLAB や Octave と似ている点も多くあります

が、その一方で違いも多くなりますが、機能は MATLAB と比べて見劣りしない程になります。これが yorick になると、行列の扱いには MATLAB に似た面もありますが、yorick は対話処理の可能な C と言える程で、MATLAB とは全く独自のソフトウェアになります。但し、その処理は比較的高速で、その上、アニメーションが簡単に作製出来るといった長所もあります。

KNOPPIX/Math には Octave と yorick が収録されています。yorick は魅力あるソフトウェアなのですが、MATLAB との違いが大きい為、ここでは触れません。但し、基本的な行列の処理等で似た面も多いので、C に慣れた方は使ってみると楽しいかと思います。そこで、この章では簡単に Octave を用いた行列の計算やグラフ表示について述べます。

猶、ここで述べる事は MATLAB でもほぼ同様の事が行えます。MATLAB と異なる個所についてはその旨を明記します。但し、ここで比較対象の MATLAB は古い MATLAB 5.2 の為、最新のものの場合は異っているかもしれない事を断っておきます。

15.2 Octave の基本

Octave のオンラインヘルプの利用方法は単純に `help < 命令 >` と入力します。Octave では、更に、`help -i < 事項 >` で関連事項を調べられます。

例えば、常微分方程式を解きたいがどの様な命令が使えるのか分からない場合、`help -i ode` と入力すれば Octave のオンラインマニュアルから、常微分方程式を扱う命令に関する事項が表示されます。

Octave の help は引数が函数名であれば、less を用いて M-file、即ち、拡張子が .m の Octave の函数が記述されたファイルの先頭の注釈部分を表示し、`help -i` の場合は、texinfo を使って info ファイルを表示します。その為、Octave を利用する場合は、これらの GNU のアプリケーションを揃えておく必要があります。但し、KNOPPIX で利用する場合は全て揃っているので必要なパッケージを自分で揃える必要はありません。

もし、MATLAB の入門書があれば、試しに MATLAB の命令を help で打ち込んで見ると面白いでしょう。共通するものが多い一方で、意外な所で違っていたりします。

15.3 入力

Octave の四則演算は C と同様の式で表現されます。Octave は Maxima と違い、入力行末にセミコロン (;) をつけておくと計算結果を表示しません。これは大きな行列データ処理や反復回数が多い処理で必須になります。


```
octave:1> 1+1;
octave:2> 1+1
ans = 2
```

代入は=で行います. Maxima を使っていると,Maxima の割当の演算子と混同し易いので注意しましょう,

```
a = 1.2000
octave:4> a
a = 1.2000
octave:5> c=pi
c = 3.1416
octave:6> s=a^2*pi
s = 4.5239
```

15.4 数学定数

Octave や MATLAB には円周率 π , 自然底 e 及び純虚数 i 等の数学定数が予め用意されていますが, その値は立ち上げ時に初期化ファイルを使って設定しています. MATLAB と Octave では, それらの変数名 (実際は組込関数) は `pi`, `e`, `i` 等となっていますが, これらの変数値は利用者が簡単に変更出来てしまう代物です. 例えば, `for` 文 を使う際に, 何気なく, 定数 `i` を変数として利用すると, 以下に示す様に純虚数 i が設定された定数 `i` が呆気無く書換えられてしまいます.

```
octave:1> pi
pi = 3.1416
octave:2> i
i = 0 + 1i
octave:3> e
e = 2.7183
octave:4> i*i
ans = -1
octave:5> a=1+i;
octave:6> b=1-i;a*b
ans = 2
octave:7>
```

この様に,MATLAB や Octave では定数 i, e, π は予約語として特別に保護されていないのでうっかり利用しない様に注意しなければなりません. 猶, 利用者が定義した変数は `who` 命令を使うと表示されます. 更に,`type` 命令で変数が利用者が定義したものか, 組込のものかどうか分ります.

以下に定数 i と π を書換えている例を示します.

```
octave:1> who
octave:2> type i
i is a builtin function
octave:3> i=2
i = 2
octave:4> i*i
ans = 4
octave:5> type i
i is a user-defined variable
2
octave:6> type pi
pi is a builtin function
octave:7> pi=10
pi = 10
octave:8> i=sqrt(-1)
i = 0 + 1i
octave:9> pi=acos(-1)
pi = 3.1416
octave:10> who

*** local user variables:

i    pi

octave:11> type pi
pi is a user-defined variable
3.1416
octave:12> for i=1:10
> i*2;
```

```
> end;
octave:13> i
i = 10

octave:14> type i
i is a user-defined variable
10
```

この例では、一番最初に `who` 命令を実行した時点では何も表示されませんでした。が、`i` や `pi` に値を入れてしまうと利用者定義の変数になってしまうので、次に、`who` を実行すると利用者定義の変数として `i` と `pi` が表示されます。実際、変数の型が調べられる `type` 命令を使うと、これらの定数は書換えを行った為に利用者定義変数になっている事が分ります。次の `for` の例では、`i` が最終的に 10 で置換えられている事を示しています。ここで、`i` は添字で、`for` 文等の反復処理で迂濶に利用する可能性が高い為、変数の末尾に 1,2,3 等の数字を付けたり、`ii` の様に変数を全て二文字以上にする等の工夫をする方が安全です。

猶、便利な定数に `eps` があります。この定数は微少値 (Octave の場合は `eps=2.2204e-16`、但し、計算機環境によっては異なるかもしれません) を表現しており、零による割算を避ける為や、`while` 文等の反復処理で `eps` の何倍かに誤差が収まった場合に反復処理を停止する様に設定する等と何かと便利な定数です。

15.4.1 Octave の演算

MATLAB, Octave で演算は C 等のプログラム言語と変わらない形で入力を行います。先ず、Octave の演算の一覧を示しましょう。猶、以下の一覧では、大文字を行列、小文字をスカラーとしています。

Octave の演算

演算	例	概要
和 (+)	$A+B, a+b, A+b$	同じ大きさの行列, 行列とスカラーの和
差 (-)	$A-B, a-b, A-b$	同じ大きさの行列, 行列とスカラーの差
積 (*)	$A*B, a*b, a*B$	行列同士の行列積, 行列とスカラーの積
冪 (^)	a^b, A^b, a^B	行列の冪
商 (/)	$A/B, a/b, A/b$	行列同士の 場合, 演算子右側を逆行列にして行列積を計算. $A/B=A*B^{(-1)}$ と同値
商 (\)	$A \setminus B, a \setminus b, a \setminus B$	行列同士の 場合, 演算子左側を逆行列にして行列積を計算. $A \setminus B=A^{(-1)}*B$ と同値
積 (.*)	$A.*B, a.*b, a.*B$	同じ大きさの行列の成分毎の積, 行列とスカラーの積
冪 (.^)	$A.^B, a.^b, A.^b, a.^B$	行列の冪
商 ./)	$A./B, a./b, a./B$	同じ大きさの行列の成分毎の商 (/), 行列とスカラーの商
商 (. \)	$A.\setminus B, a.\setminus b, a.\setminus B, A.\setminus b$	同じ大きさの行列の成分毎の商 (\), 行列とスカラーの商
転置 (')	A'	行列の転置

では, 実際に動作を確認してみましょう. 行列 A と行列 B は以下のものとします.

```
octave:1> A=[1,2;3,1]
```

```
A =
```

```
1 2
3 1
```

```
octave:2> B=[5,1;2,1]
```

```
B =
```

```
5 1
2 1
```

和 (+), 差 (-)

```
octave:3> A+1
ans =
```

```
2 3
4 2
```

```
octave:4> A-1
ans =
```

```
0 1
2 0
```

```
octave:5> A+B
ans =
```

```
6 3
5 2
```

演算子が和+と差-の場合, 行列同士の演算の場合は両者が同じ大きさでなければなりません, 行列とスカラーの場合は可能で, その場合, 各成分毎の和と差となります.

積 (*) と積 (.*)

```
octave:9> A*B
ans =
```

```
9 3
17 4
```

```
octave:10> A.*B
ans =
```

```

5 2
6 1

```

積には二種類あり, 演算子`*`と演算子`.*`は二つの被演算子の内, どちらか一方がスカラーであれば結果は同じになりますが, 両方が行列の場合は意味が違います. 先ず, 演算子`*`は通常の行列の積となります. これに対し, 演算子`.*`は成分毎の積となります. 即ち, 行列 $A = (a_{ij})$ と $B = (b_{ij})$ に対し, $A * B = (a_{ij} * b_{ij})$ となります. この演算子`.*`の様に, 先頭に`.`が付く演算子は全て成分毎の演算を意味します.

冪 (`^`と`.^`)

```

octave:29> A.^2
ans =

```

```

1 4
9 1

```

```

octave:30> 2^A
ans =

```

```

5.6453 4.3104
6.4656 5.6453

```

通常の冪`^`のみ行列同士では利用出来ません.

商 (`/`), 商 (`\`)

```

octave:15> A/B
ans =

```

```

-1.00000 3.00000
0.33333 0.66667

```

```
octave:16> A\B
```

```
ans =
```

```
-0.20000    0.20000  
 2.60000    0.40000
```

```
octave:17> A*B^(-1)
```

```
ans =
```

```
-1.00000    3.00000  
 0.33333    0.66667
```

```
octave:18> A^(-1)*B
```

```
ans =
```

```
-0.20000    0.20000  
 2.60000    0.40000
```

```
octave:19> 2\B
```

```
ans =
```

```
 2.50000    0.50000  
 1.00000    0.50000
```

```
octave:20> A/2
```

```
ans =
```

```
 0.50000    1.00000  
 1.50000    0.50000
```

同じ大きさの行列に対しては、 $A/B=A*B^{-1}$ 、 $A\backslash B=A^{-1}*B$ と各々同値になります。行列とスカラーの場合、スカラーを分母にする計算は可能ですが、行列を分母にする計算、例えば、 $B \setminus 2$ や $2/A$ の計算はエラーになります。

Octave の様なインタプリタ型の言語では毎度の事です、C と比べて処理速度で劣ります。但し、上記の行列の演算のみであれば、そんなに気にする程ではありません。しか

し、for 文による反復処理を多用すれば顕著になります。

更に、四則演算も対処方法を誤れば、計算時間を無駄に消費しかねません。ここで、和+、差-、積*、商/と冪.^の順番が処理速度の速度順になります。殊に、冪乗.^は他の演算と比べて極端に処理が遅い特徴があります。例えば、 $a.^2$ や $a^{0.5}$ は $a.*a$ と $\text{sqrt}(a)$ の方が圧倒的に高速になります。乱数行列を生成する rand 命令を用いて 1000 行 1 列の行列を生成し、 $a.*a$ 、 $a.^2$ で成分の二乗を計算する例を示しましょう。

ここで、返されている数値が処理時間です。この計算は私の計算機 (Pentium III 1GHz の SuSE 9.3 環境の PC) で実行しています。但し、この数値は絶対的なものではなく、参考の為に示しています。

```
octave:9> t1=time;a.*a;t2=time;t2-t1
ans = 0.00011706
octave:10> t1=time;a.^2;t2=time;t2-t1
ans = 0.00026608
octave:11> t1=time;exp(2*log(a));t2=time;t2-t1
ans = 0.00070596
octave:12> t1=time;a.*a.*a.*a.*a.*a.*a.*a.*a.*a.*a.*a;t2=time;t2-t1
ans = 0.00032997
octave:13> t1=time;a.^12;t2=time;t2-t1
ans = 0.00032210
octave:14> t1=time;exp(12*log(a));t2=time;t2-t1
ans = 0.00071812
```

この様に二乗の計算でさえも 2 倍近く通常の積の方が冪よりも速い事が判ります。又、指数関数と対数関数の組み合わせのものと比べると通常の積は 20 倍近い事が判ります。但し、冪の指数が増えても冪や対数関数を用いたものの差は、処理時間に大きな差が生じないものの、積に関しては、回数が単純に増加する為に、計算時間に大きな違いが出ており、冪よりも遅くなる事が判るかと思います。

では、冪乗を適当に区切って計算した場合はどうなるでしょうか。今度は 9 乗の計算で 3 個単位で纏めて計算させたものも含めて比較してみましょう。

```
octave:15> t1=time;a.*a.*a.*a.*a.*a.*a.*a.*a;t2=time;t2-t1
ans = 0.00025606
octave:16> t1=time;b=a.*a.*a;b=b.*b.*b;t2=time;t2-t1
ans = 0.00020099
octave:17> t1=time;b=a.^9;t2=time;t2-t1
```



```
ans = 0.00032377
octave:18> t1=time;exp(9*log(a));t2=time;t2-t1
ans = 0.00072813
```

この結果では冪と安易な積を比較すると安易な積の方が速いものの、冪との差は随分と小さくなっています。これに対し、三乗で纏めた積は安易な積よりも 1.5 倍程度高速になっています。この様に Octave では工夫次第で積による処理の方が速くなる事が判ります。

以上から、冪乗も小分けして積で表現した方が速く、指数が大きくない冪乗と log 関数は可能な限り避けた方が賢明だと言えます。

ここで冪乗に関しては指数の大きさによる面もある為、注意が必要になりますが、一般的に、全体の計算量を削減したものの方が良好な結果を得易いと言えます。この様に全体の計算量を考慮せずに安易な処理を行うと、逆に悪化する可能性が高くなります。この様に積に関しても工夫の余地がある事が分りましたが、MATLAB 系の処理言語では安易に for 文を用いるだけで容易に処理速度を低下させる事が出来てしまいます。これは MATLAB 言語等では、配列から指定された行列の成分を取り出してコピーし、そのコピーを用いて計算した結果を再び配列に戻す事を行っている為です。これに対し、行列やベクトルの直接演算を行う場合は、標準的な行列演算ライブラリを直接利用します。その為、配列の取り出しやコピーの様な余計な手間が不要となるので、下手にプログラムを組むよりは高速に処理が行えます。

15.4.2 Octave の行列

MATLAB クローンで扱うデータは基本的に行列です。言語仕様自体が効率良く行列を扱える様に工夫されています。猶、添字は Fortran 同様に 1 から開始します。ベクトルや行列の書式は SciLab, MATLAB, Octave でも同じ書式で行います。行列の入力例を以下に示しておきましょう。

```
ans =

    1    2    3
    4    5    6

octave:20> [1,2,3;4,5,6]
ans =
```

```

1  4
2  5
3  6

```

```

octave:21> [1,2,3]
ans =

```

```

1  2  3

```

```

octave:22> [1,2,3]'
ans =

```

```

1
2
3

```

これらの例に示す様に octave では行列やベクトルを視覚的に表示します. 猶, 元の区切は空行かコンマ, を用い, 行の区切にはセミコロン; を用います. ここでセミコロン; に似ているコロンの意味を持ちます. このコロンの便利な利用方法は後で述べます.

次に, 入力した行列 a の i 行 j 列の成分を $a(i, j)$ で表現します. 猶, $a[i, j]$ と記述しないので注意して下さい. ここで添字の i, j は数学の行列表記と同様に 1 から開始します. ベクトルや行列の元の設定は上記の例の様に一気に設定する方法と, $a(1, 2)=1$ の様に成分を一々指定して設定する方法があります. もし, a が未定義で, 最初に $a(1, 2)=1$ と入力すると, a は 1 行 2 列の行列として生成され, 1 行 2 列の成分 $a(1, 2)$ のみが 1 で他が零となります. 更に, $a(3, 2)=10$ と続けて入力すれば, 今度は, a は 3 行 2 列の行列となり, 値が代入された個所を除く部分の値は全て零です. この方法以外に行列をベクトルと看做して要素を指定する事も可能です. 具体的には, a が m 行 n 列の行列だとすれば, $a(i, j)$ は $a((j-1)*m+i)$ のベクトルの成分としても解釈出来ます. 但し, この場合は a が何等の方法で予め定義された場合に限り, a が未定義であれば, この方式で a の各成分に値を設定すると, a はそのままベクトルになります.

行列の大きさは `size` 命令で返されます. 似たものに `length` 命令があり, こちらはベクトルに対してその長さを返します. 猶, 行列に対しては, その行数を返します.

```
octave:23> a=[1 2 3;4 5 6]
```

```
a =
```

```
1 2 3
4 5 6
```

```
octave:24> a(1,2)
```

```
ans = 2
```

```
octave:25> a(4)
```

```
ans = 5
```

```
octave:26> a(3)
```

```
ans = 2
```

```
octave:27> size(a)
```

```
ans =
```

```
2 3
```

```
octave:28> length(a)
```

```
ans = 3
```

行列の一部の取出しは非常に容易に行えます。まず、ベクトルに対して i 番目の成分から j ($\geq i$) 番目の成分を持つベクトルは $a(i:j)$ で得られます。同時に、行列 a の i 行目で構成される行ベクトルと j 列目で構成される列ベクトルは各々 $a(i,:)$ と $a(:,j)$ で得られます。猶、最初に注意した様に C の配列と異なり、番号は通常の行列と同様に 1 から始まります。

次の例では定義した行列から列ベクトルや行ベクトルを取り出しています。

```
octave:31> a(4:6)
```

```
ans =
```

```
4 5 6
```

```
octave:32> a=[1:4;5:8;8:11]
```

```
a =
```

```
1  2  3  4
5  6  7  8
8  9 10 11
```

```
octave:33> b=a(2,:)
```

```
b =
```

```
5  6  7  8
```

```
octave:34> c=a(:,2)
```

```
c =
```

```
2
6
9
```

猶, MATLAB や Octave では, 上の例で示す様にコロン: を用いて行列同士の代入を簡略化する事が出来ます.

これは, 先程示した様に行列 a が与えられた場合, $a(:,1)$ が行列 a の一列目の元で構成される列ベクトルを意味し, $a(2,:)$ が行列 a の二行目の成分で構成される行ベクトルを意味します. 次の例では行列 a の第一列 $a(:,1)$ と第二行 $a(2,:)$ を取り出しています.

```
octave:35> a=rand(4,4)
```

```
a =
```

```
0.204195  0.372247  0.195707  0.529230
0.063849  0.915721  0.857846  0.630308
0.191641  0.602701  0.667216  0.162591
0.261553  0.435798  0.732046  0.561905
```

```
octave:36> a(:,1)
```

```
ans =
```

```
0.204195  
0.063849  
0.191641  
0.261553
```

```
octave:37> a(2,:)
ans =
```

```
0.063849 0.915721 0.857846 0.630308
```

この記号:を利用して、行列を列或いは行ベクトルの集合と看做して、これらの成分を持つ行列を以下の例の方法で容易に生成出来ます。

```
octave:37> a=rand(4,4)
```

```
a =
```

```
0.204195 0.372247 0.195707 0.529230  
0.063849 0.915721 0.857846 0.630308  
0.191641 0.602701 0.667216 0.162591  
0.261553 0.435798 0.732046 0.561905
```

```
octave:38> b=zeros(4,4)
```

```
b =
```

```
0 0 0 0  
0 0 0 0  
0 0 0 0  
0 0 0 0
```

```
octave:39> b(:,2)=a(:,4)
```

b =

```
0.00000  0.52923  0.00000  0.00000
0.00000  0.63031  0.00000  0.00000
0.00000  0.16259  0.00000  0.00000
0.00000  0.56191  0.00000  0.00000
```

猶、`a(:,1)` で代入を行う場合、代入する側と代入される側の`:`で指定した大きさが同じものでなければなりません。

octave:39> a

a =

```
0.204195  0.372247  0.195707  0.529230
0.063849  0.915721  0.857846  0.630308
0.191641  0.602701  0.667216  0.162591
0.261553  0.435798  0.732046  0.561905
```

octave:40> c=zeros(2,2)

c =

```
0  0
0  0
```

octave:41> c(:,2)=a(:,2)

```
error: a(i, j) = x: x must be a scalar or the number of elements in i must
error: match the number of rows in x and the number of elements in j must
error: match the number of columns in x
error: assignment failed, or no method for 'matrix = matrix'
error: evaluating assignment expression near line 41, column 7
octave:41> a(:,2)=a(:,2)
```

```
error: a(i, j) = x: x must be a scalar or the number of elements in i must
error: match the number of rows in x and the number of elements in j must
error: match the number of columns in x
error: assignment failed, or no method for 'matrix = matrix'
error: evaluating assignment expression near line 41, column 7
```

```
octave:41> a=a(:,4)
```

```
a =
```

```
0.52923
```

```
0.63031
```

```
0.16259
```

```
0.56191
```

この上の例に示す様に:で示した行列が右辺と左辺の大きさが違う場合にエラーになります. 更に, 左辺が未定義の場合に:を用いて成分を指定してもエラーになります. この場合には, 予め変数を定義しておく必要があります. 但し, 右辺と左辺の大きさが一致すれば問題はありません.

```
octave:43> d=zeros(10,2)
```

```
d =
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```

```
octave:44> a
```

```
a =
```

```
0.204195 0.372247 0.195707 0.529230
```

```
0.063849 0.915721 0.857846 0.630308
```

```
0.191641 0.602701 0.667216 0.162591
```

```
0.261553 0.435798 0.732046 0.561905
```

```
octave:45> d([7:10],1)=a(:,1)
```

```
d =
```

```
0.00000 0.00000
0.00000 0.00000
0.00000 0.00000
0.00000 0.00000
0.00000 0.00000
0.00000 0.00000
0.20419 0.00000
0.06385 0.00000
0.19164 0.00000
0.26155 0.00000
```

この様に両者の大きさが異なる場合, 両辺に:を用いて代入すればエラーになりますが, 両辺のサイズが同じになる様に添字を調整すれば, 代入を行う際に問題がありません. 又, 記号:を使って数値ベクトルの間隔を指定する事が可能です. 例えば 1:5 は 1:1:5 と同じ意味で間隔が 1 の場合, 真中の間隔指定は不要です. 猶, 1:2:6 の場合, 6 を越えない値が設定される為, 得られるリストは [1 3 5] になります.

15.5 ベクトルに対する並びの照合

MATLAB 風の処理言語では, ベクトルに対して並びの照合処理が可能です. この並びの照合を適用する事によって, 処理速度を引き起し易い for 文等の loop 文を使わずに, 見通しの良いプログラムを行う事が可能になります. この機能は数値データを扱う場合には非常に強力で, MATLAB 風言語の威力が発揮される個所でもあります. この機能を利用すれば, 与えられたベクトルから適合するものがあるかどうかを検証する事が容易に行えます. 以下の例では与えられたベクトルから 2 に等しいものがあるかを検証し, その場所を find 命令を用いて探す処理を実行しています. 猶, Octave と MATLAB では真が 1, 偽を 0 と直接数値が返されます. この事を利用して, 行列の処理だけで for 文等を用いずに全てを処理します.

```
octave:66> x=[1:5,5:-1:1]
```

```
x =
```

```
1 2 3 4 5 5 4 3 2 1
```



```
octave:67> x==2
```

```
ans =
```

```
0 1 0 0 0 0 0 0 1 0
```

```
octave:68> y=find(x==2)
```

```
y =
```

```
2 9
```

```
octave:69> x(y)
```

```
ans =
```

```
2 2
```

この例では2に等しいものを検出していますが,C と比べ, 非常に簡単な処理で2に等しい元の位置を見付け出しています. この検出は等号だけでは無く, 不等号に対しても適用が可能です.

```
octave:83> x=[1:5,5:-1:1]
```

```
x =
```

```
1 2 3 4 5 5 4 3 2 1
```

```
octave:84> y=find(x>3)
```

```
y =
```

```
4 5 6 7
```

```
octave:85> z=x>3
```

```
z =
```

```
0 0 0 1 1 1 1 0 0 0
```

```
octave:86> z.*x
```

```
ans =
```

```
0 0 0 4 5 5 4 0 0 0
```

この find 命令は与えられた行列で 0 と異なる成分の位置を返す命令です. Octave で行列は (i, j) で指定しなければならないので, 並びの照合の対象がベクトルでなければ $x(\text{find}(x>3))$ の様な処理はエラーになります. $\text{find}(x>3)$ で返された列ベクトルは $x>3$ で生成された行列を列ベクトルから構成されたものと看做しています. 具体的には m 行 n 列の行列の (i, j) 成分は $m \times n$ 個の成分の列ベクトルの $m(j-1) + i$ 番目の成分に対応します.

```
octave:5> aa=rand(5);
```

```
octave:6> bb=aa>0.5
```

```
bb =
```

```
1 1 0 0 1
1 1 0 1 0
0 1 1 0 1
1 0 0 1 0
0 1 1 1 0
```

```
octave:7> find(bb)
```

```
ans =
```

```
1
2
4
6
7
8
10
13
15
17
19
20
```

21
23

```
octave:8>aa(find(bb))
error: single index only valid for row or column vector
error: evaluating index expression near line 8, column 1
octave:8>
```

MATLAB クローンでは、 $y=x(x>3)$ の様に `find` を利用せずに処理する事も可能です。
この様に並びの照合を適用して処理の簡略化が行えます。
例えば、上記の与えられたベクトルから 3 よりも大きな数値に対してのみ 2 倍する事も簡単に出来てしまいます。

```
octave:89> x=[1:5,5:-1:1]
x =
```

1 2 3 4 5 5 4 3 2 1

```
octave:90> y=find(x>3)
y =
```

4 5 6 7

```
octave:91> for i=x(y)
> 2*i
> end
ans = 8
ans = 10
ans = 10
ans = 8
```

前述の様に、MATLAB クローンでは `for` を利用する事は薦められる事ではありません。
寧ろ、次の方法で処理する方が美しく、処理も速くなります。

```
octave:1> x=[1:5,5:-1:1]
```

x =

```
1  2  3  4  5  5  4  3  2  1
```

```
octave:2> z=zeros(size(x));
```

```
octave:3> z(x>3)=2*x(x>3)
```

z =

```
0  0  0  8 10 10  8  0  0  0
```

この処理の一例として $x \geq 0$ の場合は $2, x < 0$ の場合に -1 を設定する場合は、次の様に処理すれば無駄な for 文を使った反復処理なしで、簡易に済ませられます。

```
octave:10> (x>=0)*2+(x<0)*(-1);
```

```
octave:11> tmp=(x>=0);
```

```
octave:12> tmp*2+(1-tmp)*(-1)
```

この方法の処理の変種を比較したものを以下に示します。この計算は Pentium III 1GHz 相当の PC の結果です。

```
octave:105> x=rand(100000,1);
```

```
octave:106> t1=time;(x>=0.5)*2+(x<0.5)*(-1);t2=time;t2-t1
```

```
ans = 0.042382
```

```
octave:107> t1=time;tmp=(x>=0.5);tmp*2+(1-tmp)*(-1);t2=time;t2-t1
```

```
ans = 0.027326
```

```
octave:108> t1=time;tmp=(x>=0.5);tmp*2+tmp-1;t2=time;t2-t1
```

```
ans = 0.023695
```

```
octave:109> x;t1=time;for i1=[1:length(x)]
```

```
> if x(i1)>=0.5; x(i1)=x(i1)*2; else x(i1)=-x(i1);
```

```
> end;end;t2=time;t2-t1
```

```
ans = 8.4271
```

最初の例では $x \geq 0.5$ と $x \geq 0.5$ の二種類の並びの照合を実行しています。二番目の例では 0.5 の並びの照合のみを行っていますが、ここで -1 の積を余計に実行しています。三番目の例では二番目の例と似ていますが、最後の積を予め実行したものです。そして、最後の例は for 文を用いて、一々同じ処理を繰返したものです。最速のものと比較して実に 400 倍近くもの差が生じていますね。結局、行列の積では専門のライブラ

リが用いられているお陰で, 下手に for 文を用いるよりは効率的に計算が出来ている事が分ります. この理由からも, 安易な for 文の利用は出来るだけ避けて, 行列の演算に置換えられるものは徹底して置換えた方が無難な事が分ると思います.

猶, 並びの照合に関連して, MATLAB や Octave の便利な命令に any と all 命令があります. any 命令は行列データに零でない成分があれば 1, 零行列であれば 0 を返します. これに対し, all は全ての成分が零でない場合のみ 1 を返し, 他は 0 となります. この命令を用いれば, 更に余計な処理を行わずに済みます.

```
octave:1> a=rand(4,5)-rand(4,5)
a =

-0.671536    0.539990    0.205556    0.171495    0.276634
-0.784795    0.585699   -0.274086   -0.448760    0.131415
-0.072425    0.276092    0.355440   -0.257676    0.357314
 0.356246    0.061500   -0.318618    0.241485   -0.295221
```

```
octave:2> if any(a(:,1)>0)
> lst=find(a(:,1)>0);
> b=exp(a(lst,1));
> end;
```

```
octave:3> b
b = 1.4280
```

この様に any 命令を使うと, 添字集合として使うリストが空リストかどうかを心配する必要が無くなります. 但し, 全てが一致するかどうかは any 命令だけでは判別出来ません. 全てが 1 の場合に 1 を返す命令として all 命令があります.

```
octave:11> all(a(:,1)==a(:,3))
ans = 0
octave:12> a(:,1)=a(:,3);
octave:13> all(a(:,1)==a(:,3))
ans = 1
octave:14>
```

この様に並びの照合を適用した数値処理は非常に強力で, for 文による反復処理無しに全てを簡潔に済ます事が可能になります.

15.6 便利な行列の定義方法

MATLAB や Octave では等間隔のベクトルの設定を行う時に, 全ての数値を直接入力する必要はありません.[初期値:刻み幅:終値] で設定出来てしまいます.

```
octave:111>bb=[0:0.1:0.5]
bb =
```

```
0.00000 0.10000 0.20000 0.30000 0.40000 0.50000
```

勿論, 禁断の for 文を用いて次の様に設定する事も出来ますが, 手間は逆に増えてしまいます.

```
octave:115> bb
bb =
```

```
0.00000 0.10000 0.20000 0.30000 0.40000 0.50000
```

先程調べた様に, 下手に for 文を用いると速度の低下に繋がり易い問題がありました. これは行列の生成でも同様です. 速度を比較してみましょう.

```
octave:116> t1=time;for i=1:100; bb(i,i)=0.01*i;end;t2=time;t2-t1
ans = 0.0047450
octave:117> t1=time;dd=[1:100];t2=time;t2-t1
ans = 4.9114e-05
```

この場合でも,for 文を使うと著しく速度が低下する事が分りますね.

そこで, 行列の定義でも可能な限り for 文の利用は避けたいものです. 特定の行列に関しては, 容易に行列が生成出来る命令が幾つかあります. 先ず, 対角成分に決まった数値を設定する場合は diag 命令を用います.diag 命令では対角成分に設定する数値をベクトルで与えます. ここで, 対角成分を指定した列程ずらす事も, 与えられた行列の対角成分を抜き出す事も可能です.

対角成分が全て 1 で他が全て 0 の行列の生成は `eye` 命令 を使います.

```
octave:118> diag([1,2])
```

```
ans =
```

```
1  0
0  2
```

```
octave:119> diag([1,2],1)
```

```
ans =
```

```
0  1  0
0  0  2
0  0  0
```

```
octave:120> diag([1,2],2)
```

```
ans =
```

```
0  0  1  0
0  0  0  2
0  0  0  0
0  0  0  0
```

```
octave:121> diag([1,2],-2)
```

```
ans =
```

```
0  0  0  0
0  0  0  0
1  0  0  0
0  2  0  0
```

```
octave:122> a=rand(3,3)
```

```
a =
```

```
0.58905 0.61873 0.63411  
0.19251 0.11602 0.18785  
0.54143 0.83113 0.83952
```

```
octave:123> diag(a)
```

```
ans =
```

```
0.58905  
0.11602  
0.83952
```

```
octave:124> eye(3,2)
```

```
ans =
```

```
1 0  
0 1  
0 0
```

```
octave:125> eye(2,3)
```

```
ans =
```

```
1 0 0  
0 1 0
```


15.7 多項式の扱い

Octave や MATLAB では多項式を扱う事も可能です. 但し, 数式処理の様に直接多項式は扱えません. Octave では多項式の係数ベクトルの形で扱う事になります.

——— 多項式の変換 ———

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \quad \Leftrightarrow \quad [a_n, a_{n-1}, \dots, a_1, a_0]$$

この様に, 多項式は 1 変数のものに限定されます.

猶, 多項式同士の積は `conv` 命令で行い, `deconv` 命令で多項式の商と剰余を計算し, `polyval` 命令で多項式の変数に値を代入した結果を返します.

```
octave:2> conv([1,2],[1,-2])
```

```
ans =
```

```
1    0   -4
```

```
octave:3> conv([1,2,2,1],[1,-2])
```

```
ans =
```

```
1    0   -2   -3   -2
```

```
octave:4> [aa,bb]=deconv([1,3,3,1],[1,1,1])
```

```
aa =
```

```
1    2
```

```
bb = -1
```

```
octave:5> polyval([1,2,3,4],2)
```

```
ans = 26
```

この例では, 最初に `conv([1,2],[1,-2])` で多項式 $(x+2)(x-2)$ の展開を計算しています. この結果は `[1,0,-4]` となり, これは $x^2 + 0 \cdot x - 4$, 即ち, $x^2 - 4$ に対応します. `deconv` の例では $x^3 + 3x^2 + 3x + 1$ を $x^2 + x + 1$ で割った時の商と剰余を計算しています. この結果は $x^3 + 3x^2 + 3x + 1 = (x^2 + x + 1)(x + 2) - 1$ である事を示しています. こ

の他にも多項式を扱う関数が幾つか存在しますが、どれも直接的な計算を行うものではなく、あくまでも多項式をリストで扱うものです。

この様に、Octave では多項式の処理は得意とは言い難い面があります。だからこそ、この様な処理で複雑なものは Maxima 等の数式処理に任せ、逆に数値行列の演算に徹するのが効率の面でも良いでしょう。何でも一つの道具だけで済まそうとする事は、便利そうに見えても思った程、便利でも無く、賢明な手段はありません。

15.8 安直な関数の定義

関数の定義は拡張子が `.m` のファイル (M-file) を記述します。この M-file がカレントディレクトリ、或いは M-file が存在するディレクトリが `path` に含まれていれば、自動的に M-file の読込んで実行します。

```
function [z]=nekoneko(x,y)
    if length(x)>length(y)
        z=x;
    else
        z=x./y;
    end;
```

次に、Octave で関数を直接定義したい場合には、M-file の内容をそのまま記述します。以下に、Octave 上で直接関数を定義している様子を示します。ここで、`>` は Octave の入力待ちのプロンプトです。

```
octave:1> function [z]=nekoneko(x,y)
> if length(x)==length(y)
> z=x./y;
> else if length(x)>length(y)
> z=x;
> else
> z=y;
> end
> end
> end
octave:2> nekoneko([1:3],[3:-1:1])
ans =
```

```
0.33333  1.00000  3.00000
```

```
octave:3> nekoneko([1:3],[3:-1:0])
ans =
```

```
3  2  1  0
```

```
octave:4> nekoneko([1:5],[3:-1:0])
ans =
```

```
1  2  3  4  5
```

関数の内容を見たい場合には type 命令を利用します. 実例を以下に示しておきます.

```
octave:6> type nekoneko
nekoneko is a user-defined function:
```

```
function z = nekoneko (x, y)
  if length (x) == length (y)
    z = x ./ y;
  else
    if length (x) > length (y)
      z = x;
    else
      z = y;
    endif
  endif
endfunction
octave:7>
```

猶,type で内容を見る事が可能なものは M-file や利用者定義の変数で, 組込関数は事が出来ません. 組込関数かどうかは who で調べる事が可能です.

```
octave:8> who
```

```
*** currently compiled functions:
```

```
length      nekoneko
```

```
*** local user variables:
```

```
aa  bb
```

```
octave:9> type aa
```

```
aa is a user-defined variable
```

```
[ 1, 2, 3 ]
```

```
octave:10>
```

この様に,who を実行して表示される変数や関数には type で内容を見られます. 猶, π 等の数学的定数は組込関数として扱われています. その為,who 命令を実行しても表には出て来ません.

15.9 外部アプリケーションの起動命令

Octave には外部命令を実行する命令として,system 命令 と exec 命令の二種類があります. 因に MATLAB の場合は!を先頭に付けて外部の命令を起動させますが, 行末に ; を追加すると外部命令のオプションとして判断されるので注意が必要です. Octave では!は否定を表わす not の意味になるので,MATLAB と Octave で動作するプログラムを開発する場合には特に注意が必要になります.

exec 命令は正常に処理の実行が終了しすると Octave 自体を終了させてしまいますが,system 命令は, 実行終了後に Octave に戻る所以, 通常は system 命令を用いると良いでしょう.

ここでは system 命令を用いた安易な例を以下で紹介しましょう.

先ず,system 命令で起動させるプログラムは外部のプログラムであれば何でも構いません. ここでは次のシェルスクリプトとします. このスクリプトはカレントディレクトリ上でファイルサイズのみをファイル x1 に出力するものです.

```
#!/bin/sh
```

```
ls -l | awk '{print $5}'>x1
```

次に,system 命令を用いるプログラム mike を以下に示します.

```
function x = mike ()
    system ("tama");
    load ("x1");
    x = sum (x1);
endfunction
```

この mike は先程のシェルスクリプト tama を system 命令で起動させて、結果ファイルの x1 を読んで、その総和を計算するものです。猶、system 命令で実行する命令 tama は環境変数 path で設定されたディレクトリかカレントディレクトリ上にあれば大丈夫です。猶、ディレクトリを system("/usr/bin/tama") の様に直接指定しても良いでしょう。更に、system 命令で実行するプログラムが引数を必要とする場合、system("tama mike") の様に通常利用する命令を単に二重引用符" で括ります。

例えば、次のシェルスクリプト pochi で指定したディレクトリ上のファイルサイズを x1 に出力します。上述の tama との違いは二行目に \$1 が追加されている点です。

```
#!/bin/sh
ls -l $1 | awk '{print $5}'>x1
```

この pochi に対し、mike を改良した kuro を以下に示します。

```
function x = kuro (wrđ)
    evl=["pochi ",wrđ];
    system (evl);
    load ("x1");
    x = sum (x1);
endfunction
```

この kuro では evl で文字列の結合を行っています。例えば、wrđ として "/usr" が与えられると、/usr 上のファイルサイズの総和を計算する事になりますが、evl では文字列 "pochi " と word として与えられた "/usr" が結合された "pochi /usr" が代入されます。この値を system 命令で評価 (その為に evl を "" で括っていません) して生成された x1 を用いて総和が計算されます。

この様に、system 命令を用いる事で外部プログラムを用いる事が可能となり、全てを Octave 言語に書き直す手間を省いたり、Octave への外部プログラムの組込みに悩む前に、安易にシステム動作の確認が行えるので重宝します。

猶、Octave には cd, ls 及び pwd といった命令が利用可能です。働きも UNIX のそれと全く同じものです。

15.10 グラフ表示機能

Octave ではグラフ表示に `gnuplot` を用います。その為、`gnuplot` がインストールされていない環境ではグラフ表示が行えません。

基本的に、二次元グラフの表示では Matlab との互換性の問題はあまりありませんが、曲面表示になると、Octave では機能的にも劣っている為、互換性は低くなります。

では、簡単にグラフの表示について解説しましょう。グラフ表示機能で最も互換性が高いのが、単純な二次元グラフの表示です。二次元表示を行う場合、`plot` 命令を用います。

plot 命令

```
plot(<Y - データ>, <書式>, ...)  
plot(<X - データ>, <Y - データ>, <書式>, ...)
```

ここで、`plot` 命令に Y 軸データ単体を与える場合、行や列ベクトル、或いは行列を与える事が出来ます。ここで、`m` 行 `n` 列の行列 `A` を与えた場合、`plot` 命令は、`A` の `n` 個の列ベクトルを描きます。この場合、X 軸は 1 から `n` の整数で、Y 座標は、その X 座標に対応する列ベクトルの成分となります。

次に、X と Y のデータを分けて描く事も可能です。この場合、X と Y は同じ大きさのベクトル、或いは行列が設定可能です。

網目を入れたい場合

グラフに網目を入れたければ、`grid` 命令を使います。この `grid` 命令は、引数として文字列の "on" か "off" だけを取ります。on の場合に網目を入れ、off の場合には網目の表示を解除します。

グラフの重ね描きをしたい場合

色々なデータのグラフを重ね描きをしたい場合、`hold` 命令を使います。この `hold` 命令も `grid` 命令と同様に、引数として文字列の "on" か "off" だけを取ります。on の場合は重ね描きを行い、off の場合には、重ね描きを行わずに、グラフの更新を行います。

グラフにラベルや表題を入れたい場合

X 軸、Y 軸、Z 軸上にラベルを入れたければ、`xlabel`, `ylabel`, `zlabel` 命令を用います。又、上側に表題を入れたければ、`title` 命令を用います。これらの命令は一つの文字列を引数とします。

では、実際に曲線を描いてみましょう。

```
octave:1> a1=[0:0.05:1]*2*pi;  
octave:2> b1=sin(a1);  
octave:3> plot(b1);  
octave:4> plot(a1,b1);  
octave:5> xlabel("X")  
octave:6> ylabel("Y")  
octave:7> title("sine curve");  
octave:8> grid("on");  
octave:9> hold("on");  
octave:10> c1=cos(a1);  
octave:11> plot(a1,c1);  
octave:12> title("red:sin, green:cos");  
octave:13> hold("off");  
octave:14> plot(a1,b1,a1,c1);  
octave:15> X=[a1;a1];  
octave:16> Y=[b1;c1];  
octave:17> plot(X,Y);  
octave:18> plot(X',Y');
```

この例では正弦函数と余弦函数を描いています. 最初に配列 b1 を描くと, 図 15.1 に示す様に, X 軸には配列番号が振られます.

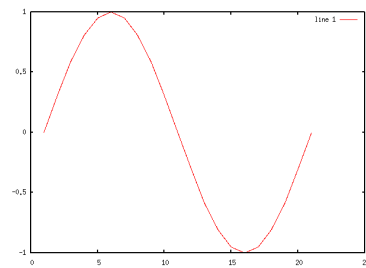


図 15.1: 正弦函数の描画 (Y 座標のみ)

次に, 配列 `a1` と配列 `b1` の対にすると, 図 15.2 に示す様に, 配列 `a1` を X 軸, 配列 `b1` を Y 軸の座標としてグラフを描きます.

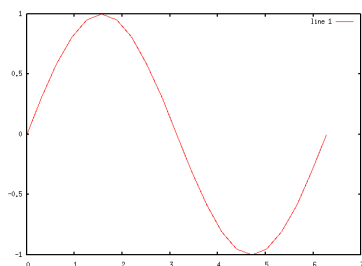


図 15.2: 正弦函数の描画

次に, `xlabel` と `ylabel` 命令で X 軸と Y 軸にラベルを入れ, `title` 命令でグラフの表題を入れてみたものが, 図 15.3 になります. 因に, `xlabel` 等を用いると, その都度, グラフは更新されます.

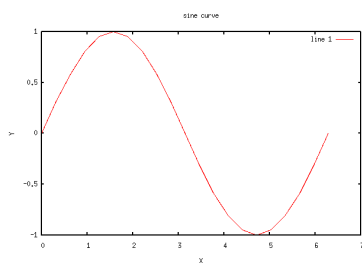


図 15.3: グラフにラベルと表題を追加

それから, `grid("on")` で網目を入れたものが図 15.4 になります.

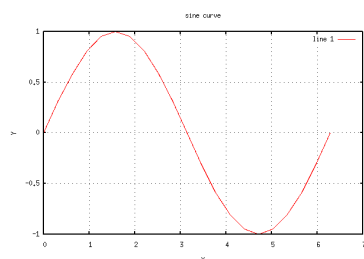


図 15.4: 網目を追加

複数のグラフを描く場合は, 二種類の方法があります. 一つは, `hold` 命令を使って, `plot` 命令を繰り返して描く方法, もう一つは, `plot` 命令で一度に描く方法です. `plot` 命令で一度に描く場合, `plot(a1,b1,a1,c1)` の様に `X` と `Y` の値の対を並べる方法と, `X=[a1;a1]` と `Y=[b1;c1]` の様に `X` の値と `Y` の値の行列を作って `plot(X',Y')` で一度に描く方法があります. どちらの方法でも図 15.5 の様なグラフが描かれます.

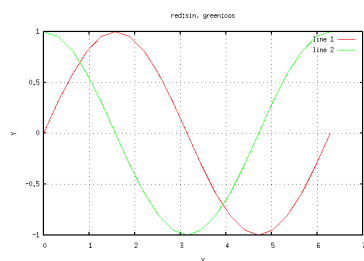


図 15.5: `plot(X',Y')` のグラフ

猶, `plot(X,Y)` とすると, `plot` 命令は列ベクトルで描こうとする為, 図 15.6 の様な意味不明のグラフが出るので, この点には注意しましょう.

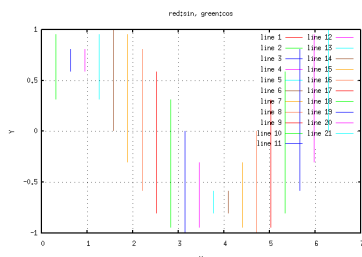


図 15.6: `plot(X,Y)` のグラフ

15.11 Octave で file を利用する話

この節では, Octave でファイルを利用する方法について簡単に説明します. その為, ここで説明する命令についても, その機能の一部のみを用いるだけで, 全てを活用しているとは限りません.

15.11.1 load 命令によるデータファイルの処理

最も基本的な, 数値行列を含むファイルの読込処理は `load` 命令で対処出来ます. 実際に試してみましょう. ここではファイル名を `neko` にして, 行列データは次の数値にしましょう.

——— ファイル `neko` の内容 ———

```
1 2 3
4 5 6
```

では実際にファイル `neko` の読込を行いましょ. 読込は単純に `load neko` の様にファイル名を必要があればパス付きで指定するだけです.

```
octave:1> load neko
octave:2> neko
neko =
```

```
1 2 3
```

```
4 5 6
```

これで読み込みが出来ました. この例で示す様に行列の名前はファイル名が割当てられていますね. では, 続けて, 同じ名前のファイルを読み込んでみましょう. どうなりますか?

```
octave:3> load neko
warning: load: local variable name 'neko' exists.
warning: use 'load -force' to overwrite
error: load: unable to load variable 'neko'
error: evaluating index expression near line 3, column 1
```

何か, 色々と文句を言っていますね. Octave では自動的に読み込まれたファイル名と同じ名前の変数が既に利用されていると, この様にエラーが返されます. そして, 既存のデータは上書きされずに保存されます.

そこで, load 命令に -force オプションを付けてみましょう. ここでは, `load force neko` と入力してみましょう.

```
octave:3> load -force neko
octave:4> neko
neko =
  1  2  3
  4  5  6
```

今度はどうでしょう. 上書きされていますね.

では, 行列名はどのような規則で付けられているのでしょうか. 実は, neko と同じ内容のファイルで, neko.matrix という名前のファイルがあるので, 続けて読み込んでみましょう.

```
octave:5> load neko.matrix
warning: load: local variable name 'neko' exists.
warning: use 'load -force' to overwrite
error: load: unable to load variable 'neko'
error: evaluating index expression near line 5, column 1

octave:5> load -force neko.matrix
```

```
octave:6> who
```

```
*** local user variables:
```

```
neko
```

```
octave:7>
```

またエラーが出ていますね。行列名は拡張子を外した名前の部分が使われるので、変数名は、neko.matrix の場合も neko になります。そこで、load 命令に-force オプションが無かった為に、変数の内容保護の為にエラーとなったのです。最後に who と入力してみましょう。neko と返されていますね。これは利用者定義の変数が neko しか無い事を意味します。この who は利用者が定義した変数を返す命令です。

数値行列ファイルの読み込みが出来るなら、その逆はどうするのでしょうか。単純な数値行列データのファイルへの保存には save 命令が使えます。save 命令は先程の who 命令を実行して表示される利用者定義のデータ全ての指定ファイルへの保存と、個別のデータの保存の両方が行え、その上、既存ファイルへのデータの追加も行えます。

```
octave:1> a=rand(4,4);
```

```
octave:2> b=rand(3,1);
```

```
octave:3> save neko a
```

```
octave:4> save test
```

```
octave:5> who
```

```
*** local user variables:
```

```
a b
```

この例では、行列 a,b を各々 rand 命令で生成した 4 行 4 列と 3 行 1 列の行列としています。 `save neko a` で、行列 a をファイル neko に保存し、`save test` で who で表示されるデータ全てをファイル test に保存しています。ここで、ファイル neko とファイル test の内容を確認しましょう。

ファイル neko の内容

```
# Created by Octave 2.0.16, Thu May 10 08:21:44 2001
# name: a
# type: matrix
# rows: 4
# columns: 4
0.590789258480072 0.222358718514442 0.876821994781494 0.949454307556152
0.741063475608826 0.656238257884979 0.365377485752106 0.979949235916138
0.395543217658997 0.417380422353745 0.444111585617065 0.857901215553284
0.568090081214905 0.558982253074646 0.0379265695810318 0.475694209337234
```

ファイル test の内容

```
# Created by Octave 2.0.16, Thu May 10 08:21:53 2001
# name: a
# type: matrix
# rows: 4
# columns: 4
0.590789258480072 0.222358718514442 0.876821994781494 0.949454307556152
0.741063475608826 0.656238257884979 0.365377485752106 0.979949235916138
0.395543217658997 0.417380422353745 0.444111585617065 0.857901215553284
0.568090081214905 0.558982253074646 0.0379265695810318 0.475694209337234
# name: b
# type: matrix
# rows: 3
# columns: 1
0.628571033477783
0.415022879838943
0.216913774609566
```

この様に,save 命令で変数を指定すると,変数に割当てられた値がファイルに保存され,変数を指定しないと,who 命令で表示される変数に割当てられた値がファイルに保存されます.

既存ファイルにデータの追加する場合,save 命令に-append オプションを付けます. 例えば, 上記の test ファイルに行ベクトル c を追加したければ, 以下の様に処理を行います.

```
octave:6> c=[1,2,3];
```

```
octave:7> save -append test c
```

ここで, c を追加後のファイル test の内容を確認しておきましょう.

```
# Created by Octave 2.0.16, Thu May 10 08:21:53 2001
```

```
# name: a
```

```
# type: matrix
```

```
# rows: 4
```

```
# columns: 4
```

```
0.590789258480072 0.222358718514442 0.876821994781494 0.949454307556152
```

```
0.741063475608826 0.656238257884979 0.365377485752106 0.979949235916138
```

```
0.395543217658997 0.417380422353745 0.444111585617065 0.857901215553284
```

```
0.568090081214905 0.558982253074646 0.0379265695810318 0.475694209337234
```

```
# name: b
```

```
# type: matrix
```

```
# rows: 3
```

```
# columns: 1
```

```
0.628571033477783
```

```
0.415022879838943
```

```
0.216913774609566
```

```
# name: c
```

```
# type: matrix
```

```
# rows: 1
```

```
# columns: 3
```

```
1 2 3
```

ファイルの保存の場合は load 命令と異なり, 指定したファイルが存在していても, 無警告で処理が実行されます. 実際, test ファイルが存在する状態で test に行列 a と c を保存する例を以下に示しておきます.

```
octave:8> save test
```

```
octave:9> save test a c
```

```
octave:10>
```

save 命令で保存した行列 a と c を含むファイル test の中身を以下に示します.

```
# Created by Octave 2.0.16, Thu May 10 08:29:54 2001
# name: a
# type: matrix
# rows: 4
# columns: 4
0.590789258480072 0.222358718514442 0.876821994781494 0.949454307556152
0.741063475608826 0.656238257884979 0.365377485752106 0.979949235916138
0.395543217658997 0.417380422353745 0.444111585617065 0.857901215553284
0.568090081214905 0.558982253074646 0.0379265695810318 0.475694209337234
# name: c
# type: matrix
# rows: 1
# columns: 3
1 2 3
```

猶,save で保存したデータは load 命令でそのまま読込めます.

```
octave:1> load test
octave:2> who
*** local user variables:
a c
octave:3> a
a =
0.590789 0.222359 0.876822 0.949454
0.741063 0.656238 0.365377 0.979949
0.395543 0.417380 0.444112 0.857901
0.568090 0.558982 0.037927 0.475694
```

```
octave:4> c
c =
    1    2    3
octave:5>
```

これで、最も基本的な数値行列のファイルの読込の方法を終えます。

MATLAB や Octave ではもっと複雑な処理が出来ます。両方とも言語的に C に似ている為、C 風にファイルの操作が出来ます。この機能を用いれば、非常に複雑な形式のファイルの読込と書込も可能です。ところが、Octave の方が処理言語の機能が上なので、MATLAB でも利用可能なプログラムを構築しなければならない時には注意が必要になります。ここでは MATLAB との互換性に留意して記述を纏め、Octave 独自の機能は利用しない様になっています。

15.11.2 ファイルの Open と Close

ファイルを開く場合、fopen 命令を用います。

—— fopen 命令の構文 ——

```
id = fopen(<ファイル名>,<型>)
```

先ず、<ファイル名> には 'neko' や 'test/neko.dat' の様なディレクトリを含めたファイル名前を設定します。次に、fopen 命令の <型> は以下のものを用います。

—— fopen の型 ——

1. 'r' 読み込み
2. 'w' 新規に書込み
3. 'a' 末尾に追加
4. 'r+' 読み込み. 内容の更新も可
5. 'w+' 書込み. 内容の更新も可
6. 'a+' 末尾に追加. 内容の更新も可.

fopen 命令は整数値を返します。この値 id は以降のファイル操作で利用し、操作するファイルの指定に利用します。猶、ファイルが存在しない等のファイルのオープンでエラーを出した場合、-1 が返されるので、この返却値を利用してエラー処理が出来ます。次に、ファイルの close は C と同様に fclose 命令を使って、fclose(id) で指定したファイルを閉じます。

ファイル内部の移動ではポインタを用います。ファイルを開くとポインタはファイルの先頭に置かれていますが、ファイルの読込等を行うと下の方に移動して行きます。そこで、ファイルの先頭にポインタを動かす必要が出て来ると、`frewind` 命令を使います。データの読込や書込では `fgets`, `fputs`, `fscanf` 等の命令が存在し、基本的な処理は C と同様の命令が揃っています。ところが、実際の機能は C の同名の命令とは微妙に異なるので注意が必要になります。特に MATLAB との互換性を考慮すると、Octave の機能をフルに活用したプログラムはそのままでは使えず、修正が必要になってしまいます。これは入力と出力書式の指定で顕著です。MATLAB では基本的に行単位で書式が固定され、微妙な調整が行えません。

その為、一行に整数、文字列、浮動小数が混在する場合、入出力の書式を指定して読込む方式是避け、`fgets` で一行を `stream` として取込んで、`stream` を分解して `sscanf` で形式の変換を行う事を薦めます。この方法に関しては次の節で説明しましょう。

15.11.3 データの読み込み

ファイルのデータ読み込みは `fgets`, `fscanf` 等の命令が使えます。猶、Octave の `fscanf` 命令は上述の様に MATLAB のものと比べ機能が強化されており、その上、`C-flag` を立てる事によって C 言語の `fscanf` と同じ利用が可能です。ところが、MATLAB では書式が行毎で数値や文字単位ではない為、Octave 専用のプログラムになってしまうので、MATLAB との互換性を重視したプログラムでは、文字、数値が混在する行を扱う場合、`fgets` 命令を使って一行を `stream` として取り込み、その `stream` を文字列照合や長さで分割したものに対して `sscanf` を用いて形式の変換を行う方が、処理は繁雑になるかもしれないが安全です。

以下に単純な実例を示しましょう。ここでの例では単純な数値行列データに日本語を含めた文字列を含むものです。

neko.txt ファイルの内容

```
1 2 3
3 2 1
はい,1 2 3 ある日森の中, 熊さんに出逢った。
```

このファイルは 1 行と 2 行が数値ですが、3 行目は数値と文字が混在しています。このファイル ('neko.txt') を Octave で開き、`fgets` を用いて一行ずつ読み込む様子を以下に示しましょう。

```
octave:43> fid=fopen('neko.txt','r');
```

```
octave:44> L1=fgets(fid)
L1 = 1 2 3
octave:45> L2=fgets(fid)
L2 = 3 2 1
octave:46> L3=fgets(fid)
L3 = はい,1 2 3 ある日森の中, 熊さんに出逢った.
octave:47> frewind(fid)
ans = 0
octave:48> L4=fgets(fid)
L4 = 1 2 3
```

この例では、ファイルの内容参照を行う為に、`fopen` 命令の型の指定で、`'r'` を指定しています。一般的には、`'r'` か `'r+'` でファイルを開きます。ところで、間違って `'w'` や `'w+'` を指定すると、直ちにファイルが更新され、以前の内容が消去された状態となるので注意が必要です。

`fgets` 命令は例で示す様にファイルの先頭から一行ずつ読みを行います。ここで、ファイルの先頭にポインタを移動させる為には、`frewind` 命令を使います。

ところで、`fgets` で返される値の型は実は文字列型です。この例で、一見するとベクトルにしか見えない `L1` は文字列型です。実際に、`L1` の和を計算しようとするすると以下のエラーが出ます。

```
octave:56> L1+L1
error: invalid conversion from string to real matrix
error: invalid conversion from string to real matrix
error: evaluating assignment expression near line 56, column 3
octave:56> size(L1)
ans =
    1    6
octave:57> L1
L1 = 1 2 3
```

この様に、テキストファイルデータを `fgets` 命令で読み込むと、`fgets` が返す値は全て文字列型になってしまいます。そこで、必要に応じて型の変換を行わなければなりません。型の変換は C と同様に `sscanf` 命令を用います。

sscanf の型の指定

%d ⇒ 整数型データに変換
%f ⇒ 浮動小数点型データに変換
%s ⇒ 文字列型データに変換

以下に、文字列 L1=1 2 3 と L2=3 2 1 を sscanf 命令を使って型の変換を行った实例を示します。

```
octave:51> a11=sscanf(L1,'%d')
a11 =
    1
    2
    3
octave:52> a12=sscanf(L1,'%s')
a12 = 123
octave:53> a12=sscanf(L1,'%f')
a12 =
    1
    2
    3
octave:54> a11=sscanf(L1,'%d')
a11 =
    1
    2
    3
octave:55> a12=sscanf(L2,'%d')
a12 =
    3
    2
    1
octave:56> a11+a12
ans =
    4
    4
    4
```

猶,d の場合,Octave では自動的に列ベクトルになります. ところが, MATLAB では行ベクトルになります. これは Octave が列ベクトルを基準としている傾向がある為でしょう. この点は Octave と MATLAB の両方で動作するプログラムを作成する際には, 特に注意が必要な個所の一つです.

この様にファイルが文字列か数字列の何れかで構成されている場合,fgets 命令で行毎読んで,sscanf で形式の変換を一度に行えば良い事になります. ところが,L3 の様に数と文字が混合している場合は厄介です.L3 の様に意地の悪い代物は, 悪意を持って例として示しているので仕方ありませんが, 次のファイル tama の様な形式は非常に普通でしょう.

#	No.	Flag	Value
1		t	10
2		f	-10
3		t	20
4		f	-20

この様なデータの場合,sscanf で一気に変換する事は意味が無く, 本来なら書式指定で各々を変換するのが本来の姿です. ところが,MATLAB には与えられたストリームを一気に変換する事しか出来ません.

先ず, ファイル tama を開き, 安易に一行づつ sscanf で整数型 ('%d') や文字列型 ('%s') へと一気に変換した例を示しましょう.

```
octave:73> fid2=fopen('tama','r')
```

```
fid2 = 3
```

```
octave:74> tama1=fgets(fid)
```

```
tama1 = # No. Flag Value
```

```
octave:75> tama2=fgets(fid)
```

```
tama2 = 1 t 10
```

```
octave:76> tama3=fgets(fid)
```

```
tama3 = 2 f -10
```

```
octave:77> tama4=fgets(fid)
```

```
tama4 = 3 t 20
```

```

octave:78> tama5=fgets(fid)
tama5 = 4      f      -20

octave:79> st1=sscanf(tama1,'%d')
st1 = [] (0x1)
octave:80> st1=sscanf(tama2,'%d')
st1 = 1
octave:81> st1=sscanf(tama3,'%d')
st1 = 2
octave:82> st1=sscanf(tama4,'%d')
st1 = 3
octave:83> st1=sscanf(tama5,'%d')
st1 = 4
octave:84> st1=sscanf(tama1,'%s')
st1 = #No.FlagValue
octave:85> st1=sscanf(tama2,'%s')
st1 = 1t10
octave:86> st1=sscanf(tama3,'%s')
st1 = 2f-10
octave:87> st1=sscanf(tama4,'%s')
st1 = 3t20
octave:88> st1=sscanf(tama5,'%s')
st1 = 4f-20

```

この様に,sscanf による変換では形式に対応しない個所以降の列は除外されてしまいます. 例えば, '%d' で整数型に変換する個所を見て頂くと判りますが, 二列目の flag に当たって変換が途中で終了してしまい, flag の前の No. に相当する数のみが返されています. 更に, ファイル先頭行の文字列は変換が出来ずに空行が返されています. その上, ストリームを一気に文字列に変換した場合, 空行やタブは省略されています. 実際, 2 f -10 が 2f-10 に変換されたりしていますね.

この様に MATLAB の sscanf, fscanf の互換性のある方式ではこれ以上の処理が出来ません. そこで, Octave で sscanf 命令に C フラグを立てると C と同様に複雑な形式のファイルにも対応出来る様になります.

```

octave:94> [s1,s2,s3,s4]=sscanf(tama1,'%s %s %s %s','C')
s1 = #

```

```
s2 = No.
```

```
s3 = Flag
```

```
s4 = Value
```

```
octave:95> [n1,flg,n3]=sscanf(tama2,'%d %s %d','C')
```

```
n1 = 1
```

```
flg = t
```

```
n3 = 10
```

この様に Octave であれば、より C 風に sscanf や fscanf を利用出来ますが、この C フラグを立ててしまうと今度は MATLAB では利用出来ません。この場合、スマートではありませんが、次の様にストリームを分けて対処すれば互換性に問題が生じる事もなく、上手に処理が行えます。

```
octave:96> find(tama2=='t' | tama2=='f')
```

```
ans = 7
```

```
octave:97> n1=sscanf(tama2(1:6),'%d')
```

```
n1 = 1
```

```
octave:98> n2=sscanf(tama2(8:length(tama2)),'%d')
```

```
n2 = 10
```

```
octave:99> flg=sscanf(tama2(7),'%s')
```

```
flg = t
```

ここで、最初に find を用いている個所はストリーム tama2 からフラグ値の t か f のどちらかがある個所を求めて、その個所から前後に分けて整数に変換しています。ここで、記号'—'は Octave/MATLAB の論理和になります。

同様にコメント行かどうかは'#' がストリームに存在するかどうか検証するだけで出来てしまいます。だから、全てを一旦文字列に変換し、先頭が'#' であるかを判別する様にすれば良い事になります。

猶、重要な事に Octave と MATLAB の両方では、扱う行列データは文字か数値のみしか許容されず、両者が混在した行列を扱う事は出来ない事です。

この様に行列データに制約があるので、文字と数値が混在する表の扱いでは、色々な行列データを準備する必要がある様に思えます。

Octave と MATLAB の双方に C の構造体と同様のデータ構造を用いる事が可能なので、そちらを使うと多くの行列を利用する必要が無くなってしまいます。その上、MATLAB と Octave で構造体を利用する場合は、他の変数と同様に予め宣言する必要は無く、以下の例の様に直接利用しても構いません。

実際に Octave で構造体を用いた例を示しましょう。

```
octave:102> [neko.n1(1),neko.flg(1),neko.n2(1)]=sscanf(tama2,'%d %s %d','C')
neko.n1 = 1
```

```
neko.flg = t
```

```
neko.n2 = 10
```

```
octave:103> [neko.n1(2),neko.flg(2),neko.n2(2)]=sscanf(tama3,'%d %s %d','C')
neko.n1 = 2
```

```
neko.flg = f
```

```
neko.n2 = -10
```

この例では sscanf で変換したデータを neko.n1,neko.flg,neko.n2 に割当てています。ここで各々の配列は数値と文字列になっている事に注意して下さい。

MATLAB と Octave ではこの様に構造体を用いて文字と数値が混在したデータを一括して扱えます。ここで、データが構造体かどうかは直接データ名を入力する事でも判別可能ですが、この場合、全てのデータが一度に表示されるので、Octave の場合、is_struct 命令で構造体かどうかを判定し、struct_elements 命令で構造体の構造が調べられます。但し、MATLAB の場合は命令の名前は似ているが、全く別の命令を用います。その為、互換性に注意が必要です。

```
octave:104> neko
neko =
{
    n2 =
```

```
10

-10

flg =

t
f

n1 =

1
2
}

octave:107> is_struct(neko)
ans = 1
octave:108>
octave:109> struct_elements(neko)
ans =

n2
flg
n1
```

この様に MATLAB との互換性を考慮すると、泥臭い処理が必要になりますが、fgets と sscanf 命令を上手く用いれば、通常のファイルの処理も出来ます。

15.11.4 ファイルの更新とデータの追加の例

fopen の指定でファイルの更新、例えば、内容の入れ替えや、データの追加が行えます。先ず、ファイルの更新では、'w' を指定しましょう。こうすると、既存のファイルの内容は消去され、もしも、存在しない場合には、指定された名前のファイルが新規に生成されます。又、既存のファイルを利用する場合、'a' を指定すると、既存のファイルのデータに続けて書込めます。

次に、例として与えられた行列データをフラグに沿って指定されたファイルに追加したり、置き換えるプログラムを示しましょう。

```
function [err]=appendDATA(fname,mv,flg)
    err = 0;
    % ファイル名の設定.
    vfname=[fname, '.vdt'];
    % フラグ flg==0 であれば上書き, それ以外は末尾にデータを
    % 追加する.
    if flg==0
        vfp = fopen(vfname, 'w');
    else
        vfp = fopen(vfname, 'a');
    end;

    % データの書き込み
    [m,n]=size(mv);
    if m>0
        if flg==0
            fprintf(vfp, ' %d ', mv(1,:));
            fprintf(vfp, '\n');
        end;
        for k=[2:m]
            fprintf(vfp, '%22.15e', mv(k,:));
            fprintf(vfp, '\n');
        end;
    else
        err=1;
    end;
    fclose(vfp);
```

この例では、ファイル名は修飾子".vdt"を抜いた型で与える。又、ファイルの先頭行はカラムを分類する為に整数としており、2行目以降が実際のデータで、行列 mv もその様な形式となっています。このデータ行の出力並びは fprintf 命令の '%22.15e' で指定したもので、この書式の設定は C や FORTRAN のそれと同じ形式になります。更に、この処理を"%データの書き込み"と註釈を入れた個所から下で行っています。

以上の様に Octave のファイル処理では、命令やオプションがほぼ C に似た仕様となっているので、MATLAB との互換性を考慮しなければ、C 風に記述して C オプションを立てておけば便利です。更に、扱うデータを数値行列にすれば非常に苦労も少なく、非常に気楽にファイル操作が行える仕様となっています。

しかし、MATLAB との互換性を考慮すれば、MATLAB で書式指定を伴う処理では行単位で行われるので、一行に文字と数値が混在データファイルを扱う場合には工夫が必要になります。その上、Octave の独自の機能に頼ると MATLAB で動作しないプログラムとなる可能性が非常に高くなってしまいますので注意しましょう。

15.12 Maxima との簡単なインターフェイス作製

この節では簡単な Maxima と Octave のインターフェイスを作製しましょう。最初に、Octave と Maxima の行列の定義方法を確認しておきましょう。

— Octave と Maxima の行列の定義を比較 —

```
mat1 = [1, 2, 3; 4, 5, 6]; ⇔ mat : matrix([1, 2, 3], [4, 5, 6]);
```

違いは、Maxima では行を大括弧で括るのに対して、Octave ではセミコロンで区切り、Maxima では `matrix` 命令を用いるが、Octave では直接記述し、割当てが Octave では `=` で、Maxima はコロンである事が判ります。

ところで、Maxima の行列データをリストに変換すると、Octave の行列にもっと似てきます。この場合、`matrix(...)` が `[...]` になります。その為、行ベクトルの場合は、Maxima の行ベクトルと Octave のデータは一致します。又、Octave では `[[1, 2, 3], [4, 5, 6]]` は `[1, 2, 3, 4, 5, 6]` の様に一つの行ベクトルとして評価されます。この事から、Maxima から Octave に変換する為には、Maxima 上で行列を配列に変換し、その行列の大きさを示すデータと一緒に Octave の m-file として記述しておけば、後は Octave で m-file 名を実行すれば行列が行ベクトルとして定義され、それを本来の行列となる様にプログラムも起動する様にしてしまいましょう。

先ず、Maxima のプログラムを示します。

```
SIZEofMAT(mat) :=
block(
  [ans:false],
  if matrixp(mat) then
    ans:map(length, map(lambda([x], substpart("[", x, 0)),
```

```

        [col(mat,1),col(transpose(mat),1)])
    else
        false,
    return(ans)
)$

m2oct(filename,x):=
block(
    [size,tmp,tmp1],
    if matrixp(x) then
        (
            tmp:substpart("[",x,0),
            size:SIZEofMAT(x),
            tmp:[filename,'maximat_size=size','maximat=tmp'],
            apply(stringout,tmp)
        )
    else
        print("Please enter MATRIX data!")
)$

```

このプログラムは、単純に、計算した行列の大きさを変数 `maximatrix_size` に入れ、行列は一つの行ベクトルを生成する Octave の M-file を生成するものです。

これに対し、Octave の Maxima で生成した行ベクトルを行列に変換する変換プログラムと、Octave で生成した行列を Maxima の命令ファイルに変換するプログラムを示します。

```

function [z] = m2octmat(maximat,maximat_size)
m=maximat_size(1);
n=maximat_size(2);
for i=[1:m]
    z(i,[1:n])=maximat([1:n]+(i-1)*n);
end;
end;

function []= oct2maximat(fname,mat)
[m,n]=size(mat);

```

```
id=fopen(fname,'w');
fprintf(id,'%s\n',"maximat:matrix(\n");
for i=[1:m]
    fprintf(id,'%s%22.15e',"[,mat(i,1));
    if n>1
        fprintf(id,', %22.15e',mat(i,[2:n]));
    end;
    if i!=m
        fprintf(id,'%s\n',"],");
    else
        fprintf(id,'%s\n',""]);
    end;
end;
fprintf(id,'%s\n',"");
fclose(id);
end;
```

これらのプログラムは非常に初歩的なものですが, 最低限の事が出来ます.

第16章 KNOPPIX/Math

この本にはKNOPPIX/Math2006が収録されています。このCDを通常のPC(PC/AT互換機)のCD-ROMドライブに入れて立ち上げると、KNOPPIXと呼ばれるLinuxが立ち上がります。このKNOPPIX/Mathの面白い点は、膨大な数学アプリケーションが附属し、それらが動作する環境がDVD-ROM一枚で得られる事です。

KNOPPIX/Mathには幾つかのアプリケーションの日本語の解説や附属文書等が収録されていますが、アプリケーションが多過ぎて戸惑うかもしれません。

この章では、KNOPPIX/Math 2007に収録されたアプリケーションの簡単な紹介を行います。KNOPPIX/Mathを楽しむ為の道標となれば幸いです。

16.1 KNOPPIX/Mathについて

KNOPPIXは1枚のCD(或いはDVD)からブート可能なLinuxです。他のLinuxディストリビューションと異なり、PCにインストールする手間どころか、PCのHDDに触る事さえも不要で、非常に簡単にLinux環境が使えるシステムです。

MaximaはKNOPPIX/MathとEduに収録されていますが、特に、KNOPPIX/MathにはMaxima以外にも様々な数式処理や可視化ツールが入っているので、手っ取り早くMaximaやその他の楽しい仲間達で遊ぶには最適のシステムです。

先ず、KNOPPIX/Mathのデスクトップ環境はデフォルトでKDEと呼ばれる環境が利用されています。このKDEはWindowsと似た使い方が可能です。

画面の左にはフォルダやファイルが並んでいますが、その中で一番上に数学デモと名付けられたフォルダがあります。このフォルダにKNOPPIX/Mathに関連する様々な文書やデモファイルがあります。先ずはKNOPPIX-Math-j.htmlを参照して下さい。収録アプリケーションや文書について概要等の情報があります。

16.2 KNOPPIX/Math に含まれている数式処理

ここでは簡単に KNOPPIX/Math に含まれている数式処理の仲間を紹介しましょう.

— KNOPPIX/Math に付属の数式処理 —

AXIOM[59]	汎用の数式処理システム. Maxima と同様に Common Lisp で記述されています.
GAP[61]	整数論向けの数式処理システム.
Macaulay2[62]	可換代数, 代数幾何学向けの数式処理システム. SINGULAR のライブラリを利用しています
PARI/GP[68]	整数論向けの数式処理システム. C で利用可能なライブラリを持っています.
Risa/Asir[70]	国産の汎用数式処理システム. 多項式の処理は非常に高速です.
SINGULAR[72]	可換代数, 代数幾何学向けの数式処理システム.

これらの数式処理は各々個性的なもので, Maxima を使っていて不自由を感じた場合に試してみると良いでしょう.

これらの数式処理の中で Risa/Asir, PARI/GP と GAP は Maxima に比較的似ていますが, SINGULAR と Macaulay2 は最初に環を定義する必要があり, 考え方が Maxima と随分異なります.

この SINGULAR と Macaulay2 を利用する為には可換環の知識が多少必要になります. 但し, この本の内容で最低限の利用が可能でしょう. 後は代数学の教科書を眺めながら, 色々な命令を試すのも楽しいかと思います. 又, surf.lib を用いて代数曲線や曲面を描く道具にする軟派な使い方も可能です. 特に SINGULAR はオンラインマニュアルが充実しています. この SINGULAR のオンラインマニュアルは?; と入力すると立ち上がります. 後はオンラインマニュアルで気の向くままに遊ぶと良いでしょう.

Macaulay2 は SINGULAR で用いられているライブラリを利用している為, SINGULAR の事が多少判っていれば, こちらも似た使い方が出来てしまいます. 言語的には Macaulay2 の方が色々面白いシステムです. ヘルプも充実していますが, HyperText を用いた SINGULAR のマニュアルと比べて, 単純にテキスト表示を行うものの為, あまり親切ではありません. Macaulay2 を利用する場合には emacs や TeXmacs をフロントエンドにして利用する事を強く薦めます.

SINGULAR と Macaulay2 には, 英語ですが Greuel-Pfister [50] と Schenck [54] といった読み易い本があります. どちらも可換環の教科書として使えますが, Schenck の本の

方が薄くて読み易いと思います。

Risa/Asir は Maxima と同様の汎用数式処理システムで、多項式の処理が非常に高速である事の特徴とします。この Risa/Asir は GPL とは別のライセンスに基くアプリケーションです。その為、Risa/Asir 自体は KNOPPIX/Math に収録されていません。但し、メニューバーにある $\sqrt{x}$ メニューに含まれている OpenXM から asir を最初を選択するとインストーラが立ち上がります。因にインストーラでは wget 命令が用いられています。その為、外部への接続で PROXY を用いられている環境では、予め、.wgetrc ファイルをホームディレクトリ上で作製し、PROXY の情報を書込んで置く必要があります。例えば、http と ftp の proxy が www.mikeneko.org でポート番号が 8080 であれば以下の内容を記述します。

— .wgetrc ファイルの内容 —

```
http_proxy=http://www.mikeneko.org:8080/  
ftp_proxy=http://www.mikeneko.org:8080/
```

こうする事で、Risa/Asir のインストーラが上手く動作し、Risa/Asir がインストールされます。尚、このインストーラで入手したファイルは全て /home/knoppix ディレクトリ¹の下に .asir-tmp ディレクトリに収納されています。

Risa/Asir を一度インストールすれば、KNOPPIX/Math の電源を落す迄は Risa/Asir の通常の利用が可能です。必要があれば USB メモリ等で保存しておくといいでしょう。Risa/Asir の処理言語は C と殆ど同じです。オンラインマニュアルも充実しています。KNOPPIX/Math の数学デモフォルダの PDF フォルダには Risa/Asir ドリル、2003 (asir-book.pdf) があります。

PARI/GP は数論向けの数式処理システムで、膨大な関数を持っています。PARI/GP は数論向けのライブラリが PARI で、gp は PARI のライブラリを利用するフロントエンドです。この gp は非常に地味ですが、意外と便利なフロントエンドです。尚、PARI のライブラリは Risa/Asir でも利用されており、Risa/Asir から PARI の関数を呼出す事が可能です。こちらも巨大なマニュアルが標準で付属しており、ドキュメント (Cohen [48] 等) も揃っています。

¹KNOPPIX/Math を HDD にインストールし、knoppix 以外のユーザー id でログインした場合、そのユーザー id に対応するホームディレクトリになりますが、KNOPPIX/Math 側でインストールしたファイルが /home/knoppix 下のディレクトリに保存される事を前提にしている為、リンク先が存在しない事になるので注意が必要です。

16.3 可視化ツール

KNOPPIX/Math には可視化ツールも揃っています. この本で取り上げた, gnulot, Geomview [83] や surf [84] に加えて, dynagraph [81] というツールもあります. この dynagraph は Maple V のグラフ表示機能を独立させた様なツールで, 構文や使い方は Maple V のグラフ表示機能と殆ど同じものです.

気楽に楽しめるアプリケーションとしてマンデルブロー集合をカラフルに描く XaoS [85] があります. こちらの使い方は簡単で, XaoS を実行して現われる XaoS の画面上で, `[a]` と入力して下さい. 勝手にズームして行きます. それから 0 から 9 までの数字を入力しましょう. すると, マンデルブロー集合が切替えられます. 終了は同様に, `[q]` と入力すると, 終了か継続かを聞いて来るので, そこで, `[1]` を入力すれば終了します. XaoS でカラフルな絵を楽しんで下さいね.

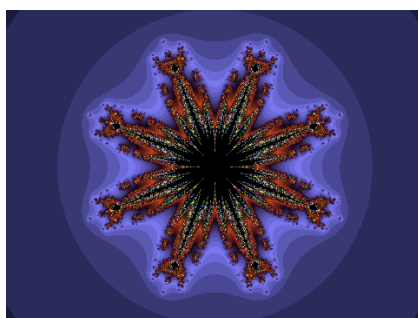


図 16.1: XaoS の画面.6 を入力した場合

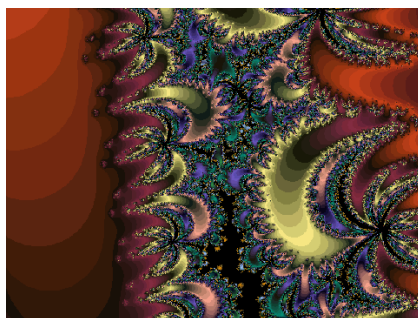


図 16.2: XaoS の画面.a を入力し, 只今, 拡大中…

16.4 MATLAB もどき

KNOPPIX/Mathには幾つかのMATLAB風の数値行列処理のアプリケーションが含まれています。MATLAB風のシステムとしてOctave [75]とC風のYorick [77]の二つがあります。先ず、この本でも解説したOctaveはMATLABのクローンと言える程、MATLABとの互換性が高いシステムです。但し、ライブラリに関してはMATLABに劣ります。その一方でファイルの処理ではOctaveの方が柔軟に対処出来たりします。その一方でYorickは非常に個性的なシステムです。言語的にはCに近く、対話処理の可能なCと言える程です。Octaveと比較して、処理は非常に高速です。このYorickで面白いのはPLPlotライブラリを使って割と派手なグラフ表示が可能な事です。このグラフィックスはyorickを立ち上げて、付属のデモを実行すると楽しめます。派手なデモは太鼓の振動のdemo2、機構のカオス的な運動を解析するdemo3、羽根の回りの流体を計算するdemo4とオブジェクトを回転させたりするdemo5があります。これらのデモの実行は、`include,"demo2.i"`の様に入力すると、デモファイルdemo2.iが読込まれます。デモの実行は`demo2`と入力するだけです。

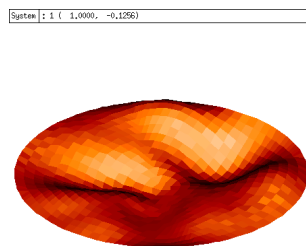


図 16.3: Yorick の demo2 を実行している様子

尚、demo3の実行を古いPCで実行する事は薦めません。非常に重い処理になります。尚、これらのデモファイル自体がYorickのアニメーションや計算の実例となっています。非常に残念な事に現時点でまとめた文書は付属のマニュアル程度しかありません。ライセンスの関係でKNOPPIX/Mathに収録出来されていないものにSciLab [76]があります。こちらは非常に高機能で、MATLABにもそこそ似ています。但し、Octave程、MATLABに似ておらず、独自の面もあります。どちらかと言えば、MATLABがCとするなら、SciLabがFORTRANを感じさせる面があります。但し、この本のOctaveの章で説明した程度の事であれば、殆どそのまま利用可能です。

SciLab の凄い所は、付属のライブラリや各種ツールの充実ぶりです。実際、MATLAB の Simlink の様に GUI を用いて制御系が構成出来るツールの SCICOS が付属している程です。この様に SciLab に関しては、下手な商用ソフトウェアも太刀打ち出来ない程に充実しています。この SciLab には Windows 版や Linux 版等のバイナリも存在する為、数値行列の計算や制御系の解析を行われる方は、是非、入手される事を薦めます。

16.5 対話的幾何学ソフト KSEG

KNOPPIX/Math には対話的に処理が行える幾何学ソフトウェア KSEG [82] が収録されています。KSEG は非常にユニークなソフトウェアです。これと似たもので、より本格的なものに Cinderella [80] があります。この Cinderella は商用のもので、シュプリンガーから日本語の書籍が幾つか出版されています。

KSEG は機能的には Cinderella と比べてやや劣りますが、初等平面幾何学の問題で遊ぶには十分過ぎる機能を持っており、その上、軌跡を描く事も出来ます。KSEG は最初に点を定義して、それから線を描くという非常に直感的な方法で幾何形状を構築します。更に、KSEG では不必要な対象を隠す事も可能です。

それでは KSEG を使って簡単な図形を描いてみましょう。KSEG で点はマウスの右ボタンをクリックすると、マウスポインタが置かれた個所に点が生成されます。生成した点等の対象はマウスポインタをその対象に移動させてマウスの左ボタンを押すと選択されます。選択されると対象は赤く表示されるので判ります。線分や円等の生成で複数の対象を選択しなければなりません、その場合はシフトキーを押しながら、対象を選択すると良いでしょう。

先ず、最初に円を生成しましょう。その為には円の中心点と円上の点を生成しなければなりません。ここではシフトキーを押しながら右ボタンをクリックして二つの点を生成します。それから、メニューバーから円の生成を選択します。次に円上の点を 3 個生成しましょう。これはマウスポインタを円の上に移動させて右ボタンをクリックすると円上に拘束された点が生成されます。この操作の後に、円上の三点と円の中心点をシフトキーを押しながら、マウスの左ボタンを押して選択します。それから線分を生成しましょう。KSEG では線分の長さ、角の角度、多角形の面積等の計算が行えます。ここでは円周角の定理を見る為に、角度を表示させましょう。すると図 16.4 に示す様になっている筈です。

KSEG で生成した対象には親子関係があります。例えば、円の生成で用いた中心点と円周上の点をマウスの左ボタンを押しながら動かすと円の大きさが変化しますが、それ以外の円や円周上の点を選択して、マウスを移動させると全体が動きます。

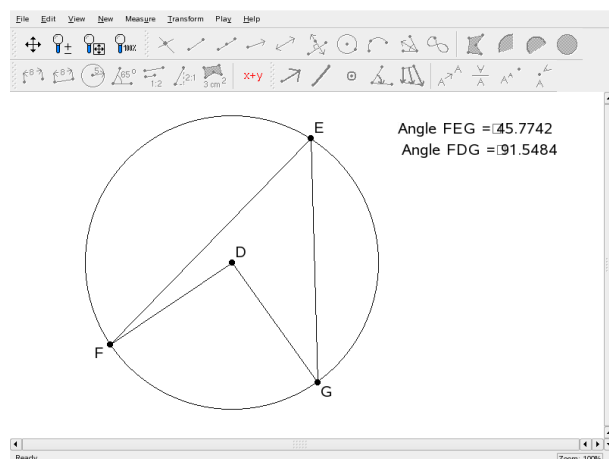


図 16.4: KSEG の画面. 円周角の定理

KSEG のファイルをこの様にして見ても構築の方法は簡単には判りません. KSEG で面白い点はコンストラクションと呼ばれる幾何形状生成の構成手順があり, コンストラクションに変換する事で, 図形の構成手順が容易に判ります. 図 16.5 に先程の円周角のコンストラクションを示します.

コンストラクションはファイルメニューのコンストラクションとしてコピーを選択する事で描画の構成順序が右側に表示されます. KSEG による図形の構成はコンストラクションに基いて行われます. 幾何形状の構築はウインドウ右側のコンストラクションの上から順に実行されています. 因に, 各コンストラクションをクリックすると左側の図形で対応する対象が赤くなる事で, 構成の方法が判ります. 別ウインドウを生成して, このコンストラクションに従って図形を構築すると良いでしょう. 更に, KSEG ではこのコンストラクションを上手く利用する事で図形を自動生成する事も可能になります. 詳細はヘルプメニューから日本語のヘルプを参照して下さい.

このツールは単純な幾何学遊びにしか使えない様に見えますが, 様々な部品を組合せた機構の解析でも十分に利用可能と私は思っています.

例えば, 図 16.6 と図 16.7 にスライダー・クランク機構と呼ばれる機構の軌跡を描かせています. このモデルでは二つの部品に相当する線分 S3 と線分 S4 の長さは下の二本の線分の長さで作成した為に, これらの線分の長さで制御可能となっています.

図 16.6 では線分 S4 の方が線分 S3 よりも長い為, 線分 S3 を回しても問題がありません.

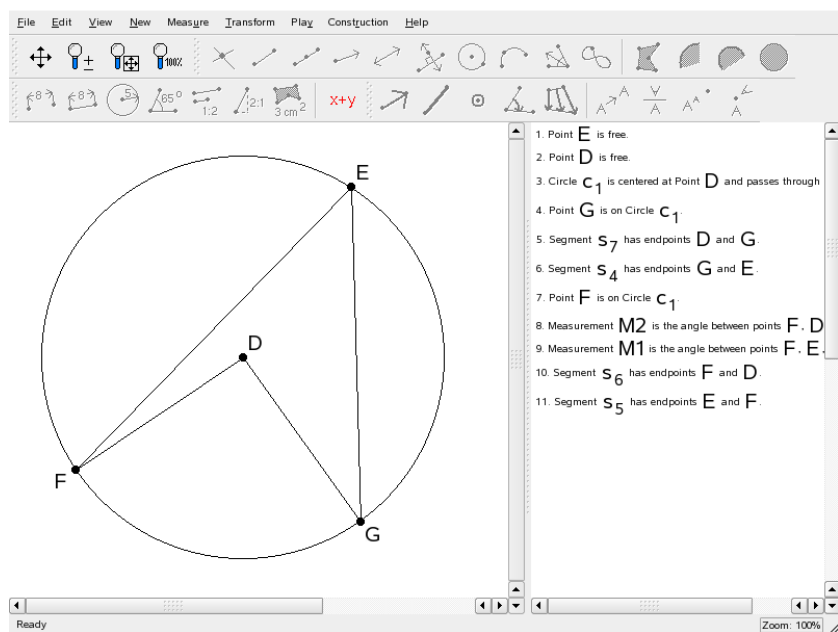


図 16.5: KSEG のコンストラクション

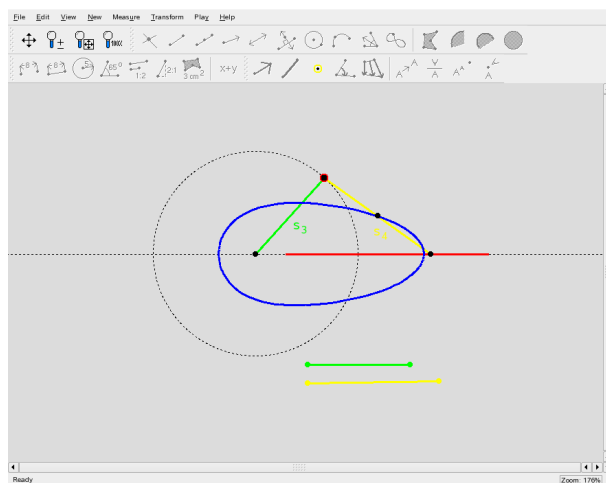


図 16.6: スライダークランク機構 (その 1)

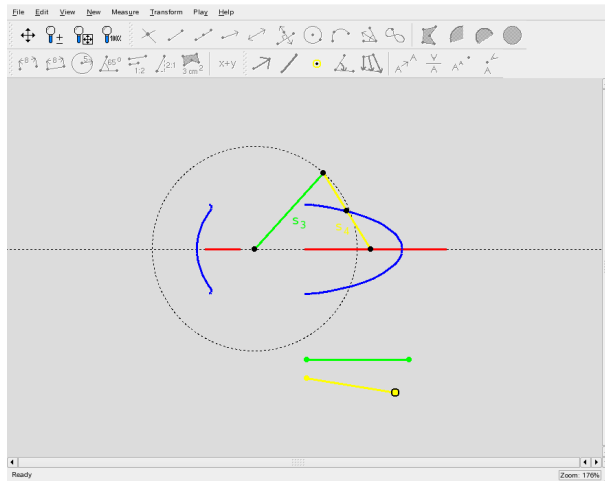


図 16.7: スライダークランク機構 (その 2)

図 16.7 の様に線分 S4 の方が線分 S3 よりも短くなると, 幾何学的に成り立たない箇所が発生する為, 軌跡が途切れています.

KSEG の例題は KNOPPIX/Math に数多く含まれているので、是非試してみてください。このソフトウェアに関しては、実際に動かしてみる事が一番の理解への近道です。数学デモフォルダの中に `kseg_sample` という名前のフォルダに幾つかあります。ファイルをクリックすると KSEG による描画を見る事が出来ます。そして、ファイルメニューからコンストラクションとしてコピーを選択し、コンストラクションを調べる事で、構築手順を確認すると良いでしょう。

16.6 Maxima の例題の実行

14 章の `surfplot` や 13 章で解説した `calcAlexanderPoly` といったプログラムは `/cdrom/Math/MaximaSomeExamples` に収録されています。デスクトップの左最上段にある数学デモフォルダの中からも辿る事が可能です。尚、これらのプログラムはこの本に収録したものよりも少し古い版ですが、動作に決定的な違いはありません。尚、`surfplot` 函数は Maxima のカレントディレクトリ (=Maxima を起動したディレクトリ、KNOPPIX のメニューから起動した場合は `/home/knoppix` になります) に `surf` に引渡すスクリプトファイル `surf.tmp` を書込む仕様となっています。その為、コンソールから Maxima を立ち上げる際に、`MaximSomeExamples` ディレクトリに移動した場合、`surf.tmp` ファイルの書込みが出来ないので、`surfplot` 函数が動作しない問題が生じます。その為、`/home/knoppix` に `/cdrom/Math/MaximaSomeExamples/` 以下のファイルの複製を置くと良いでしょう。この場合はメニューから `maxima`, `xmaxima` や `wxmaxima`, それに TeXmacs から Maxima を呼出しても、`/home/knoppix` に置かれた Maxima の初期化ファイルの `maxima-init.mac` の内容が実行されるので、`surfplot` 函数や `calcAlexanderPoly` 函数が直ちに利用可能となります。

勿論、この本の `surfplot.mc` や `AlexanderPoly.mc` を直接打ち込んでも構いませんし、複製とこの本のプログラムの違いを確認するのも良いでしょう。

後は、この本の 14 章の代数曲線や曲面の例題や 13 章の例題を試す事が出来ます。

第17章 Maximaのインストール

この章では,KNOPPIX/MathをVMWare PlayerやcoLinuxを用いてKNOPPIX/Mathを立ち上げる方法,それとは別に,Windows版,Linux版のMaximaをインストールする方法について簡単に述べます.

猶,Linux以外のUNIX系OSの場合も,コンパイルに必要な環境を整えていれば,Linux版の方法でインストールが可能です.

17.1 道標

最初のFAQで述べた様に,a.我慢強い方,b.軟派な方,c.windowsで済ませたい方の三種類に読者の皆さんを分類しています.先ず,a.の方はソースファイルを入手してコンパイルしてみましょう.§17.5に概要を解説しています. bの軟派な方は早速,附属のKNOPPIX/Mathで遊んで下さい.説明は特に必要は無いでしょう.c.のMS-Windowsで全てを済ませたい方で,初心者の方は§17.4でインストール方法を解説しているので参考にして下さい.

猶,新奇なものの好きな方やKNOPPIXに興味のある方はVirtualBoxやVMware Playerを利用と良いでしょう.特に,b.の軟派な方で,MS-Windowsも一緒に使いたい方は,VirtualBoxやVMware Playerで遊んでみて下さい.VirtualBoxを使う場合は次の§17.2,VMware Playerを使う場合は§17.3を参照して下さい.

猶,附属のKNOPPIX/Mathには有益な文献や例題があるので,KNOPPIXを使う気が無くても,CD-ROMのMath-jaフォルダの内容を確認する事を薦めます.

17.2 VirtualBoxでKNOPPIXを利用する場合

17.2.1 VirtualBoxの概要

VirtualBoxはドイツのInnotek社が作成した仮想計算機ソフトです.VirtualBoxにはソースファイルのOpenSource版とバイナリ版の二種類があります.OpenSource版は

GPL version 2 に基いて配布されています。バイナリ版に関しては、個人的利用と評価目的に対しては無償利用が可能となっています。

自分で最初から構成したい人は OpenSource 版で良いでしょう。但し、手軽にバイナリを入手して遊びたい人は、評価版としてバイナリを入手する事になるでしょう。VirtualBox は <http://www.virtualbox.org/wiki/Downloads> から入手出来ます。猶、バイナリ版としては、MS-Windows,x86 及び AMD64 環境の LINUX と OpenSolaris、β 版ですが、MacOS X 環境に対応したものがあります

猶、後述の VMware Player とは違い、VirtualBox の場合は GUI 操作のみで処理を行う事が可能です。その為、全くの初心者が仮想計算機環境を構築する事が非常に容易になっている事も、VirtualBox の長所として挙げておきます。

以下では、VirtualBox が既にインストールされていると仮定して解説を行います。バイナリ版のインストール自体は非常に容易なのでここでは解説はしません。

17.2.2 仮想計算機の設定

仮想計算機を利用する為には、当然の事ですが、仮想計算機を生成しておかなければなりません。VMwarePlayer の場合、Player という性格上、仮想計算機の生成は行えない為、利用者が QEMU を用いて別途生成しておく必要があります。

VirtualBox の場合、VirtualBox と独立して仮想計算機を生成する必要は無く、VirtualBox 内部で、GUI を用いた処理を行う事で済みます事が可能です。

では、VirtualBox を最初に立ち上げてみましょう。

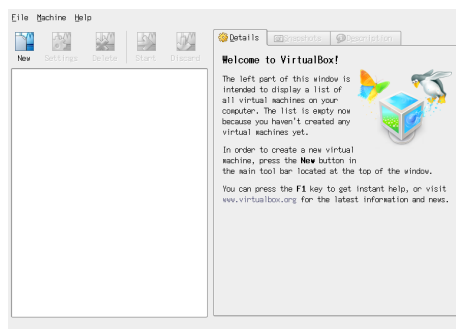


図 17.1: VirtualBox のウィンドウ

すると図 17.1 の様なウィンドウが現れます。このウィンドウのメニュー群の下にアイコンが幾つか並んでいると思います。基本的な操作は、これらアイコンを押すだけ

で, Wizard 形式で処理を進めて行く事が可能です.

さて, 仮想計算機の生成は, 図 17.1 の左上に並んだアイコンの  を押します.

すると, 図 17.2 に示す Wizard が起動します. 基本的に流れ作業で仮想計算機の生成が簡単に行えます.

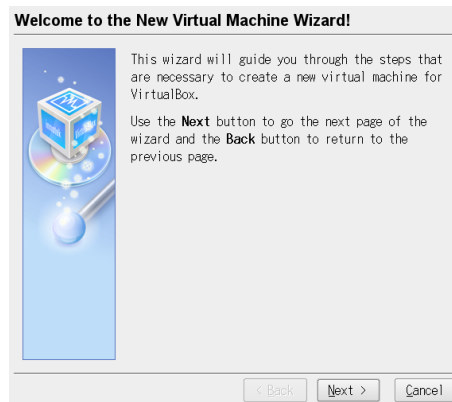


図 17.2: 仮想計算機生成 Wizard

17.2 の右下の **Next >** ボタンを押すと図 17.3 の画面に進みます. ここでは仮想計算機の名称と OS を選択します.

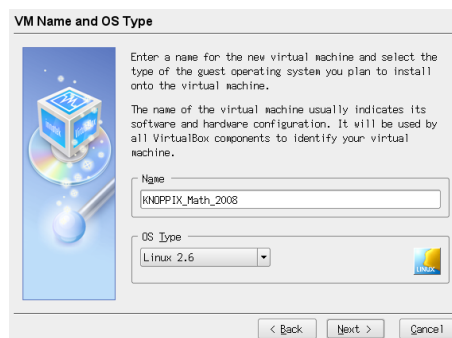


図 17.3: 仮想計算機の名称と OS の設定

ここでウィンドウ右下の **New >** ボタンを押せば, 図 17.4 に移り, 仮想計算機のメモリの容量の設定が行えます.

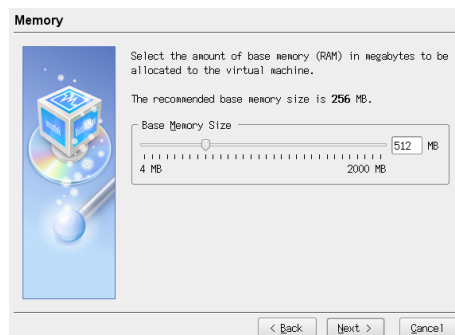


図 17.4: 仮想計算機のメモリの設定

メモリは環境によって様々なので一概に言えません。メモリに関しては可能な限り大きく設定するのが望ましいのですが、KNOPPIX/Math では最低 128MB、KDE の様なデスクトップ環境を用いるのであれば最低 256MB、500MB 以上を設定していれば、快適に使えると思います。

メモリを指定すると、ウィンドウ右下の **New >** ボタンを押して図 17.5 に示すハードディスクの設定に移ります。

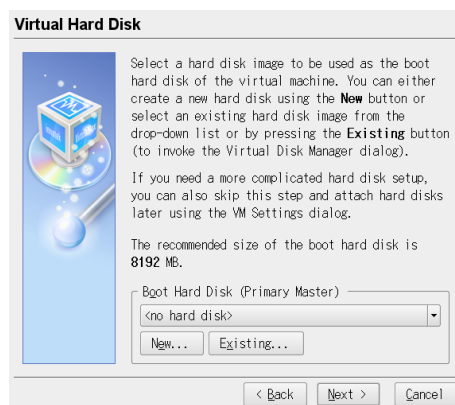


図 17.5: 仮想計算機のハードディスクの設定

KNOPPIX だけを起動させる場合、このハードディスクは不要です。KNOPPIX で遊ぶ為に、仮想計算機側に必要なものは、仮想計算機に割り当てられるメモリと KNOPPIX の DVD/CD-ROM を読込む為の DVD/CD-ROM ドライブ、或いは DVD/CD-ROM

の ISO イメージファイルのみです。ここで、ハードディスクを多少でも設定しておけば、KNOPPIX の計算結果や諸環境を保存したり、KNOPPIX で swap の利用が可能になる等と、後の処理が何かと楽になるのも事実です。

更に、ハードディスクの大きさを大き目に指定していても、ハードディスクのイメージファイルの大きさを固定しておかなければ、必要に応じてファイルが大きくなる様に指定出来ま。す勿論、ハードディスクは後でも設定可能なので、ここでの指定はあまり神経質にならなくても構いません。

初心者で、この様な設定が不安であれば、ここの設定を飛ばしても問題ありません。この場合、何も設定せずに、**Next >**を押します。すると、図 17.6 のウィンドウが出ます。

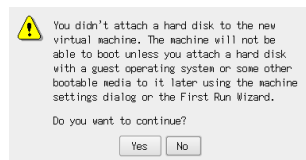


図 17.6: ハードディスクイメージを生成しない場合のメッセージ

この警告の内容は特に気にする必要はありません。ここで**Yes**を押すと、図 17.7 のウィンドウに切り替わり、**Finish**を押せば仮想計算機の生成が終了します。

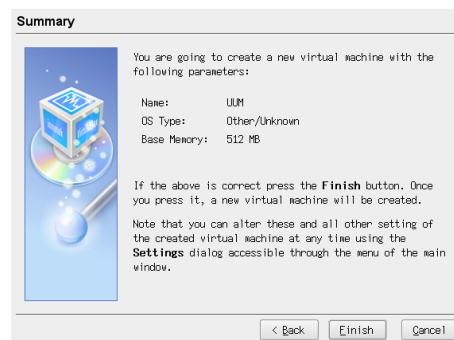


図 17.7: 仮想計算機のハードディスク無しの環境

一方で、ハードディスク新規に作成する場合、図 17.5 の**New...**ボタンを押しましょう。すると、図 17.8 の仮想ハードディスク生成の Wizard に移動します。



図 17.8: 仮想ハードディスク生成の Wizard

このウィンドウで **New >** ボタンを押すと、今度は図 17.9 に移動し、ここでハードディスクのイメージファイルの種類を指定します。一番上の動的に大きくなる様に指定しておけば、未使用の領域で無駄に膨れ上がる事の無いので、実用的かと思います。

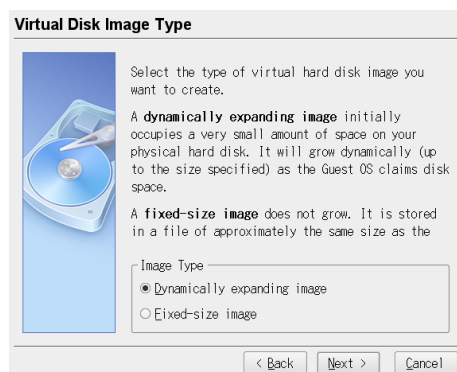


図 17.9: 仮想ハードディスクイメージの型を指定

さて、ハードディスクイメージファイルの型を指定して、**New >** ボタンを押しましょう。

すると, 図 17.10 に移動し, 今度は仮想計算機のハードディスクのイメージファイルの大きさを指定します.

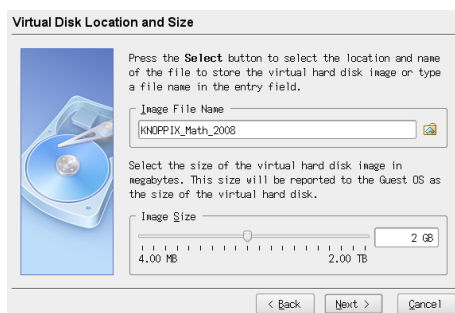


図 17.10: 仮想ハードディスクイメージファイルとその大きさを指定

設定が終ると, **New >** ボタンを押しましょう.

すると, これから生成する仮想計算機の概要が図 17.11 に示す様に表示されます.

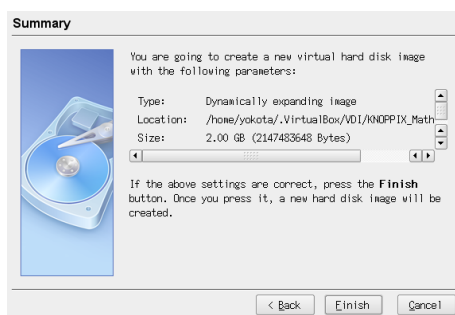


図 17.11: ハードディスクイメージファイルの概要

この設定で良ければ, **Finish** を押しましょう.

すると仮想ハードディスクが生成され、ウィンドウは図 17.12 に示すものになります。



図 17.12: 仮想計算機のハードディスクの指定後

さて、これで仮想計算機のハードディスクのイメージファイルが指定されました。これで良ければ、**Next >** ボタンを押しましょう。

すると図 17.13 に示すウィンドウに移り、このウィンドウの右側には生成した仮想計算機の概要が表示されています。

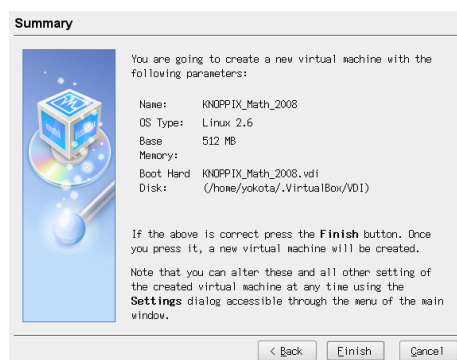


図 17.13: 生成した仮想計算機の概要

これで良ければ **Finish** を押します。

これで Wizard は終了し, VirtualBox の画面は図 17.14 に示す様に新たな仮想計算機名が追加されている筈です。

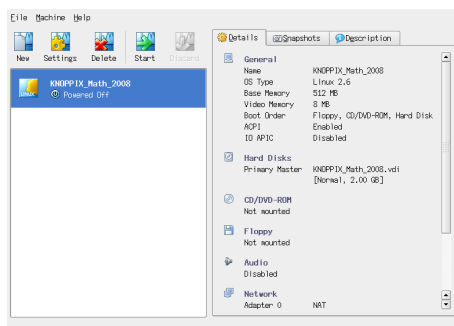


図 17.14: Virtualbox の画面

さて, 今度は周辺機器の設定を行いましょう. そこで, 左側のリストから仮想計算機を選択しましょう. すると右側のウィンドウに仮想計算機の概要が出ているので, CD/DVD-ROM の箇所をクリックします.

すると, 図 17.15 の画面に切り替わります.

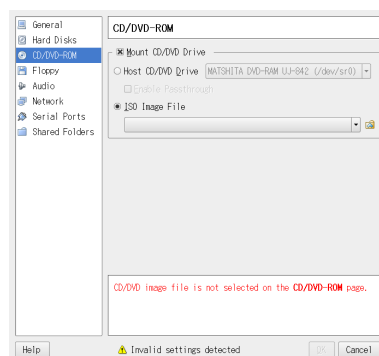


図 17.15: 仮想計算機の CD/DVD-ROM の設定画面

ここで KNOPPIX/Math は CD-ROM/DVD-ROM, 或いは ISO イメージファイルとして提供されます. VirtualBox では, VMwarePlayer と同様に ISO イメージファイルを用いる事も可能です. ISO イメージファイルを用いる利点は, CD-ROM ドライブを用いるよりも読込速度が格段に速い点, Net から入手する場合は ISO イメージファイルと

なる事が挙げられます。但し,ISO イメージファイルではハードディスクや USB メモリ等の記憶装置の容量を消費するのが欠点といえは欠点です。

この点は、貴方の利用している環境に従って、最適なものを選択すると良いでしょう。今回は ISO イメージファイルを選択しましょう。そこで,ISO Image File のラジオボタンにチェックを入れ、その右下にある  アイコンを押すと図 17.16 の画面に移動します。

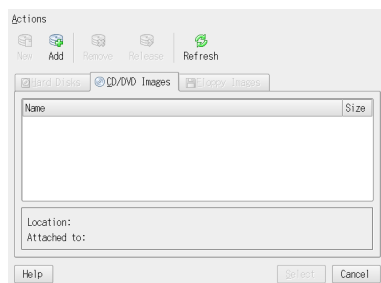


図 17.16: 仮想計算機の CD/DVD-ROM のイメージファイルを選択

このウィンドウでは上に並んだアイコンを押す事で、仮想計算機のハードディスクやフロッピー等のイメージファイルの生成、指定、開放や削除が行えます。

ここで左上にある  アイコンを押せば、仮想計算機にメディアのイメージファイルが設定出来ます。今回は CD/DVD なので,ISO イメージファイルの選択になります。図 17.17 にイメージファイルを選択した後の様子を示しておきます。

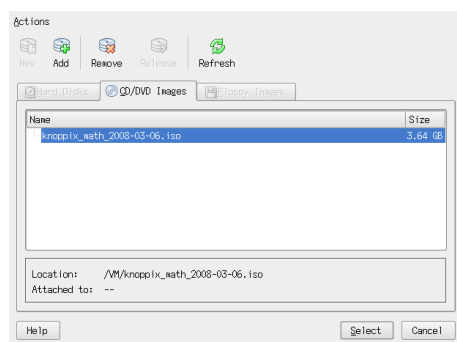


図 17.17: 仮想計算機の CD/DVD-ROM のイメージファイルの選択後

これで良ければ **Select** ボタンを押して図 17.18 に戻ります。

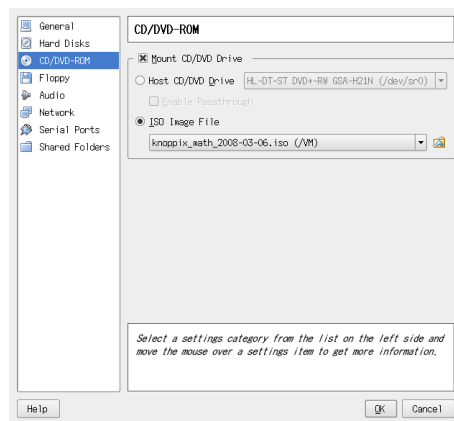


図 17.18: 仮想計算機の CD/DVD-ROM 設定画面

そこで **Ok** を押しましょう。すると、図 17.19 に示す内容に切り替わっている筈です。

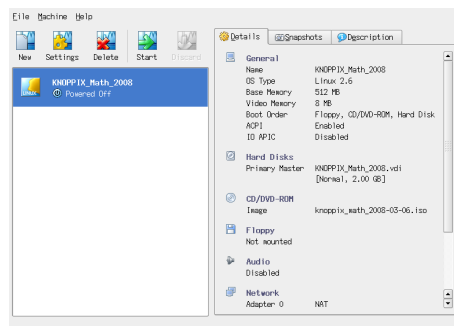


図 17.19: 仮想計算機の設定内容 (最終版)

これで準備が完了しました。

17.2.3 仮想計算機の起動について

この様に,VirtualBox では直接設定ファイルを弄る必要はありません. 更に, 仮想計算機の起動も非常に容易です.

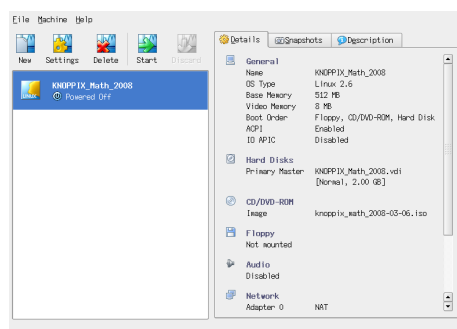


図 17.20: 仮想計算機の起動

仮想計算機の起動は, 図 17.20 の左側のリストから仮想計算機を指定し, それから, 左上の  アイコンを押せば, 指定した仮想計算機が別ウィンドウで起動します. 例えば, 図 17.21 に KNOPPIX/Math 2008 の起動画面を示しておきます.

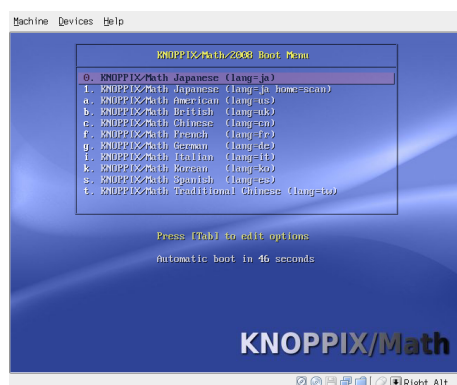


図 17.21: 仮想計算機の起動の様子

最後に, 図 17.22 に KNOPPIX/Math 2008 の起動中の様子を示しておきます.

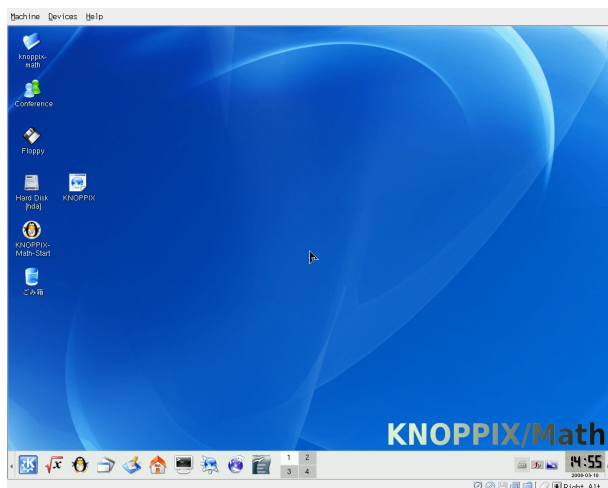


図 17.22: 仮想計算機の起動の様子

さて,VirtualBox の仮想計算機にマウスを移動させると, 図 17.23 に示すウィンドウが出ます.

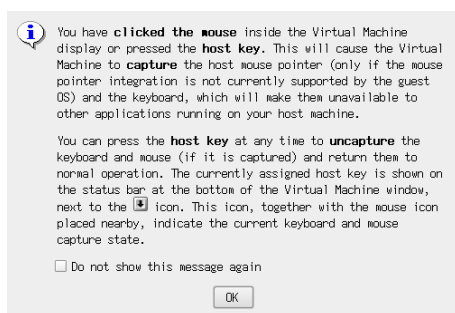


図 17.23: マウスポインタの移動について注意事項

これは,VirtualBox に一旦マウスのポインタを, 仮想計算機のウィンドウをクリックする事で移動させると, 特定のキーを押すか, 仮想計算機を停止しなければ, マウスのポインタが移動しない事を知らせるものです.

このキーの設定は仮想計算機の右下の枠にあるアイコン  の右側に,  Right Alt の様に表示されています

このキーの設定は VirtualBox のファイルメニューの **Preferences...** から変更可能です. 図 17.24 に preferences... を選んで現われたウィンドウの様子を示しておきます.

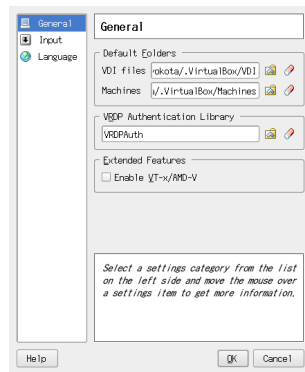


図 17.24: VirtualBox 一般の設定

ここで左側の項目で **Input** を選択すると図 17.25 に切替わります. ここでキーの設定が可能です.

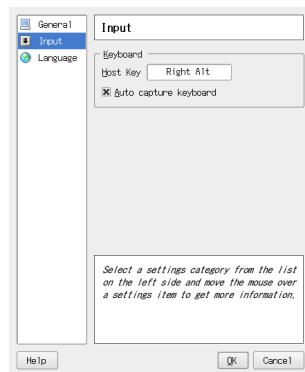


図 17.25: VirtualBox のマウスポインタ切替キーの設定

この様に, VirtualBox は MS-Windows で利用されるアプリケーションの設定方法と同様に設定が行え, その設定も基本的に GUI ベースなので, 後述の VMware Player よりも簡単に設定が行えるのが長所です.

17.3 VMware Player で KNOPPIX を利用する場合

17.3.1 VMware Player のインストール

VMware Player を用いると、MS-Windows や Linux 上で KNOPPIX/Math が使えます。幾つかのソフトウェアを準備する必要がありますが、簡単に KNOPPIX が利用可能な環境の構築が行えます。猶、UNIX 環境では root、Windows 環境では管理者権限を持つユーザーになって作業を行います。

先ず最初に VMware Player を入手してインストールします。VMware Player は <http://www.vmware.com/products/player/> から入手出来ます。VMware Player は無償で利用が可能です。但し、VMware Player には仮想計算機を生成する為のツールは一切付属しません。又、MS-Windows を仮想計算機にインストールした場合、VGA 以外の解像度はドライバが付属していない為に使えません。

但し、Linux を仮想計算機上で用いる場合には問題はありません。更に、仮想計算機の生成も GPL で配布されている QEMU を用いれば生成可能です。

さて、VMware Player のインストール後に VMware Player の初期設定を行う必要があります。Linux 版の場合、`/usr/bin/vmware-config.pl` を起動して `vmplayer` を立ち上げる前に設定を行いますが、デフォルトの設定で十分です。猶、Linux でカーネルのアップグレードを行った場合、必ず `vmware-config.pl` を起動して設定を行う必要があります。猶、MS-Windows の場合は GUI 操作でより簡単に行えます。

17.3.2 仮想計算機の生成

VMware Player で遊ぶ為には、KNOPPIX/Math の CD-ROM か ISO イメージファイルが必ず必要です。他に準備するファイルとして `vmdk` ファイルと `vmx` ファイルの二種類があります。先ず、`vmdk` ファイルは VMware Player の仮想計算機のファイルで、このファイルには仮想計算機のディスクも含まれています。VMware Player で遊ぶ為には必ずこのファイルが必要です。但し、`vmdk` ファイルは VMware Player で生成する事が出来ません。そこで QEMU の `qemu-img` を用いて `vmdk` ファイルを生成します。QEMU 自体は <http://fabrice.bellard.free.fr/qemu/> から入手可能です。猶、QEMU の MS-Windows 版もあるので、MS-Windows 環境でも利用可能です。

猶、UNIX 環境で QEMU のソースファイルを入手してコンパイルする場合、最初に `./configure` で `configure` を実行し、`make` と入力するだけでコンパイルが出来ます。そして、最後に `make install` とする事で、`/usr/local/bin` に QEMU の実行ファイルがインストールされます。

vmdk ファイルを生成では QEMU の含まれる qemu-img 命令を用います. ここで vmdk ファイルとして knoppix-math.vmdk を生成しましょう. このファイルの生成では以下の設定で qemu-img 命令を使います.

```
gemu-img  
qemu-img create -f vmdk knoppix-math.vmdk 10G
```

最後の 10G は vmdk ファイルの上限になります. このファイルは必要に応じて大きくなります.

17.3.3 vmx ファイルの作成

vmdk ファイルと同じディレクトリ (フォルダ) に今度は vmx ファイルを記述します. この vmx ファイルには vmdk ファイルやイメージファイル等のファイルを指定したり, 計算機のデバイスの設定を行うファイルです.

先ず, CD-ROM から KNOPPIX/Math を立ち上げる場合, vmx ファイルには以下の設定を行います.

```
CD-ROM から立ち上げる場合の vmx ファイル例  
#!/usr/bin/vmware  
config.version = "8"  
ide0:0.present = "true"  
#qemu-img で生成した vmdk ファイルを指定  
ide0:0.fileName = "knoppix-math.vmdk"  
ide1:0.present = "true"  
memsize = "256"  
#CD-ROM から KNOPPIX を立ち上げる場合の設定  
ide1:0.fileName = "auto detect"  
ide1:0.deviceType = "atapi-cdrom"  
ethernet0.present = "true"  
ethernet0.connectionType = "nat"  
#VMware Player で立ち上げる OS 名  
guestOS = "other26xlinux"
```

この設定例では, CD-ROM から KNOPPIX を立ち上げる設定となっています. 勿論, KNOPPIX の iso イメージファイルを利用する事も可能です. この場合は, ide1:0.fileName

にイメージファイルを指定し、ide1:0.deviceType を cdrom-image にします。そして、CD-ROM イメージファイルは ide1:0.filename にフルパスで記述します。

— ISO イメージファイルを用いる場合の vmx ファイル例 —

```
#!/usr/bin/vmware
config.version = "8"
ide0:0.present = "true"
#qemu-img で生成した vmdk ファイルを指定
ide0:0.fileName = "knoppix-math.vmdk"
ide1:0.present = "true"
memsize = "256"
#ハードディスクに置いたイメージファイルから KNOPPIX
#を立ち上げる場合の設定
#ISO イメージファイルを指定. この場合, イメージファイルは
#/KNOPPIX ディレクトリ上の knoppix_v4.0.2-math-20060217-ja.iso
ide1:0.fileName = "/KNOPPIX/knoppix_v4.0.2-math-20060217-ja.iso"
ide1:0.deviceType = "cdrom-image"
ethernet0.present = "true"
ethernet0.connectionType = "nat"
#VMware Player で立ち上げる OS 名
guestOS = "other26xlinux"
```

因に、仮想計算機に MS-Windows 等の OS をインストールする場合は KNOPPIX/Math の CD-ROM による立上げの設定のままで、後は guestOS を win2000pro 等、OS に合せて修正して、その OS のインストールディスクを CD-ROM ドライブに設定しておけば、仮想計算機の立上げ時にインストーラーが立ち上がるので、その後は通常のインストールと同じ操作をすれば良いのです。

以上で VMware Player で遊ぶ為の準備は終わりました。

17.3.4 VMware Player で仮想計算機の起動

vmplayer を立ち上げると、最初に vmx ファイルを要求して来ます。そこで、必要に応じて vmx ファイルを指定します。ここで KNOPPIX を CD-ROM から立ち上げる場合、予め PC に CD-ROM を入れておきましょう。但し、OS が Linux の場合、CD-ROM を手動でマウントする必要はありません。立上げた仮想計算機側で PC の CD-ROM ドライブを認識して CD-ROM をマウントするからです。

この様にして VMware Player を使って KNOPPIX を立上げる事が出来ます。猶,vm-player で起動している KNOPPIX にマウスポインタを移したければ,vmplayer のウィンドウにポインタを移動させて,Ctrl+G を押すか,KNOPPIX のメニューや画面をマウスでピックすればポインタが移ります.vmplayer の外にマウスのポインタを移動させる場合は,Ctrl+Alt を押します。

ここで,KNOPPIX の X-Window System の画面の大きさを変更したければ, 図 17.26 に示す KNOPPIX の立ち上げ時で boot: プロンプトの個所に `knoppix screen=640x480` の様に入力します。

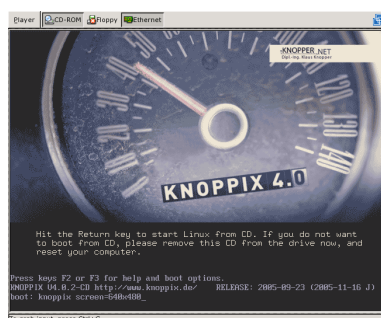


図 17.26: KNOPPIX/Math のチートコードの入力

この様に boot: プロンプトが出ている時に knoppix の後に続けて入力する命令の事をチートコードと呼びます。

KNOPPIX の立上げ中はチートコードによる画面の大きさは反映されませんが, X が立上がると指定した解像度になります。

例えば, 図 17.27 に ThinkPad X24 上で 640x480 の解像度で KNOPPIX/Math を起動させている様子を示しておきます。

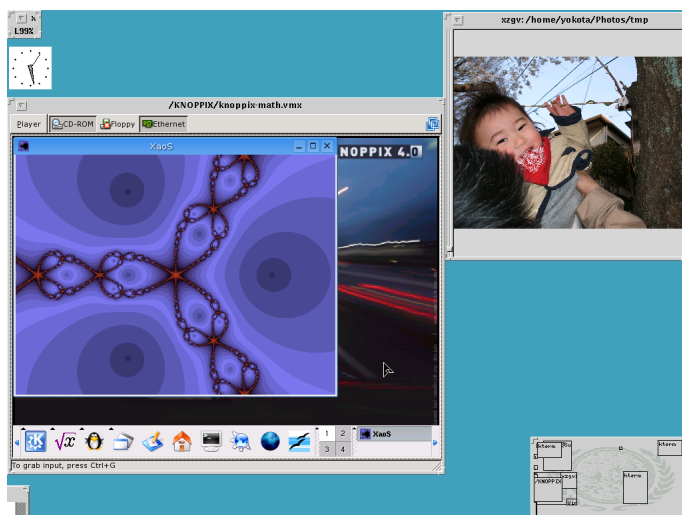


図 17.27: SuSE Linux と KNOPPIX/Math の共存

17.4 MS-Windows 環境への Maxima のインストール

17.4.1 インストール環境について

この章では MS-Windows 環境に Maxima をインストールする方法について述べます。猶, ここで解説するのは maxima-5.13.0 のインストールを MS-Windows2000 の環境で行った手順ですが, 他の MS-Windows XP や Vista でも恐らく大差無い処理手順になるかと思います。

猶, 「はじめての Maxima」に附属の CD-ROM には, MS-Windows 版の Maxima パッケージは収録されていないので, この作業には使えませんが, 様々な文書が収録されているので, 有用である事を強調しておきます。

17.4.2 Maxima の入手

Maxima を入手する為には、先ず、Maxima の開発拠点のサイトに行きます。

URI は <http://sourceforge.net/project/maxima/> です。図 17.28 は Firefox で上記のサイトに行った時の様子です。Internet Explorer を利用していても同じ様に表示されている筈です。

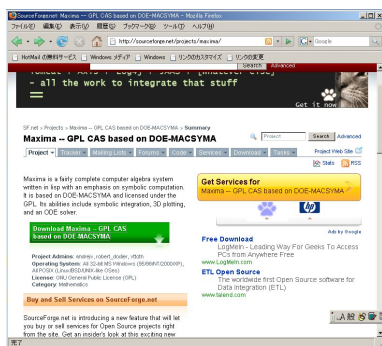


図 17.28: <http://sourceforge.net/project/maxima>

猶、皆さんもこのサイトにアクセスした時点で似た画面が表示されていると思いますが、PC の画面の大きさや、ブラウザのウィンドウの大きさの違いで、この表示が若干異なっているかと思います。今後、私の指示するボタン等が見えない場合は、マウスのホイールを使ったり、ウィンドウ右側のスライダーバーを使ってページを上下に動かしてみてください。

さて、話を元に戻します。この図 17.28 の左中程の大きな緑の箱に注目して下さい。この箱の中には以下の文言があります。Download Maxima ... GPL CAS based on ODE-MACSYMA 訳せば、「Maxima を入手する ... ODE-MACSYMA を基にした GPL の CAS」となります。GPL と言うのは、ライセンスの一形態です。このライセンスは、該当するソフトウェアの無料使用、自由な改変や自由な配布を保証するものです。詳しくは、<http://www.opensource.jp/gpl/gpl.ja.html> を参照して下さい。それから CAS ですが、これは Computer Algebra System、訳して計算機代数システムの事になります。さて、Maxima を入手する場合はこの緑の大きな箱にマウスのポインタを移動させ、それからマウスの左ボタンを押しましょう。すると図 17.29 に示すページに移ります。



図 17.29: 緑の箱を押した後

では、図の左下側の緑の大きな箱をマウスで先程と同様に押してみてください。すると、図 17.30 のページに移ります。

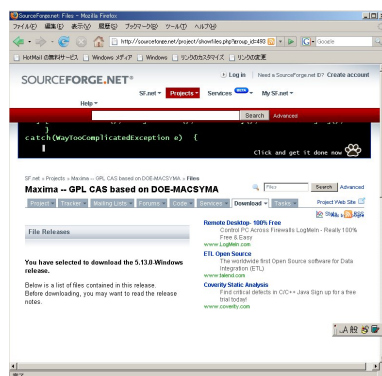


図 17.30: Download ボタンを押した後

図 17.30 に示すページには Maxima が入手出来そうな個所がありませんね。実は、Maxima を入手する為、このページの下側を見なければなりません。その為、ウィンドウ右側のスライダーバーを使うか、マウスのホイールを回す等して、このページの下側を見ましょう。肝心の下側を図 17.31 に示しておきます。

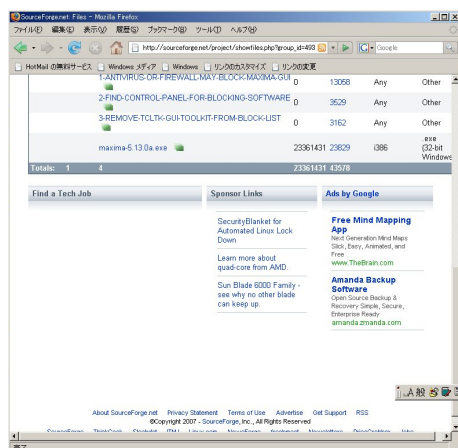


図 17.31: Download ボタンを押した後に移動したページの下側

さて、図 17.31 には maxima-5.13.0a.exe というファイルが右上にあります。ここで入手可能な Maxima は 5.13.0a という版になります。今後も、このページに maxima-x-xx-x.exe の様なファイルへのリンクがある筈です。

では、このファイル名にマウスのポインタを移動させて、マウスの左ボタンを押しましょう

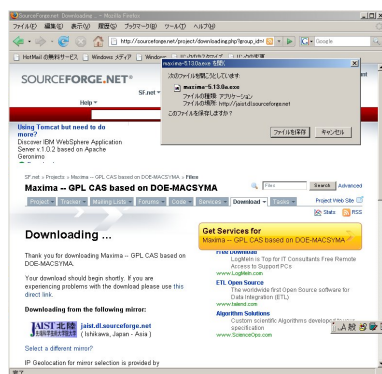


図 17.32: maxima-5.13.0a.exe を押して現われたページ

すると、図 17.32 に示すページに移動しますが、暫くすると、ダイアログが出て決めます。このダイアログ図 17.32 で右上に現われているもので、貴方がマウスのポインタで指定

したファイル入手するかどうかを尋ねるものです。ここでは勿論、ファイルを保存 ボタンを押しましょう。するとダウンロードを開始します。ここで、図 17.32 の左下を見て下さい。JAIST 北陸先端科学技術大学院大学とのアイコンが出ています。これは、Maxima のファイルは全世界の様々なサーバーに登録されており、今回の要求では JAIST にあるサーバーからファイル入手する事になったからです。ダウンロードが上手く出来ない場合、アイコン下側に表示されている [Select a different mirror](#) をマウスで押して、入手先を自分で選び直す事も出来ます。

ここでファイルを保存を選択すると、図 17.33 に示す様にダウンロードを開始します。ファイルは 22.3MB もあるので、多少時間がかかります。お茶でも飲んで待ちましょう。

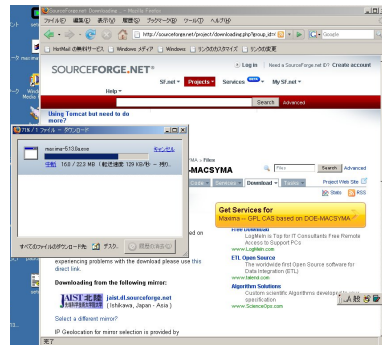


図 17.33: maxima-5.13.0a.exe を押して現われたページ

ダウンロードが無事に終了すると、デスクトップ上 (PC を立ち上げた時に、最初に出て来るアイコンが並んでいる所です) に Maxima-5.13.0a.exe というファイルがある筈です。これでインストーラが入手出来ました。次に、Maxima のインストールに移ります。ここで PC に詳しくない方の為に念を押しておく、先程入手したものは、Maxima のインストーラと呼ばれるソフトです。これは貴方の計算機に Maxima が使える様に環境を整えて呉れるソフトです。次のインストールと呼ばれる作業で貴方が Maxima を使える様に環境を整えるのです。

17.4.3 Maxima のインストール

Maxima のインストールは基本的にライセンスに同意するかどうかチェックを入れる箇所を除くと、後はそのまま **Next>** ボタンを押し続けるだけでインストールが行えます。但し、何の意味か判らずに Next を押し続けても面白くはないので、ここでは詳細を述べる事にします。

では、デスクトップ上の Maxima のインストーラをマウスでダブルクリックして立ち上げましょう。すると以下の図 17.34 に示すウィンドウが出て来る筈です。

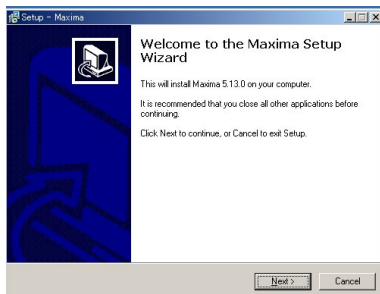


図 17.34: maxima-5.13.0a.exe を実行して出てきたウィンドウ

右下に二つボタンが並んでいますが、今回は Maxima のインストールを行うので、ここでは当然 **Next>** を押します。

すると、図 17.35 に示す表示に切替わります。これは Maxima が GPL に従うソフトウェアであり、そのライセンスに従うかどうかを尋ねるものです。

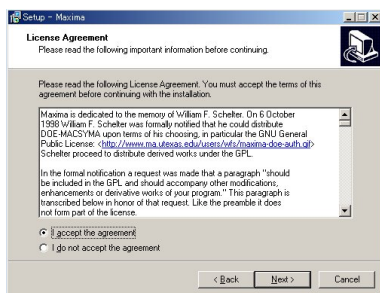


図 17.35: ライセンスに同意するかな？

先ず、このライセンス条項に違反する事は通常の利用では無いので、迷わずに

`I accept the agreement` のラジオボタンにチェックを入れて、図 17.35 の状態にします。猶、同意しない限り、先には進めません。

次に進むと、図 17.36 の表示に切替わります。ここでも迷わずに `Next>` を押します。

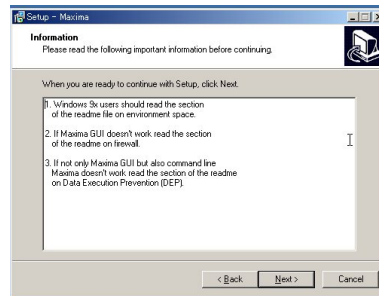


図 17.36: インストールに関する注意書き

因に、この図 17.36 で記述されている事柄は、第一に、MS-Windows 9x 利用者への注意事項、第二に Maxima が動作しない場合は readme の中の Firewall の項目を参照する事、最後に、GUI の Maxima だけではなく、コマンドラインの Maxima さえも動作しない場合、readme の DEP の箇所を参照する事といった注意書です。

最初の MS-Windows 9x の注意事項は、Maxima を MS-Windows 9x で動作させた時に、Out of environment space というエラーメッセージが出た場合の対処方法の話です。この対処方法は附属の readme ファイルにあります。

第二の Maxima が動作しないという問題は、MS-Windows に入れている firewall ソフト (anti-virus ソフトも含まれます) によって、インストールが出来ない事やインストールが出来ていても、Maxima 本体とフロントエンド間の通信が阻害される為に、結果が返って来ない現象の事です。ここで少しこの事を解説しておきましょう。先ず、Maxima の GUI 環境は Maxima そのものではなく、別個の独立したアプリケーションだという事です。MS-Windows 版には現在、wxMaxima と XMaxima の二つのフロントエンドがあります。これらのフロントエンドは、利用者が Maxima の処理文を書込むと、これを Maxima に送り込み、その結果を Maxima から送り返してもらって、フロントエンドが結果の表示を行っています。この Maxima への通信が Anti-virus アプリケーションや Firewall アプリケーションで阻害されてしまうと、何時まで経っても結果が表示されなかったり、フロントエンド側に、Timeout したとか言ったエラーが出る事になります。この対処方法は貴方の anti-virus アプリケーションや firewall アプリケーションの設定に依存するので、具体的な設定方法を述べる事が出来ませんが、Maxima の GUI

アプリケーション (wxMaxima や XMaxima) の通信を阻害しない様に設定すれば良いのです

そして, 最後の注意書きは MS-Windows の DEP(Data Exception Prevention) の対策です. 多分, この問題は Maxima の下層にある LISP で生じる事で, バイナリ配布の Maxima では生じないのではないかと思います.

では, **Next>** ボタンを押して, 次に移りましょう. すると図 17.37 に示すインストール先の設定に移動します.

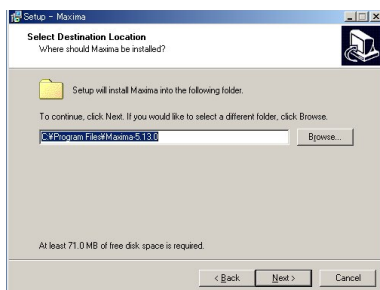


図 17.37: インストール先の指定

デフォルトでは `C:\Program File\Maxima-x-xx.x` のになっています (x-xx-x は Maxima の版, 例えば, 5-13.0). 通常はそのままが良いでしょう. 問題があれば変更しても構いません. 猶, 初期化ファイル `maxima-init.mac` を利用したい方は, この場所の事を良く憶えておきましょう. では, **Next>** を押しましょう. 今度は図 17.38 に示す様にパッケージの指定が行えます.

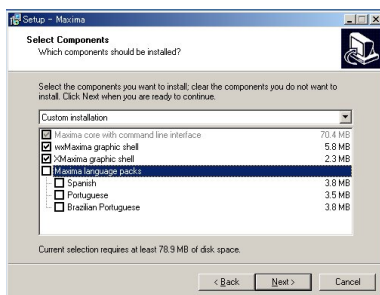


図 17.38: パッケージの指定

猶, Maxima の Language Pack には日本語は残念ながらありません. スペイン語やポ

ルトガル語で使う必要が無い限り, このパッケージを入れておく必要は無いでしょう. 勿論, デフォルトのままでも構いません. 不要ならチェックの入った箱を押せば, チェックが外れます.

では, **Next>** を押しましょう.

今度は図 17.39 に示す様にスタートメニューに登録するフォルダ名の指定が行えます.

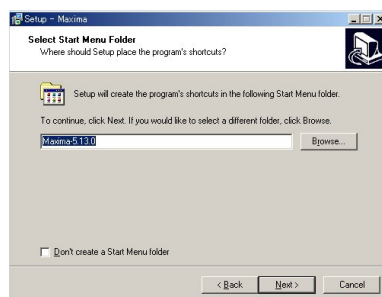


図 17.39: スタートメニューフォルダ名の指定

これは, MS-Windows 2000, 或いは MS-Windows XP で Classic な表示を選択した際に, スタートメニューを押して出て来るアプリケーションに含まれているフォルダ名を指定するものです. 変更しても構いませんし, そのままでも問題はありません.

色々出てきて疲れませんか? もう少しです. **Next>** を押しましょう.

今度は図 17.40 に示す表示になります. これはデスクトップに表示する Maxima のフロントエンドのアイコンを選択するものです.

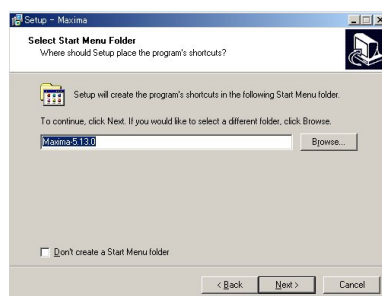


図 17.40: スタートメニューフォルダ名の指定

ここで MS-Windows 版の Maxima には wxMaxima と XMaxima の二つのフロント

エンドが付属しています. 初心者の方には wxMaxima の方が使い易いと思います. デフォルトでは wxMaxima になっているので, そのままで十分でしょう. GUI フロントエンドを選択したら **Next>** を押しましょう. すると図 17.41 に示す表示になります.

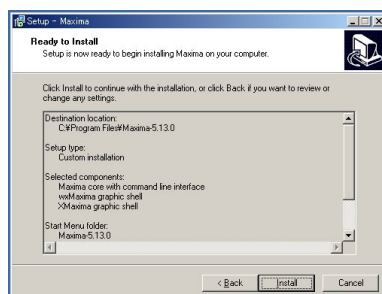


図 17.41: インストール内容の確認

これは今迄の設定を纏めたものです. 問題があれば, **<Back** を押して後に戻って設定し直しても構いません.

さてお疲れ様でした! ここで **Install** を押すといよいよ図 17.42 に示す様にインストールを開始します.

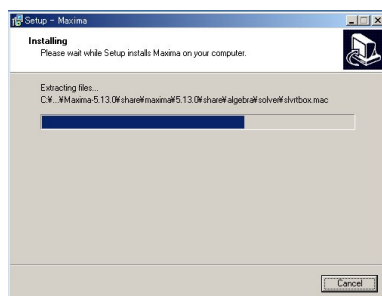


図 17.42: 只今インストール中

この作業も暫く時間がかかるので, ここで一寸一服しましょう.

さて、インストールが終了すると図 17.43 に示す様に,readme の内容が表示されます。

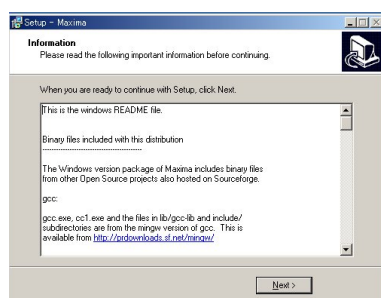


図 17.43: readme を表示中

ここで表示されているファイルは readme_en ファイルで、フォルダの指定がデフォルトのままであれば、C:\Program Files\Maxima-x-xx-x の直下にある筈です。今度も **Next>** を押しましょう。

すると、図 17.44 の画面が出ます。ここで **Finish** を押すとインストール作業は完了です。

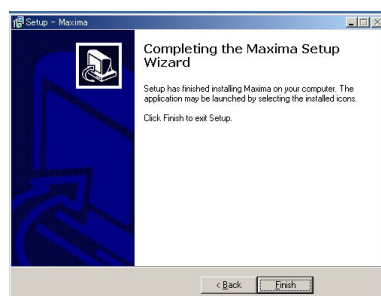


図 17.44: インストール終了のお知らせ

これでインストール作業は終了です。お疲れさま！

17.4.4 初期化ファイル

インストールも終わったので、ここでこの冊子を終えても良いのですが、Maxima で遊ぶ為に必要な事柄について、もう少しお話しておきます。

先ず、Maxima に初期化ファイル `maxima-init.mac` が利用可能です。この初期化ファイルには、Maxima の通常の文を書き込む事が可能で、Maxima が立ち上がる時にこのファイルの内容を実行します。

その為、日常的に用いるパッケージや関数の読込、定数の設定等に使えます。ここで MS-Windows 環境の場合はマウス操作による立ち上げもある為、この初期化ファイルの置場所に注意する必要があります。ここでは以下の各種の場合に分けて解説しておきましょう。

wxMaxima を利用する場合

wxMaxima をスタートメニューやデスクトップから立ち上げる場合には、初期化ファイルの置場を wxMaxima 本体と同じ階層のフォルダ、即ち、wxMaxima フォルダにします。デフォルトで Maxima は `C:\Program Files` に置かれますが、この Maxima フォルダの中に、wxMaxima のフォルダがあります。因に、その中に `wxMaxima.exe` があります。

具体的には、Maxima-5.13.0 の wxMaxima のフォルダはデフォルトの場合は、`C:\Program Files\Maxima-5.13.0\wxMaxima` になります。猶、wxMaxima を使っていて、グラフ表示のグラフ出力を画像ファイルに変更した場合、このフォルダに画像ファイルが生成される筈です。

猶、特定のフォルダに移動して DOS 窓から立ち上げる場合は事情が異なります。基本的には立ち上げたフォルダの中に初期化ファイルが必要になります。

スタートメニューから XMaxima や Maxima を選ぶ場合

先ず、XMaxima や Maxima をスタートメニューから立ち上げる場合は初期化ファイルを

`C:\Document and Settings` \以下にある利用者個別のフォルダに置きます。例えば、PC のログイン名が `ponpoko` であれば、ディスク C の `Document and Settings` フォルダの中にある `ponpoko` フォルダの中に置きます。

DOS 窓から立ち上げる場合

この場合は Maxima を何処で立ち上げるかに依存します。逆に言えば、Maxima を立ち上げる直前に移動したフォルダに初期化ファイルを置いておけば良いのです。例え

ば,DOS 窓で D:\Mike\Neko に移動して,そこで Maxima や wxMaxima を立ち上げるのであれば,D:\Mike\Neko に maxima-init.mac を置けば良いのです.

17.4.5 環境変数 Path の設定

ここで述べる事は,デスクトップにある Maxima のアイコンをダブルクリックして遊んでいる方には不要な事なので,読まなくても構いません.一寸したシステムの改善が関係します.

コマンドプロンプトから Maxima を立ち上げる場合,環境変数 Path の設定を行っていないと上手く出来ない事があります.その為に,何処でも Maxima を呼出せる様に MS-Windows のシステムの環境変数である Path の設定を行う必要が出ます.

ここで環境変数 Path を簡単に説明しておきましょう.MS-Windows ではアイコンをクリックしてアプリケーションを立ち上げたりしますね.これはリンクと呼ばれる手法で,要するに,飼犬を名札付きの紐で繋いだ様なものです.アイコンをクリックする操作は,名札付きの紐を引く事で例えられるでしょう.ところで,コマンドプロンプトから操作する場合,アイコンをクリックする手段はありませんね.そこで,計算機にアプリケーションを探させて立ち上げさせるのです.やり方は,環境変数 Path に予めアプリケーションの在処を記録しておけば,アプリケーションの呼出を受けた計算機が Path に登録されたフォルダを探して,該当するアプリケーションを立ち上げるという方法です.逆に言えば,この環境変数 Path に含まれていなければ,計算機はアプリケーションを見付けられないので,立ち上げる事が出来ない事になります.

さて,このシステム環境変数 Path の設定は,コントロールパネルのシステムから設定が出来ます.この時,注意する事は,既に値が設定されているので,迂闊に消さない様に,Maxima の所在を追加する事です.

さて,Maxima をここでは C:\Program Files\Maxima-5.13.0 にインストールしていたとしましょう.この時,Maxima の実行ファイルは Maxima のフォルダの下の bin フォルダに置かれています.その為,環境変数に追加する値は,C:\Program Files\Maxima-5.13.0\bin となります.ここで,環境変数 Path はこの様な表記をセミコロン (;) で繋いだ文字列です.その為,上記の文字列の前にセミコロン (;) を置いて,先程の文字列を追加すれば良いのです.

これで何が良いかというと,バッチ処理と呼ばれる自動処理が容易になるからです.バッチ処理と呼ばれる処理は,処理手順を予めファイルに書き下しておき,それを計算機に処理させる方法です.こうしておくと,定期的に作業を計算機に自動実行させる事や,手間を掛けずに定型的な処理を実行させる事が容易になるのです.実際,定期的な自動実行をさせる場合,計算機にアイコンをマウスでダブルクリックさせるのは現時

点では現実的ではないので、別のアプリケーションソフトから起動させる手順になるでしょう。この時、Path が明確でなければ、Maxima を立ち上げる事が出来ないのです。

17.5 我慢強い方向け

ここで述べるのは、訳があって KNOPPIX が使えない/使いたくない場合、Maxima を愛するがあまりに改造に燃える方向けです。

ここでは rpm や deb を用いてインストールする話は除外します。あくまでも古来からの *configure* ⇒ *make* という手順で説明します。

17.5.1 Common Lisp の選択

Maxima をソースからコンパイルする為には Common Lisp が絶対が必要です。無課金で利用可能な Common Lisp で代表的なものとして GCL, CLISP, CMUCL があります。他には, SBCL, ACL, OpenMCL があります。猶, 利用可能な Common Lisp を調べたければ, Maxima のソースファイルに含まれる *configure* を *--help* オプションで起動させると, *--enable* オプションの説明の個所で色々と出ています。

代表的な Common Lisp で, GCL, CMUCL や CLISP が動作する環境であれば, 選択の基準として処理速度を重視するのであれば CMUCL, share ライブラリ等の互換性を重視するのであれば GCL, 幅広い環境で利用したければ CLISP となるでしょう。

因に, これらの LISP を処理速度順で並べると, CMUCL > GCL > CLISP となる様です。CLISP の処理速度は今一つですが, CLISP は様々な環境に移植され, 更に, 安定している事もあって, 複数の環境で利用する場合, CLISP で統一しておく, 後の処理が楽な事もあります。

MacOSX の場合, CLISP に加え, SBCL や OpenMCL が移植されています。但し, 現時点で安定性を重視するのであれば, CLISP が最も無難と考えられます。

猶, 計算機の CPU が Core2Duo や Opteron で OS が 64bit OS の場合, SBCL が 64bit に対応しているので, こちらを使う事も良いでしょう。

ここで, Maxima と Common Lisp のバージョンには注意して下さい。例えば, 古い CLISP では新しい Maxima が上手く動作しない事がよく発生しています。この場合, 問題の少ない他の LISP に変更するか, 問題の無い LISP のバージョンに変更します。猛者は勿論, Maxima を作り替えても構いません。その場合, Maxima のソースファイルは *src* ディレクトリに含まれているので, 問題の個所を一つ一つ潰して行く事になります。

17.5.2 コンパイルの手順

現在の Maxima のコンパイルは非常に簡単になっています. 基本的には, 書庫ファイルを展開し, 附属の `configure` で利用する LISP 等を指定し, `make` でコンパイルを行い, 最後に `make install` でインストールを行うという手順になります.

少し前迄は, `gcl` でしかまともに動作せず, 色々とソースファイルから書換えを行っていましたが, その必要性は現在の様に環境が整えられている事もあって, その様な事をしなくてはならなくなる事はまずありません.

最初に適当なディレクトリ上で Maxima のソースファイルの書庫ファイルを展開します. 書庫ファイルには `gzip` や `bzip2` で圧縮したものがありますが, 新しい `tar` 命令を使えば, `tar -xvf 書庫ファイル` で展開出来ます.

展開して出来た Maxima のディレクトリに移動し, `./configure` を実行します. 通常はこれだけで済みますが, 環境によっては細かい設定を行いたい事もあるでしょう. 例えば, `clisp` や `gcl` 等の Common Lisp が複数存在する場合に, コンパイルに用いる Common Lisp を設定する必要もあるでしょう. この時, 設定可能なオプションは `./configure --help` を実行すると表示されます. 但し, `configure` が出力する情報は多いので, `./configure --help|less` の様に `less` を併用すると良いでしょう.

`configure` による設定で個人の環境に合せる為, インストール先のディレクトリを指定したい方が多いのではないかと思います. この場合は, `--prefix` オプションを使います. 具体的には, `--prefix="/usr/share"` の様にディレクトリを直接指定します. 何も指定しなかった場合, `/usr/local/share/maxima` にバージョンに対応するディレクトリが生成されます. 例えば, 2007 年 7 月の最新版は 5.12.0 ですが, デフォルトのインストール先は `/usr/local/share/maxima/5.12.0/` になります.

`configure` の実行後, 早速, `make` を実行します. 無事, `make` が完了すると, `make test` を行い, 出来上がった Maxima に問題がない事を確認しておきます. それから, `make install` で Maxima のインストールを行います. Maxima のインストールでは `configure` の `--prefix` オプションで指定したディレクトリの下に `maxima` ディレクトリが生成され, 更に, このディレクトリの下にバージョンに対応したディレクトリが生成されます. このディレクトリには以下のディレクトリが生成され, ファイルが分類されてます.

- `demo` ディレクトリ: デモファイルを格納
- `doc` ディレクトリ: マニュアル等の文書を格納
- `msgs` ディレクトリ: `ru.msg` を格納
- `share` ディレクトリ: Maxima のライブラリを含む

- src ディレクトリ:ソースファイルを格納
- tests ディレクトリ:テスト用ファイルを格納
- emacs ディレクトリ:emacs をフロントエンドとして利用する為のファイルを格納
- xmaxima ディレクトリ:xmaxima 関連のファイルを格納

このディレクトリ構成は MS-Windows 版でもほぼ同様です。

Maxima を改造したければ,Maxima のソースを変更し,Maxima のコンパイルとインストールを行う本格的な方法もありますが,手軽に修正する場合は, src ディレクトリに含まれるソースファイルを,適当なディレクトリに複製を取り,そのファイルを修正して Maxima から load 関数で読んでも構いません. 基本的にコンパイルを行った方が,処理は速いのですが,一寸した修正を行う程度であれば,手軽に行えるこの方法の方が安全です.

因に,src ディレクトリに含まれるファイルは全て LISP のプログラムです.従って,プログラムの修正は LISP のプログラムの修正となります.但し,適当に LISP のプログラムに print 関数を埋め込んで,S 式がどの様に書換えられるかを観察し,それから自分の望む処理を行う様に修正して行けば,結構,気楽に修正が行えます.

17.5.3 Maxima の初期設定

Maxima の初期設定はファイル名を maxima-init.mac とする Maxima の batch ファイルを Maxima を起動するディレクトリ上に置く事で行います.

基本的に,Maxima の通常の命令文と同じもので構いません. この初期設定で,自分専用のプログラムを読込ます様に出来ます. ですから,Maxima を一度コンパイルしておけば,プログラムの虫を見つけた場合や,改良は src ファイルから取出したソースファイルを適当なディレクトリに置いて,そのファイルを修正し, maxima-init.mac で読込む様にしておけば,古い函数は新しい函数で置換えられます.

第18章 最後に

Faust

Was ich verbaute,richte du grade,
was ich versäumte, schöpfe du nach.
So stell' ich mich über die Regel,
umfaß in einem die Epochen

und vermenge mich den letzten
Geschlechtern:

Ich,Faust,ein ewiger Wille!

ファウスト

俺が歪に築上げた事を, お前は正せ,
俺が怠けて損ねた物を, お前は掴み取るのだ.
そうすれば, 宿命を越えて俺は生き続け,
様々な時代を一つに包含し,

そして, 俺は遂には ^{はらから} 同胞 に混ざり合う:

我はファウスト, 永遠の意思!

Buzoni,Doktor Faust[58] より

この本は Maxima と銘打っておきながら, 実体は KNOPPIX/Math に収録されたアプリケーションの紹介も目的の一つでした. 皆さんは楽しんで頂けたでしょうか?

Maxima には, ここで解説したものの他に, 様々なライブラリも付属しています. 勿論,LISP の環境によっては十分に使えないものや, メンテナンスが十分に行なわれていないものもありますが, これらに関しては,Maxima に付属の文書を参照して下さい. この本で Maxima の計算原理に関連する事に重点を置いた訳はソフトウェアを使う際に, 使いこなせない問題の多くが, ソフトウェアの仕様や原理に理解不足である事, それに加えて, 対処しようとする問題への理解不足, 即ち, ソフトウェアと問題の双方の理解不足に大きな問題があると感じる事が多いからです. そこで, この本では Maxima の函数の一覧や, その例題集を作るよりは, 敢て入力した式がどの様に内部で処理されてゆくかに焦点を当てています. 結局, 内部で式がどの様に表現され, どの様に処理されているかを理解さえしていれば, 自分が必要とする函数も臆げながらも見当が付く, 後は Maxima に付属の文書で調べれば良いからです.

この Maxima が Maple や *Mathematica* と比べて最も優れている点は, 誰もが疑問を持ったら自由に中身を調べる事が出来る点でしょう. 勿論,Maple ではライブラリを自ら調べる事が可能だったり,Reduce の様にソースコード込で購入する事が可能な数式処理システムもあります. しかし, 無条件で誰でもシステム全体を調べる事が可能なのは Maxima の長所です. ここでシステム全体の動作を完全に理解する必要はありません. 自分が必要とされる函数にどの様な函数や大域変数が用いられているか調べる事,

より進んで、ソースファイルの関数内部に print 関数を挿入して、S 式がどの様に変化するかを眺めるだけでも十分なのです。それだけでも、何も知らずに使うのとは全く違います。積極的に Maxima を弄る事で、Maxima の限界や問題点だけではなく、それらを越える新たな可能性にも気付かされる筈です。

数学の色々な事には、Maxima で遊ぶ為に最低限必要と私か感じた事項をとりとめもなく書いています。冒頭のロゴス賛歌の様に、脈絡も何もなく混沌としたアトムの世界を論理 (logic) で束ね、そこから色々なものが派生して行く様子を表現したかったのですが、私の力不足で十分とは思えません。

更に、Frege の解説が必要以上に多いと思われるかもしれませんが、Frege の著作にはそれだけの魅力 (魔力?) があると理解して頂ければ幸いです。兎に角、一つの世界を創造してゆく過程を目の当たりにする事が出来るのです。又、個人的な思い出になりますが、Frege の「算術の基礎」を読んでいて、成績不良の中学生の頃に、「何故、 $1+1=2$ なのか」と考えていた事を懐しく思い出した事も付け加えておきます。優秀な同級生は私の質問に真面に答える事は無く、寧ろ、「勉強の出来ない真面目な子」の方が説明しようとしたものです。中には鉛筆と鉛筆キャップを使って説明してくれた子も居ましたが、意地悪にも、鉛筆キャップを鉛筆に付けて、一つになったから $1+1=1$ じゃないかと言えば、その子は流石に私を憐む様な顔をして答に窮していましたが…。

結び目の章は、規則とその評価に重点を置いています。基本的に、計算機に入力される式は、ASCII データの羅列でしかありません。その中の演算子で式を区切り、演算子の属性、更に関数の持つ性質 (公式等) で式を簡易化して行く訳ですが、そこで重要なのは置換規則、変数並びの指定等で、如何に適切に規則を与えられた並びに対して適用するかが問題となります。この章は十分描き切れたとは思えませんが、空間図形から妙な文字の羅列が出来上り、それに色々と細工をする内に意外な意味が生じる所を楽しんで頂ければと思っています。

surf でお絵描きを行う章は、代数学の様々な概念を気楽に楽しんでもらおうと思って記述しました。SINGULAR も引っ張ってきましたが、見掛けは堅苦しい SINGULAR も surf と併用すれば案外軟派な使い方が出来るので、Maxima で不十分と感じれば、色々と試してみて頂ければと思っています。

Octave の話は、Maxima に限らず数式処理で強引に非常に大きな数値行列の処理を実行したものの、処理や速度に難があり、それが数式処理一般の評価を下げる面がある様に感じた点もあります。数値行列は MATLAB や Octave の様なシステムで処理すべきであり、数式処理は MATLAB や Octave が苦手とする記号処理で活用すべきだと私は思います。猶、その割には貧弱なインターフェイスプログラムしか書いていませんが、これを参考にして良いものを作ってみて下さい。

Maxima の簡単な改良の話は、目標とするものと、その結果の落差が大きく、腰砕け気

味ですが,色々と改造して行く事が可能なのは Open Source ならではの事です.
インストールの話では自力でコンパイルする話の分量が少なくなっています. これは,
少し前と比べて格段に Maxima を使う環境が整っている事の現れと理解して下さい.
ここでは寧ろ, 仮想計算機環境の事を記述していますが, この仮想計算機は少し前の
OS 論争を完全に過去のものとするだけの威力が十分にあります.
最後に,Maxima を自分流に使って楽しんで下さい.

関連図書

- [1] アリストテレス, 形而上学 上, 岩波文庫, 2007.
- [2] 飯田隆, 言語哲学大全 I 論理と言語, 勁草書房, 2003.
- [3] 井筒俊彦, イスラーム思想史, 中公文庫, 中央公論社, 1991.
- [4] 岩熊幸雄, ライプニッツの内包論理-Generales Inquisitiones の一解釈-, 哲学論叢 1-Aug-1978, 京都大学哲学論叢刊行会, <http://hdl.handle.net/2433/24432>
- [5] アンドレ・ヴェイユ, アンドレ・ヴェイユ自伝-ある数学者の修行時代, シュプリンガー・フェアラーク, 1994.
- [6] ヴォルテール, カンディード:他五篇, 岩波文庫, 岩波書店, 2005.
- [7] G.H. フォン・ウリグト, 論理分析哲学, 講談社学術文庫, 講談社, 2000.
- [8] 河上徹太郎, 近代の超克, 富山房百科文庫 23, 富山房, 1979.
- [9] 桂紹隆, インド人の論理学, 中公新書, 中央公論社, 1998.
- [10] ガリレオ, 新科学対話 (上), 岩波文庫, 岩波書店, 2007.
- [11] クセルゴン, 自由・平等・清潔-入浴の社会史, 河出書房新社, 1992.
- [12] ポール・グレアム, ANSI Common Lisp, ピアソン・エデュケーション, 2002.
- [13] 河内明夫編, 結び目理論, シュプリンガー・フェアラーク東京, 1990.
- [14] J. リヒター-ゲバート/U.H. コルテンキャンプ著, 阿原一志訳, シンデレラ 幾何学のためのグラフィックス, シュプリンガー・フェアラーク東京, 2001.
- [15] 小平邦彦, 幾何学の誘い, 岩波書店, 1991.
- [16] 下地貞夫, 数式処理, 基礎情報工学シリーズ, 森北出版, 1991.

- [17] ソーマディーヴァ(著), 上村勝彦(訳), 屍鬼二十五話-インド伝奇集, 東洋文庫 323, 平凡社, 1978.
- [18] ゲーデル, 林晋・八杉満利子訳・解説, 不完全性定理, 岩波書店, 2006.
- [19] 高木貞治, 復刻版 近世数学史 数学雑談, 共立出版, 1997.
- [20] 田畑博敏, フレーゲの論理哲学, 九州大学出版会, 2002.
- [21] 田中尚夫, 選択公理. 遊星社, 1987.
- [22] 寺坂英孝編, 現代数学小事典, ブルーバックス, 講談社, 2005.
- [23] 日本科学哲学学会 [編], 野本和幸 [責任編集], 科学哲学の展開 [1], 分析哲学の誕生, フレーゲ・ラッセル, 勁草書房, 2007.
- [24] 広瀬健 横田一正, ゲーデルの世界 -完全性定理と不完全性定理-, 海鳴社, 1985.
- [25] ヒルベルト, 幾何学基礎論, 筑摩書房, 2005.
- [26] ヒルベルト ベルナイス, 数学の基礎, シュプリンガー・フェアラーク, 1993.
- [27] R. フィンスター, G. ファン・デン・ホイフェル著, 沢田允茂監訳, 向井久他訳, ライプニッツ その思想と生涯, シュプリンガー・フェアラーク東京, 1996.
- [28] クロウエル, フォックス, 結び目理論入門, 現代数学全書, 岩波書店, 1989.
- [29] フレーゲ, フレーゲ著作集 1 概念記法, 勁草書房, 1999.
- [30] フレーゲ, フレーゲ著作集 2 算術の基礎, 勁草書房, 1999.
- [31] フレーゲ, フレーゲ著作集 3 算術の基本法則, 勁草書房, 2000.
- [32] 藪内清, 墨子: 東洋文庫, 平凡社, 1996.
- [33] 本間龍雄, 組合せ位相幾何学, 共立出版, 1980.
- [34] 前原昭二, 数学基礎論入門, 朝倉書店, 2007.
- [35] 丸山茂樹, クレブナー基底とその応用, 共立叢書 現代数学の潮流, 共立出版, 2002.
- [36] 村上順, 結び目と量子群, 数学の風景 3, 朝倉書店, 2000.
- [37] 日本数学会編, 数学辞典 第 3 版, 岩波書店, 1987.

- [38] 谷沢淳三, 論証の学としてのインド論理学:帰納法と演繹法, 人文科学論集. 人間情報学科編 信州大学 Vol41(20070315)pp.233-252, <http://ci.nii.ac.jp/naid/110006389066/>
- [39] 矢吹道郎, 大竹敢, 使いこなす GNUPLOT(改訂新版), テクノプレス,2001.
- [40] 山川偉也, 条件文についての古代の論争:メガラ・ストア論理学の理解のために (共同研究:「言語の本質」についての総合的研究), 総合研究報 St.Andrew's University,Bulletin of Research Institute, Vol.8,No.1(19820930)pp. 1-14, 桃山学院大学 ISSN:03850811
- [41] 湯浅太一, 萩谷昌己,Common Lisp 入門, 岩波書店,2007.
- [42] 吉田利信, 芹沢昭生, はじめての LISP, 技術評論社,1985.
- [43] B. ラッセル (著), 野田又夫 (訳, 私の哲学の発展, みすずライブラリー, みすず書店,1997.
- [44] ルキアノス, ルキアノス選集, 叢書アレキサンドリア図書館第 8 巻, 国文社,1999.
- [45] ルキアーノス, 神々の対話 他 6 篇, 岩波文庫, 岩波書店,1953.
- [46] G.Boole, The Calculus of Logic,The Cambridge and Dublin Mathematical Journal,vol. 3 (1848), <http://www.maths.tcd.ie/pub/HistMath/People/Boole/CalcLogic/>.
- [47] G.Boole, Investigatio of the Law of Thought,Dover,1973, [http://http://www.gutenberg.org/etext/15114](http://www.gutenberg.org/etext/15114).
- [48] H.Cohen, A Course in Computational Algebraic Number Theory,GTM 138, Springer-Verlag,New York-Berlin,2000.
- [49] D.Cox,J.Little and D. O'Shea,Ideals, Varieties, and Algorithms, UTM,Springer-Verlag,New York-Berlin,1992.
- [50] Gert-Martin Greuel, Gerhard Pfister, A Singular Introduction to Commutative Algebra, Springer-Verlag,New York-Heiderberg-Berlin,2000.
- [51] Heijenoort, From Frege to Gödel, A Source Book in Mathematical Logic ,1879 - 1931, HARVARD UNIVERSITY PRESS,1976.
- [52] D.Rolfsen, Knots and Links. Publish or Perish, Inc.,1975.

- [53] B.Russell, The Principles of Mathematics, W.W.Norton & Company, Inc., 1996.
- [54] Hal Schenck, Computational Algebraic Geometry London Mathematical Society student texts; 58, 2003.
- [55] Michael Schroeder, A BRIEF HISTORY OF THE NOTATION OF BOOLE'S ALGEBRA, Nordic Journal of Philosophical Logic, Vol.2, No.1, pp.41 -62, <http://www.hf.uio.no/ifikk/filosofi/njpl/vol2no1/history/history.pdf>.
- [56] Stanford Encyclopedia of Philosophy. <http://plato.stanford.edu/contents.html>
- [57] Frege's Logic, Theorem, and Foundations for Arithmetic, Stanford Encyclopedia of Philosophy. <http://plato.stanford.edu/entries/frege-logic>
- [58] Ferruccio Busoni, Doktor Faust. (ERATO)
- [59] Open AXIOM のサイト <http://wiki.axiom-developer.org/FrontPage>
- [60] DERIVE のサイト <http://www.derive.com>
- [61] GAP のサイト <http://www-gap.dcs.st-and.ac.uk/>
- [62] Macaulay2 のサイト <http://www.math.uiuc.edu/Macaulay2/>
- [63] Mathsoft Engineering & Education, Inc. <http://www.mathcad.com/>
- [64] Wolfram Research Inc. <http://www.wolfram.com/>
- [65] Waterloo Maple Inc. <http://www.maplesoft.com/>
- [66] Maxima の SOURCEFORGE のサイト <http://maxima.sourceforge.net/>
- [67] MuPAD のサイト <http://www.mupad.de/>
- [68] PARI/GP のサイト <http://pari.math.u-bordeaux.fr/>
- [69] REDUCE のサイト <http://www.zib.de/Symbolik/reduce/>
- [70] Risa/Asir 神戸版 <http://www.math.kobe-u.ac.jp/Asir/asir-ja.html>
- [71] OpenXM (Open message eXchange for Mathematics)
<http://www.math.sci.kobe-u.ac.jp/OpenXM/index-ja.html>
- [72] SINGULAR のサイト <http://www.singular.uni-kl.de/>

- [73] 株式会社シンプレックスのページ <http://www.simplex-soft.com/>
- [74] The MathWorks, Inc. <http://www.mathworks.com/>
- [75] Octave WebPage <http://bevo.che.wisc.edu/octave/>
- [76] Scilab のサイト <http://www.scilab.org/>
- [77] Yorick の公式サイト <ftp://ftp-icf.llnl.gov/pub/Yorick/>
Yorick の非公式サイト <http://www.maumae.net/yorick/doc/index.php>
- [78] R のサイト <http://www.r-project.org>
- [79] Insightful Corporation のページ <http://www.insightful.com/>
- [80] Cinderella のサイト
本家：<http://www.cinderella.de/>
日本語版ホームページ <http://cdyjapan.hp.infoseek.co.jp/>
- [81] dynagraph のサイト <http://www.math.umbc.edu/rouben/dynagraph>
- [82] KSEG のサイト <http://www.mit.edu/ibaran/kseg.html>
- [83] Geomview のサイト <http://http.geomview.org/>
- [84] surf の sourceforge のサイト <http://surf.sourceforge.net/>
- [85] XaoS のサイト <http://wmi.math.u-szeged.hu/kovzol/xaos>
- [86] Begriffsschrift in $\text{\LaTeX}$ <http://arche-wiki.st-and.ac.uk/ah-wiki/bin/view/Main/BegriffsschriftLaTeX>
- [87] fge パッケージのサイト <http://sview01.wiredworkplace.net/pub/jjg/en/code/fge.html>
- [88] Numbers, constants and computation, <http://numbers.computation.free.fr/Constants/constants.html>

索引

逆引

三角函数

三角函数を含む式の展開を行いたい, 554

三角函数を含む式を簡単にしたい, 557

式

Maxima に式の自動展開をさせたい, 386

式の因子分解をしたい, 339

式を展開したい, 390

常微分方程式

解の検証, 618

微分方程式の書き方, 615

常微分方程式の簡単な解き方, 616

数値

Maxima の数学定数, 304

精度を変更したい, 303

積分

数値積分を行いたい, 610

代数的数を使って積分したい, 600

変数変換をしたい, 598

ラプラス変換を使いたい, 605

留数の計算をしたい, 605

定積分を行いたい, 608

微分

式の微分をしたい, 586

評価

函数や演算子に影響を与える大域変数を知りたい, 295

式を手軽に評価したい, 291

方程式の解を手軽に他の式に入りたい, 293

式を手軽に展開したい, 290

gnuplot の使い方を知りたい, 723

plot2d 函数の使い方, 698

plot3d 函数の使い方, 701

方程式

決った範囲内で実数解だけを求めたい, 573

実数解だけを求めたい, 572

自動的に方程式の解を変数に入りたい, 569

連立方程式を解きたい, 574

演算子

～の型, 190

～の束縛力, 20, 188, 848

～の束縛力 (bp), 188

～の左束縛力 (lbp), 188

～の右束縛力 (rbp), 188

外挿表現の～, 186

可換積, 197

後置表現の～, 186

前置表現の～, 186

内挿表現の～, 186

非可換積, 197

無引数の~, 186

A

and, 200

D

do, 203

Dot 積 (=非可換積), 17

E

else, 203

elseif, 203

F

for, 203

from, 203

I

if, 203

if 文で利用可能な演算子, 482

in, 483

N

next, 203, 483

not, 200

O

or, 200

S

step, 203, 483

T

then, 203

thru, 203, 484

U

unless, 203, 484

W

while, 203, 484

記号

:lisp, 181, 298, 689

:=, 489

??, 665

?, 665

!!, 199, 526

! , 199, 526

’’, 294

’, 294, 493

**, 197

*, 197

+, 197

-, 197

., 197

/, 197

:=, 201, 229, 493

::, 201

:=, 14, 201, 229

::, 14, 173, 201, 907

;, 13

<=, 200

<, 200

=, 14, 200, 567, 615

>=, 200

>, 200

?, 16, 298

@, 487

#, 200

\$, 13

^^, 31, 197

^, 197

型

A

any, 190, 508, 510

any_check, 508

B

big, 508

bignum, 168, 508

boole, 508

- boolean, 508
- C
 - clause, 190
 - complex, 508
- E
 - expr, 190
- F
 - fixnum, 168, 508
 - fixp, 508
 - float, 508
 - floatnum, 508
 - flonum, 508
- I
 - integer, 508
- L
 - list, 508
 - listp, 508
- N
 - none, 508
 - number, 508
- R
 - rat, 508
 - rational, 508
 - real, 508
- た
 - 多倍長浮動小数点数, 169
- ふ
 - 浮動小数点数, 169
- 函数
 - ?
 - ?round, 310
 - ?truncate, 310
 - %
 - %, 14
 - %ith, 651
- %j, 677
- %th, 14
- A
 - absolute_real_time, 675
 - acos, 554
 - acosh, 554
 - acot, 554
 - acoth, 554
 - acsc, 554
 - acsch, 554
 - activate, 258
 - add_zeros, 807
 - addcol, 443, 859
 - addrow, 443
 - adjoin, 421
 - adjoint, 447, 859
 - airy_ai, 620
 - airy_bi, 620
 - airy_dai, 620
 - airy_dbi, 620
 - algsys, 27, 574
 - alias, 229, 681
 - allroots, 571
 - aload_mac, 517
 - alphanumericp, 464
 - antid, 614
 - antidiff, 614
 - append, 402
 - appendfile, 41, 519
 - apply, 496
 - apply1, 270, 272
 - apply2, 272
 - approps, 681
 - apropos, 16
 - args, 362

- array, 428
- arrayapply, 431
- arrayinfo, 427
- arraymake, 428
- ascii, 466
- asec, 554
- asech, 554
- asin, 554
- asinh, 554
- askinteger, 394
- asksign, 396
- assoc, 379
- assume, 23, 175, 209, 248, 249, 850
- at, 223
- atan, 554
- atan2, 554
- atanh, 554
- atom, 401
- atomp, 244
- atvalue, 221, 228, 615
- augcoefmatrix, 435
- auto_mexpr, 517
- B
 - bashindices, 375
 - batch, 516, 778
 - batchload, 516
 - bc2, 29, 619
 - belln, 528
 - bern, 528
 - bernpoly, 528
 - bessel, 621
 - bessel_i, 621
 - bessel_j, 621
 - bessel_k, 622
 - bessel_y, 622
 - beta, 532
 - bezout, 334
 - bflhzeta, 538
 - bfloat, 244, 306
 - bfloat_approx_equal, 305
 - bfloatp, 305
 - bfzeta, 538
 - bigfloat, 302
 - binomial, 532
 - block, 285, 480
 - bothcoef, 329
 - break, 485, 646
 - bug_report, 687
 - build_info, 687
 - buildq, 493
- C
 - cabs, 308
 - cardinality, 418
 - carg, 308
 - cartesian_product, 414
 - catch, 485
 - ceiling, 306
 - cequal, 465
 - cequalignore, 465
 - cf, 539
 - cfexpand, 539
 - cfdisrep, 539, 540
 - cgreaterp, 465
 - cgreaterpignore, 465
 - changevar, 598
 - chaosgame, 628
 - charat, 468
 - charfun, 243
 - charlist, 468

charp, 464
charpoly, 32, 450
cint, 466
clear_rules, 270
clearrule, 281
cless, 465
clessignore, 465
close, 459
closefile, 41, 519
closeps, 720
coeff, 329
coefmatrix, 435
col, 443
collapse, 645
columnvector, 453
combine, 332
compare, 245
compfile, 502
compile, 502
compile_file, 502
concat, 458
conj, 453
conjugate, 453
constantp, 244, 305
constituent, 464
content, 337, 537
copy, 378, 403, 444
copyp, 378, 402
copymatrix, 378, 443
cos, 553
cosh, 553
cot, 553
coth, 553
coutou_plot, 720
create_list, 398

csc, 553
csch, 553
cubrt, 677

D

dblnt, 612
deactivate, 258
declare, 167, 177, 179, 208, 228,
230, 250, 288
declare_translated, 512
define, 229, 489
define_variable, 214, 507
define_variable 函数の処理手順,
510
defint, 608
defmatch, 267
defrule, 230, 263, 269
defstruct, 487
deftaylor, 225, 230
del, 586
delete, 402
delta, 605
demo, 663
demoivre, 389
denom, 332
depends, 223, 230, 597
derivdegree, 588
derivlist(ev 函数の引数), 287, 291
describe, 15, 663, 824
desolve, 617
determinant, 450, 859
detout(ev 函数の引数), 287
diagmatrix, 435
diagmatrixp, 244
diff, 25, 586
diff(ev 函数の引数), 287

- digitcharp, 244, 464
- dipslay, 656
- disjoin, 421
- disjointp, 425
- disp, 656
- dispform, 197, 358
- dispfun, 195, 492
- displate, 363
- disprule, 269, 270
- dispterm, 657
- distrib, 391
- divide, 343
- divisors, 414
- do, 482
- dpart, 659
- draw, 788
- draw2d, 788
- ～と draw2d, draw3d との関係,
788
- draw3d, 788
- dvsum, 549
- ldisplay, 656
- E
 - derivlist, 291
 - echelon, 435
 - econs, 402
 - ed, 523
 - eigenvalues, 32, 453
 - eigenvectors, 32, 453
 - eivals, 453
 - eivects, 453
 - elapsed_real_time, 675
 - elapsed_run_time, 675
 - elementp, 425
 - eliminate, 334
 - elliptic_e, 625
 - elliptic_ec, 625
 - elliptic_eu, 625
 - elliptic_f, 625
 - elliptic_kc, 625
 - elliptic_pi, 625
 - ematrix, 435
 - empty, 425
 - endcons, 402
 - entmatrix, 435
 - entier, 306
 - equal, 175, 232
 - equiv_classes, 422
 - erf, 598
 - errcatch, 486
 - error, 486, 661
 - errmsg, 486, 661
 - euler, 533
 - ev, 24, 238, 284, 618
 - eval, 294
 - eval(ev 関数の引数), 287
 - evenp, 244, 305
 - every, 241, 243
 - evolution, 628
 - evolution2, 628
 - ev 関数で利用可能な引数, 287
 - example, 15, 663
 - exp, 559
 - expand, 22, 290, 390, 875
 - expand(ev 関数の引数), 287, 290
 - exponentialize, 389
 - express, 593
 - external_subset, 422
 - ezgcd, 337
- F

facout, 397
factcomb, 332
factor, 22, 340
factorout, 340
factorsum, 340
facts, 209, 248
facttimes, 332
feature, 210
featurep, 210, 250, 850
features, 208
Ffortmx, 679
Ffortran, 679
fib, 534
fibtophi, 534
file_search, 515
file_type, 515
filename, 515
fillarray, 431
first, 404
fix, 306
flatten, 416
flength, 460
float, 244
float_approx_equal, 305
float_approx_equal_tolerance, 305
floatnump, 305
floor, 306
forget, 248
fortran, 32
fposition, 460
freeof, 239
freshline, 461
full_listify, 416
fullmap, 309, 411
fullmapl, 411

fullratsimp, 344
fullratsubst, 382
fullsetify, 415
funcsolve, 581
fundef, 492
funmake, 497

G

Γ 函数, 535
gamma, 535
gcd, 337
gcde, 337
gcdex, 337
gcfactor, 337, 549
genfact, 526
genmatrix, 435
get, 207, 874
gfactor, 337
gfactorsum, 337
gnuplot_close, 716
gnuplot_pipes, 716
gnuplot_replot, 716
gnuplot_reset, 716
gnuplot_restart, 716
gnuplot_start, 716
go, 485
gradef, 223, 230
gramschmidt, 454
grind, 42, 656
gschmidt, 454

H

help, 665
hipow, 330, 588
horner, 33, 328

I

ic1, 619

ic2, 29, 619
ident, 435
identity, 243
if, 481
ifactor, 542
ifs, 628
ilt, 605
imagpart, 302, 308
infix, 192
inflag の影響を受ける関数, 401
innerproduct, 452
inpart, 367, 380, 383, 847
inprod, 452
inrt, 306
integer_partitions, 424
integerp, 244, 305
integrate, 26, 218, 595
interpolate, 580
intersect, 417
intersection, 417
intosum, 393
inv_mod, 542
inverse_jacobi_cd, 624
inverse_jacobi_cn, 623
inverse_jacobi_cs, 624
inverse_jacobi_dc, 624
inverse_jacobi_dn, 623
inverse_jacobi_ds, 624
inverse_jacobi_nc, 624
inverse_jacobi_nd, 624
inverse_jacobi_ns, 624
inverse_jacobi_sc, 624
inverse_jacobi_sd, 624
inverse_jacobi_sn, 623
invert, 447

is, 238
isolate, 363
isqrt, 306, 541
ivert, 31

J

jacobi, 549
jacobi_am, 623
jacobi_cd, 624
jacobi_cn, 623
jacobi_cs, 624
jacobi_dc, 624
jacobi_dn, 623
jacobi_ds, 624
jacobi_nc, 624
jacobi_nd, 624
jacobi_ns, 624
jacobi_sc, 624
jacobi_sd, 624
jacobi_sn, 623
jullia, 629

K

kill, 15, 644, 647
killcontext, 258, 259
kron_delta, 551
kron_delta, 551

L

labels, 651
lambda, 285, 491, 858
laplace, 605
last, 404
lcharp, 464
ldefint, 608
ldisp, 656
length, 402
let, 263, 276

- letrules, 278
- letsimp, 278
- lfreeof, 239
- lhs, 204, 567
- li, 536
- limit, 218, 584, 597
- linsolve, 27, 573
- list_matrix_entries, 451
- listarray, 427
- listify, 416
- listofvars, 362, 846
- listp, 401
- lmax, 305, 419
- lmin, 305
- lmix, 419
- load, 32, 42, 517, 853, 884
- loadfile, 42, 517
- local, 498
- local(ev 関数の引数), 287, 292
- log, 559
- logarc, 561
- logconrctat, 562
- logout, 682
- lopow, 330
- lowercasep, 244, 464
- lpart, 659
- lratsubst, 382
- lreduce, 419
- lstring, 827, 832
- lstringp, 464
- lsum, 370, 374
- stringp, 244
- lstring, 466
- M
- kill, 196
- macroexpand, 495
- macroexpand1, 495
- mainvar, 327
- make_array, 428
- make_art_q, 428
- make_elliptic_e, 626
- make_elliptic_f, 626
- make_fransform, 708
- make_random_state, 309
- make_string_input_stream, 460
- make_string_output_stream, 460
- makegamma, 527, 536
- makelist, 398, 529
- makeset, 414
- mandelbrot, 629
- map, 409, 858
- mapatom, 409
- matchdeclare, 228, 264
- matchfix, 194
- matrix, 29, 433
- matrixmap, 442
- matrixp, 244
- mattrace, 446
- max, 305
- maybe, 238
- member, 401, 847
- mfuncall, 299
- min, 305
- minfactorial, 332, 527
- minor, 445, 859
- mod, 528
- mode_declare, 229, 501, 507
- mode_identity, 507
- modeddeclare, 229
- moebius, 537

- multinomial_coeff, 527
- multthru, 391
- makefact, 527
- N
 - nary, 192
 - ncharpoly, 450
 - new, 487
 - newcontext, 258
 - newdet, 450
 - newline, 461
 - next_prime, 542
 - niceindices, 375
 - noeval(ev 関数の引数), 287
 - nofix, 193
 - nonscalarp, 244
 - nonzeroandfreeof, 614
 - notequal, 175, 232
 - nounify, 214, 294, 355
 - nouns(ev 関数の引数), 287
 - nroots, 573
 - nterms, 330
 - nthroot, 340
 - num, 332
 - num_distinct_partitions, 424
 - num_partitions, 424
 - numberp, 244, 305
 - numer(ev 関数の引数), 287
 - numerval, 396
 - numfactor, 537
- O
 - oddp, 244, 305
 - ode2, 28, 617
 - op, 205, 367, 369
 - opena, 459
 - openplot_curves, 717
 - openr, 459
 - openw, 459
 - optimize, 499
 - options, 284, 295
 - orbits, 629
 - ordergear, 182
 - ordergreatp, 183
 - orderless, 182
 - orderlessp, 183, 413
 - outermap, 412
 - print, 656
- P
 - iprimep, 542
 - pade, 351
 - parmanent, 450
 - parsetoken, 468
 - part, 367, 380, 383
 - partfrac, 344
 - partition, 366
 - partition_set, 422
 - permutation, 421
 - pfet, 390
 - pickapart, 367
 - playback, 658
 - ploarform, 564
 - plog, 559
 - plot_format, 11
 - plot2d, 35, 698
 - plot2d_ps, 720
 - plot3d, 36, 702
 - plotdf, 630
 - polar_to_xy, 708
 - poly_discriminant, 334
 - polydecomp, 340
 - polymod, 343, 528

- ponpoko, 829
- postfix, 194
- power_mod, 542
- powers, 330
- powerseries, 348
- powerset, 421
- prefix, 194, 848
- prev_prime, 542
- primep, 244, 305
- printf, 461
- printf, 523
- printlistvar, 331
- printprops, 227
- product, 217, 370
- properties, 206, 226, 288, 874
- propvars, 226
- pscom, 720
- psdraw_curve, 720
- psdraw_points, 720
- put, 207, 874
- psi, 538
- Q
 - qput, 207, 510
 - quanc8, 597, 610
 - quit, 15, 47, 682
 - qunit, 541
 - quotient, 343
- R
 - radcan, 394
 - random, 309
 - random_permutation, 421
 - rank, 446
 - rat, 315
 - ratcoef, 329
 - ratdenom, 332
 - ratdiff, 332
 - ratdisrep, 315
 - ratexpand, 344
 - rationalize, 308
 - ratnumer, 332
 - ratnump, 244, 328
 - ratp, 244, 328
 - ratsimp, 22, 344, 875
 - ratsubst, 381
 - ratvars, 315, 331
 - ratweight, 331
 - read, 517
 - readline, 460
 - readonly, 517
 - realpart, 302, 308
 - realroots, 27, 571
 - rearray, 432
 - rectform, 366
 - rem, 207, 228
 - remainder, 343
 - remarray, 432
 - remfunction, 492
 - remlet, 282
 - remove, 196, 228, 362
 - remrule, 281
 - remvalue, 361
 - reset, 642
 - reset_verbosely, 642
 - residue, 605
 - rest, 404
 - resultant, 334
 - return, 485
 - reveal, 657
 - reverse, 402
 - rhs, 204, 567

- risch, 26, 595
- risch(ev 関数の引数), 287, 291
- rk, 629
- romberg, 597, 610
- room, 673
- rootscontract, 332
- round, 306
- row, 443
- rreduce, 419
- run_testsuite, 690
- rembox, 659
- S
 - cunlisp, 466
 - save, 42, 520
 - scalarp, 244
 - scaled_bessel_i, 621
 - scaled_bessel_i0, 621
 - scaled_bessel_i1, 621
 - scaled_bessel_y, 622
 - scanmap, 410
 - sconc, 468
 - sconcat, 458
 - scopy, 468
 - scsimp, 394
 - sdowncase, 472
 - sec, 553
 - sech, 553
 - set_partitions, 424
 - set_random_state, 309
 - setdifference, 417
 - setelmx, 443
 - setequalp, 425
 - setify, 415
 - setp, 425
 - setup_autoload, 228, 641
 - showratvars, 362, 875
 - showvars, 331
 - sign, 377
 - similaritytransform, 453
 - simplode, 471
 - simtran, 453
 - sin, 553
 - sinh, 553
 - sinvertcase, 472
 - slength, 468
 - smake, 468
 - smismatch, 477
 - solve, 27, 577
 - some, 241, 243
 - sort, 402
 - specint, 622
 - split, 470
 - sposition, 468
 - sqfr, 340
 - sqrt, 308
 - sremove, 476
 - sremovefirst, 475
 - sreverse, 468
 - ssearch, 478
 - ssort, 476
 - sstatus, 672, 880
 - staircase, 628
 - status, 671, 880
 - stirling1, 551
 - stirling2, 551
 - strim, 468
 - string, 458
 - stringout, 42, 518, 875
 - stringp, 244, 464
 - sublis, 382

- sublist, 404
- submatrix, 445
- subset, 422
- subsetp, 425
- subst, 223, 380, 846, 858, 876
- substinpart, 383
- substpart, 383, 404, 858
- substring, 470
- subvarp, 240
- sum, 217, 218, 370, 371
- sumcontract, 393
- sunlisp, 466
- supcase, 471
- supcontext, 258
- surfplot, 876
- susbst, 474
- susbstfirst, 474
- symbolp, 401
- symmdifference, 417
- system, 41, 682, 872, 876
- sequal, 466
- sequalignore, 466
- T
 - tan, 553
 - tanh, 553
 - taylor, 350, 529
 - taylor_simplifier, 350
 - taylorinfo, 350
 - taylorp, 244, 350
 - taytorat, 349, 350
 - tcl_output, 657
 - tellrat, 323
 - tellsimp, 177, 213, 275
 - tellsimpafter, 177, 275
 - tex, 32, 677
 - texend, 677
 - texinit, 677
 - texput, 677
 - throw, 485
 - time, 673
 - timedate, 675
 - timer, 676
 - timer_info, 676
 - tldefint, 608
 - tlimit, 585
 - to-maxima, 297
 - to_lisp, 47, 297
 - tokens, 473
 - totaldisrep, 315
 - totient, 549
 - tr_warnings_get, 500
 - translate, 229, 500
 - translate_file, 500
 - transplate, 500
 - transpose, 445
 - tree_reduce, 419
 - triangularize, 445
 - trigexpand, 557
 - trigrat, 557
 - trigreduce, 557
 - trigsimp, 23, 557
 - trunc, 346
- U
 - ueivects, 454
 - union, 417
 - uniteigenvectors, 454
 - unitvector, 454
 - unknown, 242
 - unorder, 182
 - untimer, 676

- untrllrat, 325
- uppercasep, 244, 464
- use_fast_arrays, 429
- uvect, 454
- unique, 416
- V
 - verbify, 294, 355
 - viewps, 720
- W
 - with_stdout, 519
 - writefile, 41, 519
- X
 - xreduce, 419
 - xthru, 332
- Z
 - zeroequiv, 239
 - zeromatrix, 435
 - zeta, 538
- 記号
 - $>_m$, 179, 875
 - A_m , 179
 - %i, 646
 - %o, 646
 - expt, 198
 - nexpt, 198
- グラフ
 - 三次元～表示, 36
 - 二次元～表示, 35
 - openmath, 34
- グラフの属性
 - A
 - axis_3d, 795
 - axis_bottom, 795
 - axis_left, 795
 - axis_right, 795
 - axis_top, 795
 - C
 - columns, 789, 793
 - E
 - eps_height, 793
 - eps_width, 793
 - F
 - file_name, 793
 - G
 - grid, 796
 - L
 - logx, 796
 - logy, 796
 - logz, 796
 - P
 - pic_height, 793
 - pic_width, 793
 - R
 - rot_horizontal, 797
 - rot_vertical, 797
 - T
 - terminal, 793
 - title, 796
 - X
 - x_range, 795
 - xtics, 796
 - xy_file, 797
 - xlabel, 796
 - Y
 - y_range, 795
 - ylabel, 796
 - yticks, 796
 - Z
 - z_range, 795
 - zlabel, 796

- Ztics, 796
- 属性
 - 属性値, 206
- A
 - additive, 217
 - algebraic, 190
 - alias, 681
 - alphabetic, 167, 214
 - analitic, 220
 - argpos, 190
 - assign, 510, 511
 - assign-mode-check, 510
 - atomgrad, 224, 227
 - atvalue, 227
 - autoload, 642
- C
 - commutative, 217
 - complex, 216
 - constant, 214
- D
 - decreasing, 220
 - dependency, 224
- E
 - english, 190
 - even, 216
 - evenfun, 220
 - evflag, 213, 287, 288
 - evflag 属性を持つ大域変数, 288
 - evfun, 213, 285, 287, 289
 - evfun 関数の作用の順番, 289
 - evfun 属性を持つ関数, 289
- G
 - gradef, 224, 227
- I
 - imaginary, 216
 - increasing, 220
 - integer, 216
 - irrational, 216
- L
 - lassociative, 217
 - linear, 217
 - logical, 190
 - lpos, 190
- M
 - mainvar, 214
 - matchdeclare, 227, 264
 - mode, 508
 - mode.check_errorp, 509
 - mode.check_warnp, 509
 - mode.checkp, 509
 - multiplicative, 217
- N
 - nary, 83
 - noninteger, 216
 - nonscalar, 213, 214
 - noun, 214
- O
 - odd, 216
 - oddfun, 220
 - outative, 217
- P
 - pos, 190
 - posfun, 220
- R
 - rassociative, 217
 - rational, 216
 - real, 216
 - rpos, 190
- S
 - scalar, 214

- special, 213, 510
- symmetric, 217
- U
 - untyped, 190
- V
 - value_check, 510
- 記号の属性, 214
- 変数の属性, 213
- 数値属性, 216
- 大域変数
- %
 - %%, 648
 - %, 648
 - %e_to_numlog, 565
 - %edispflag, 653
 - %emode, 386
 - %enumer, 386
 - %gamma, 533
 - %iargs, 556
 - %num_list, 576
 - %piargs, 556
 - %enumer, 291
- A
 - absboxchar, 653
 - activecontexts, 259
 - algebraic, 322
 - algedelta, 576
 - algepsilon, 576
 - algexact, 576
 - aliases, 229, 669, 681
 - arrays, 229, 431, 669
 - assume_pos, 260
 - assume_pos_pred, 260
 - assumescalar, 440
 - atomgrad, 224

- B
 - backsubst, 568
 - backtrace, 485
 - berlefact, 342
 - besselarray, 622
 - besselexpand, 622
 - beta_args_sum_to_integer, 532
 - bftunc, 303
 - boxchar, 660
 - breakup, 579
 - bftorat, 303
- C
 - cauchysum, 371
 - cfdlength, 540
 - combineflag, 333
 - compgrind, 503
 - context, 259
 - contexts, 259
 - current_let_rule_package, 280
- D
 - dblnt_x, 613
 - dblnt_y, 613
 - debugmode, 681
 - default_let_rule_package, 280
 - demoivre, 386
 - dependencies, 224, 230, 597, 669
 - derivabbrev, 587
 - derivsubst, 587
 - detout, 440
 - dispflag, 485
 - display_format_internal, 360, 653
 - display2d, 14, 595, 652, 653
 - doallmxops, 441
 - domain, 303, 304
 - domxexpt, 441

- domxmxops, 441
- domxnct, 441
- dontfactor, 342
- doscmxops, 441
- doscmxplus, 441
- dot0nscsimp, 199
- dot0simp, 199
- dot1simp, 199
- dotassoc, 199
- dotconstrules, 199
- dotdistrib, 199
- dotexpsimp, 199, 589
- dotident, 199
- dotsassoc, 589
- dotscrules, 199, 590
- draw_command, 808
- draw_pipes, 808
- E
 - ecm_limit, 546
 - ecm_limit_delta, 546
 - ecm_max_limit, 546
 - ecm_number_of_curves, 546
 - erfflag, 598
 - error, 662
 - error_size, 662
 - error_syms, 662
 - errorfun, 485
 - errormsg, 662
 - expon, 290
 - expop, 290
 - exptdispflag, 653
 - exptisolate, 365
 - exptsubst, 380
- F
 - faceexpand, 342
 - factorflag, 342
 - features, 210
 - file_output_append, 520
 - file_search_demo, 514, 640
 - file_search_lisp, 514, 640
 - file_search_maxima, 514, 640
 - file_search_tests, 640
 - file_search_usage, 514, 640
 - file_search_demo, 663, 665
 - float2bf, 303
 - fortindent, 681
 - fortspaces, 681
 - fppintprec, 302, 303
 - fpprec, 302, 303
 - functions, 171, 489, 669
 - factlim, 535
- G
 - gcd, 337
 - genindex, 371
 - gensumnum, 371
 - globalsolve, 568
 - gnuplot_command, 707
 - gnuplot_view_args, 774
 - gradefs, 224, 230, 669
 - gammalim, 535
- H
 - halfangles, 556
 - help, 665
 - hermitianmatrix, 452
- I
 - iarray, 622
 - ibase, 653
 - in_netmath, 722
 - inchar, 648
 - inflag, 362, 383, 400

- infolists, 668
- integrate_use_rootsof, 600
- integration_constant_counter, 595
- intfaclim, 342
- isolate, 365
- ifactor_verbose, 546
- ifactors_only, 546
- K
 - keepfloat, 319
 - knowneigvals, 452
 - knowneigvects, 452
- L
 - labels, 648, 669
 - lasttime, 674
 - leftjust, 653
 - leftmatrix, 452
 - let_rule_packages, 280
 - let_rule_packages, 670
 - letrar, 280
 - letrat, 277
 - letvarsimp, 280
 - lhospitallim, 584
 - ligarc, 565
 - limsubst, 584
 - linechar, 648
 - linel, 653
 - linenum, 648
 - linsolve_params, 574
 - linsolvewarn, 574
 - lispdisp, 298, 653
 - listarith, 400
 - listconstvars, 363
 - listdummyvars, 363
 - listeigvals, 452
 - listeigvects, 452
- lmxchar, 441
- loadprint, 522
- logabs, 565
- logconcoeffp, 565
- logexpand, 565
- lognegint, 565
- lognumer, 565
- logsimp, 565
- M
 - m1pbranch, 303
 - macroexpansion, 495
 - macros, 670
 - manual_demo, 664
 - manual_demo, 665
 - maperror, 408
 - matrix_element_add, 442
 - matrix_element_mult, 442
 - matrix_element_transpose, 442
 - maxapplydepth, 274
 - maxapplyheight, 274
 - maxima_tempdir, 640
 - maxnegex, 290, 390
 - maxpogex, 390
 - maxposex, 290
 - maxtayorder, 351
 - modulus, 322, 528
 - multiplicities, 568
 - mx0simp, 440
 - mymacros, 670
 - myoptions, 681
- N
 - negdistrib, 386
 - negsumdispflag, 197
 - newfac, 342
 - niceindicespres, 376

- nolabels, 648
- nondiagonalizable, 452
- noundisp, 653
- numer, 386
- O
 - obase, 653
 - opproperties, 209
 - opsubst, 380
 - optimprefix, 500
 - optionset, 681
 - outchar, 648
- P
 - packagefile, 522
 - partswitch, 369
 - pformat, 653
 - piece, 369
 - plot_options, 702
 - ～の colour_z, 708
 - ～の gnuplot_curve_styles, 714
 - ～の gnuplot_curve_titles, 714
 - ～の gnuplot_dumb_term_command, 714
 - ～の gnuplot_pipe_term, 715
 - ～の gnuplot_pm3d, 710
 - ～の gnuplot_preambles, 715
 - ～の gnuplot_term, 712
 - ～の grid, 710
 - ～の nticks, 709
 - ～の plot_format, 707
 - ～の t, 710
 - ～の transform_xy, 708
 - ～の view_direction, 708
 - ～の x, 710
 - ～の xlog, 710
 - ～の y, 710
- ～の ylog, 710
- polar_rho_limit, 546
- polar_rho_limit_step, 546
- polar_rho_tests, 546
- polyfactor, 572
- powerdisp, 653
- prederror, 238, 485
- prevfib, 534
- primep_number_of_tests, 546
- prompt, 646, 648, 659
- props, 206, 221, 226, 671
- ps_scale, 722
- ps_translate, 722
- psexpand, 320
- R
 - radexpand, 303
 - radsubstflag, 381
 - rataigdenom, 319
 - ratdenomdivide, 320
 - ratepsilon, 303, 319
 - ratexpand, 320
 - ratfac, 320
 - ratmx, 440
 - ratprint, 319
 - ratsimpexpons, 320
 - ratweights, 320
 - ratwtlvl, 320
 - readonly, 576
 - resultant, 336
 - rightmatrix, 452
 - rmxchar, 441
 - rombergabs, 612
 - rombergit, 612
 - rombergmin, 612
 - rombertol, 612

- rootsconmode, 320
- rootsepsilon, 572
- rpgrammode, 568
- rules, 230, 269, 671
- S
- save_primes, 546
- savedef, 503
- savefactors, 342
- scalarmatrix, 440
- setcheck, 201, 688
- setcheckbreak, 688
- setval, 688
- show_openplot, 722
- showtime, 674
- simp, 177, 181, 213, 283, 386
- simpproduct, 370, 386
- simpsum, 371
- solve_inconsistent_error, 579
- solvedecomposes, 579
- solveexplicit, 579
- solvefactors, 579
- solvenullwarn, 579
- solveradcan, 579
- solvetrigwarn, 579
- sparse, 440
- sqrtdispflag, 309
- stardisp, 653
- stringdisp, 653
- structures, 487
- sublis_apply_lambda, 383
- sumexpand, 371, 386
- sumsplitfact, 332, 342
- superlogcon, 565
- T
- taylor_coefficients, 351
- taylor_logexpand, 351
- taylor_truncate_polynomials, 351
- taylordepth, 351
- timer_devalue, 676
- tlimswitch, 584, 610
- tr_array_as_ref, 505
- tr_bound_function_apply, 506
- tr_file_tty_messagesp, 506
- tr_float_can_branch_complex, 506
- tr_function_call_default, 505
- tr_numer, 505
- tr_optimize_max_loop, 506
- tr_semicompile, 505
- tr_warn_bad_function_calls, 505, 506
- tr_warn_fexpr, 505
- tr_warn_meval, 505
- tr_warn_mode, 505
- tr_warn_undeclared, 505
- tr_warn_undefined_variable, 505
- trace, 688
- trace_break_arg, 688
- trace_max_indent, 688
- trace_safety, 688
- transcomile, 505
- translate, 503
- translate_fast_arrays, 505
- transrun, 503
- trigexpandplus, 556
- trigexpandtimes, 556
- triginverses, 556
- trigsign, 556
- ttyoff, 653
- U
- undeclaredwarn, 503

- undeclearewarn の設定項目, 504
- V
 - values, 173, 230, 361, 671
 - vect_cross, 592
- W
 - window_size, 722
- Y
 - yarray, 622
- Z
 - zeobern, 531
- features, 230
- 対象
- E
 - ellipse, 805
 - ～構文, 805
 - ～の属性, 805
 - ～の border, 805
 - ～の color, 805
 - ～の fill_color, 805
 - ～の key, 805
 - ～の line_type, 805
 - ～の line_width, 805
 - ～の nticks, 805
 - ～の transparent, 805
 - ～文, 805
 - explicit, 797
 - ～の構文, 797
 - ～の属性, 798
 - ～の adapt_depth, 798
 - ～の color, 798
 - ～の fill_color, 798
 - ～の filled_func, 798
 - ～の key, 798
 - ～の line_type, 798
 - ～の line_width, 798
 - ～の nticks, 798
 - ～文, 797
 - explicit3d, 799
 - ～の構文, 799
 - ～の属性, 800
 - ～の color, 800
 - ～の contour, 800
 - ～の contour_levels, 800
 - ～の enhance3d, 800
 - ～の key, 800
 - ～の line_type, 800
 - ～の line_width, 800
 - ～の xu_grid, 800
 - ～の yv_grid, 800
- G
 - gr2d, 788
 - gr3d, 788
- I
 - image, 806
 - ～構文, 807
 - ～の属性, 807
 - ～の colorbox, 807
 - ～の palette, 807
 - ～文, 806
 - implicit, 800
 - ～の構文, 800
 - ～の属性, 801
 - ～の color, 801
 - ～の ip_grid, 801
 - ～の ip_grid_in, 801
 - ～の key, 801
 - ～の line_type, 801
 - ～の line_width, 801
 - ～文, 800
- L

label, 807
～構文, 807
～の属性, 807
～の label.allgment, 807
～の label.orientation, 807
～文, 807
label3, 807
～構文, 807

P

parametric, 801
parametric_surface, 802
～構文, 803
～の属性, 803
～の color, 803
～の key, 803
～の line.type, 803
～の line.width, 803
～の xu_grid, 803
～の yv_grid, 803
～文, 802
～の構文, 801
～の属性, 801
～の color, 801
～の key, 801
～の line.type, 801
～の line.width, 801
～の nticks, 801
～文, 801
parametric3d, 801
～の構文, 802
～の属性, 802
～の color, 802
～の line.width, 802
～の key, 802
～の line.type, 802
～の nticks, 802
～文, 801
points, 803
points3d, 803
～の属性, 804
～の color, 804
～の key, 804
～の line.type, 804
～の line.width, 804
～の point.ljoined, 804
～の point.size, 804
～の point.type, 804
～構文, 803
～文, 803
polar, 802
～の構文, 802
～の属性, 802
～の color, 802
～の key, 802
～の line.type, 802
～の line.width, 802
～の nticks, 802
～文, 802
polygon, 804
～の属性, 805
～の border, 805
～の color, 805
～の fill.color, 805
～の key, 805
～の line.type, 805
～の line.width, 805
～の transparent, 805
～の構文, 804
～文, 804

R

～文, 804
 rectangle, 804
 ～の構文, 804

V

vector, 805
 ～の構文 (2d) , 806
 ～の構文 (3d) , 806
 ～の属性, 806
 ～の color, 806
 ～の head_angle, 806
 ～の head_both, 806
 ～の head_length, 806
 ～の head_type, 806
 ～の key, 806
 ～の line_type, 806
 ～の line_width, 806
 ～文, 805
 vector3d, 805

定数

%e, 291, 304
 %gamma, 304
 %phi, 304
 %pi, 304
 false, 169, 304
 inf, 304, 597
 infinity, 304, 597
 minf, 304
 off(=false), 169
 on(=true), 169
 true, 169, 304
 unknown, 169
 zeroa, 304
 zerob, 304

内部関数

kind, 216

\$quit, 47
 assign-mode-check, 511
 defprop, 218, 225, 283, 677
 defprop 函数の operators 属性, 386
 defprop の operator 属性, 283
 kind, 210, 220
 kindp, 220
 lisp-implementation-type, 688
 lisp-implementation-version, 688
 oper-apply, 213
 par, 216
 prop1, 207
 simplifya, 213, 274, 386

内部変数

A

alphabet, 167, 229
 autoconf-host, 688
 autoconf-version, 688

F

features, 210, 671
 fixnbound, 168
 flounbound, 169

M

maxima-build-time, 688
 maxima-epilog, 683
 maxima-tesitdir, 690
 maximae-demodir, 664

O

*opers-list, 212
 opers, 211

R

ratweights, 644

U

unbound, 222

V

variable-initial-values, 643

ファイル

AlexanderPolyl.mc, 855

fox.mc, 851

maxima-init.mac, 854, 873, 994

maxima-init.mac の置き場所, 998

ponpoko.lisp, 825

surfplot, 884

surfplot.mc, 876

初期化ファイル(maxima-init.mac),
873

文脈

～の生成, 258

～の切替, 250, 259

～の削除, 259

～の生成, 258

～を無効にする, 258

～を有効にする, 258

親文脈, 258

global, 257

initial, 24, 257

も

文字, 167

か

関係, 70

し

順序数, 97

A

α 同値, 156

ambient isotopy, 836

atom, 166

B

Bell 数, 528

C

Church

～の真理値, 122

～の提唱, 162

concept, 148

D

Dedekind 無限集合, 94

Dehn

～の補題, 843

～表示, 837

De Morgan の法則, 80, 114

E

EOF, 51

external import, 111

F

Fox の微分子, 837, 844

H

Hilbert 計画, 104, 105

Honer 則, 327

Hume の原理, 134

Hurwiz の定理, 860

K

Klein の壺, 36

L

λ - 計算, 59

M

Mersenne Twister 法 (MT 法), 310

Möbius の輪, 37

monic な多項式, 91

N

New Math, 162

P

Peano の公理, 95

PID(=主イデアル整域), 77

Poincaré 予想, 843

Q

QEMU

qemu-img, 983

S

scope, 156

Seifert 曲面, 867

SQUARE, 110

T

term, 148

Tietze 変換, 837, 844

V

VirtualBox, 969

仮想計算機の起動, 980

仮想計算機の生成, 971

VMWare

mdk ファイル, 983

vmx ファイル, 984

W

Wirtinger 表示, 837

Z

Zorn の補題, 160

い

意義, 121

イデアル, 77

意味, 121

因明, 106

え

演算

～が閉じている, 72

演算子, 17

演算子項, 170

お

オンラインマニュアル, 15

か

外延, 122

外延的定義, 68

概念, 148

概念 (Begriff, concept), 122

Γ -概念, 122

概念記法, 117

移行記号, 120

条件線, 118

水平線, 117

対偶変換, 129

内容線, 117

判断線, 117

解の自動代入, 569

可換, 73

可換環, 76

可換群, 73

可換積, 17

核, 75

可算個, 93

可算濃度, 93

可算無限集合, 93

可附番集合, 93

可附番濃度, 93

環, 76

含意, 154

関係, 123

関係子, 75, 837

還元可能性公理, 102, 172

還元可能性公理 (=還元公理), 149

函数型, 7

函数項, 170

非可換, 73

非可換群, 73

き

記号, 167

基数, 68, 97

基数 (濃度), 93

規則, 262

～の削除, 280
帰納法の原理, 96
帰謬法, 103
基本命題 (=Elementary proposition), 148
逆元, 72
逆理
 Cantor の～, 100
 Russell の～, 99
 クレタ人の～, 99
 床屋の～, 100
強先祖関係, 141
共通集合, 68
行列, 29
 ～の固有値の計算, 32
 ～の特性多項式, 32
 ～の差, 29
 ～の積, 30
 ～の定義, 29
 ～の和, 29
 ～の逆行列の計算, 31
極, 7
局所環, 77
極大イデアル, 77
く
 クラス, 122
 群, 863
 群環, 78
け
 形式主義, 101
 結合律, 72
 原子, 297
 原子 (atom), 166
 原始帰納的函数, 160
こ

語, 74
項, 148, 174
後者, 96
項順序, 181
個体 (=individual), 148
さ
 最小多項式, 91
 差集合, 69
 作用範囲, 123, 156
 三段論法, 111
 仮言三段論法, 111, 128
 後件肯定, 120
 選言三段論法, 111
 前提肯定, 119
 定言三段論法, 111
 ディレンマ, 111
 両刀論法, 111
し
 CRE 表現, 313
 式, 173
 式の並び, 262, 263
 順序
 ～の定義, 85
 辞書式順序, 180
 (Maxima の) 次数リスト, 180
 弱先祖関係, 141
 斜体, 80
 主イデアル整域, 77
 自由群, 74, 837
 重積値域, 123
 自由変項, 173
 自由変数, 173
 述語
 tautology(=恒真式), 231
 恒真式, 231

- 充足可能な述語, 231
- 述語関数, 149
- 主変数, 214
 - mainvar, 179
 - ～の宣言, 214
- 準群, 72
- 順序, 84
 - ～集合, 85
 - 逆辞書式順序, 87
 - 項順序, 86
 - 斉次逆辞書式順序, 87
 - 斉次辞書式順序, 86
 - 辞書式順序, 86
 - 全順序, 85
 - 全順序集合, 85
- 準同形写像, 75
- 商環, 80
- 常微分方程式, 28
- 剰余環, 80
- 初期条件, 29
- 真部分集合, 68
- 真理関数, 175, 238
- す
- 推移律, 70
- 数学的帰納法, 96
- 数学の現代化, 162
- スケイン関係式, 867
- スタック型言語, 49
- スタックマシン, 49
- せ
- 整域, 76
- 正規部分群, 75
- 正準表現, 88
 - 多変数多項式の正準表現, 89
- 生成元, 74
- 正則元, 72
- 整列集合, 97, 160
- 整列性, 97
- 選言記号, 154
- 前件肯定, 120
- 選択公理, 159
- 前置表現, 83
- 全微分, 586
- そ
- 素イデアル, 77
- 属性, 172
 - 作用に関連する属性, 217
 - 演算子の属性, 220
 - 関数の属性, 220
 - ～を追加, 210
- 属性の表現関数, 177, 206
- 束縛変項, 123, 173
- 束縛変数, 173, 360
- 存在含意, 111
- た
- 体, 80
- 大域変数, 173, 174
- 対角線論法, 94
- 対象, 166
- 対称律, 70
- 代数的数, 91
- 代数的整数, 71, 91
- 代数方程式, 91
- 対当
 - 小反対対当, 111
 - 大小対当, 110
 - 反対対当, 111
 - 矛盾対当, 111
- 対当関係表, 110
- 多元環, 78

- 多項式と単項式の一般表現, 312
- 多様体, 844
- 単位元, 72
- 単項イデアル, 77
- 単変数多項式の CRE 表現, 313
- 断面, 885
 - ～の方程式, 885
- 単連結, 844
- ち
- 値域, 122
- チートコード, 986
- 超越数, 92
- 超限順序数, 99
- 重複度のリスト, 569
- 直感主義, 101
- つ
- 対集合, 158
- て
- 手続型, 8
- と
- 同値関係, 70
- 同値類, 70
 - ～の代表, 70
- トーラス, 37
- な
- 内挿表現, 83
- 内部関数, 171
- 内部変数, 174
- 内包的定義, 68
- 並びの照合, 263
- の
- 濃度, 97
- 濃度 (基数), 93
- は
- 媒介変数式, 698
- 排中律, 103, 112
- 背理法, 103
- パターン, 263
- パターンマッチング, 263
- バッチ処理, 14
- バッチファイル, 14
- 半群, 72
- 反射律, 70
- 半順序, 85
- 判断
 - 肯定判断, 110
 - 全称肯定判断, 110
 - 全称判断, 110
 - 全称否定判断, 110
 - 特称肯定判断, 110
 - 特称判断, 110
 - 特称否定判断, 110
 - 否定判断, 110
- ひ
- 被演算子, 170
- 非可換積, 17
- 左イデアル, 77
- 左分配律, 76
- 筆頭項, 90
- 否定記号, 154
- ふ
- 部分式, 174
- 部分集合, 68
- 分配律, 76
- 文脈, 24, 246
- 文脈原理, 121
- 分離規則, 120
- へ
- 平面
 - 平面の方程式, 885

- 冪集合, 68
- 変項, 170, 173
- 変数, 170, 173
 - ～順序, 180, 875
- ほ
- 方程式, 27, 567
- 補集合, 69
- 母体, 148
- み
- 右イデアル, 77
- 右分配律, 76
- む
- 矛盾律, 112
- 結び目
 - ～群の表示, 837
 - ～の Alexander 行列, 854
 - ～の Alexander 多項式, 854
 - ～の Alexander 多項式, 837
 - ～の Reidemeister 移動, 838
 - ～の下道, 838
 - ～の交点の総数, 836
 - ～の射影図, 838
 - ～の射影図, 836
 - ～の上道, 838
 - ～の正則な射影図, 838
 - ～の符号和, 840
 - ～の連結和, 863
 - 順な結び目, 836
 - 野性的な結び目, 836
- 無平方, 340
- 無平方因子分解, 340
- も
- 木構造, 83
- 文字列, 168
- や
- 約分, 70
- ゆ
- 有限基数, 141
- 有限順序数, 99
- 有限の立場, 104
- 有理数, 70
- り
- 両側イデアル, 77
- れ
- 零因子, 76
- 列, 168
- 連言記号, 154
- 連立方程式, 27, 567
- ろ
- ローマ曲面, 885
- 論理型, 8
- 論理主義, 101
- わ
- 和集合, 68
- 記号
 - $+$, 73
 - $*$, 73
 - $\mathbb{R}[x]$, 71
 - $\sim$, 70
 - $\cdot$, 73
 - $\ker$, 75
 - LM, 91
 - LT, 91
 - ω , 99
 - $\aleph$, 93
 - $\aleph_0$, 93
- 人名
 - B
 - Boole, 101, 114
 - Brouwer, 103

- D
De Morgan, 113
Dehn, 837
- F
Frege, 95, 101, 117
- H
Hilbert, 102–104, 153
- K
Kronecker, 103
- L
Leibniz, 95, 112
- N
Newton, 113
- P
Peano, 95, 101
Poincaré, 103
- R
Russell, 102, 145
- S
Schelter, 2
Steiner, 884
- W
Wirtinger, 837
- に
西村拓士, 310
- ぼ
墨子, 106
- ま
松本眞, 310
- Software
Application
REDUCE, 8
AutoCAD, 46
AXIOM, 8
Cinderella, 964
Common Lisp, 2
DERIVE, 8
Gap, 960
GCL, 2
Geomview, 39, 694
GNU Emacs, 46
gnuplot, 12, 38, 694
imaxima, 12
kan/sm1, 49
kseg, 964
Macaulay2, 12, 960
MACSYMA, 2
Maple, 8, 46
MathCAD, 8
Mathematica, 8, 46
MATLAB, 8
MuPAD, 8
Octave, 12, 905
openmath, 39, 694
PARI/GP, 12, 961
QEMU, 983
R, 8
Risa/Asir, 961
S, 8
SciLab, 905
SINGULAR, 184, 887, 960
S-PLUS, 8
surf, 41, 871
TeXmacs, 12
VirtualBox, 3, 969
VMware Player, 3, 983
wxMaxima, 10
XaoS, 962
xmaxima, 10
yorick, 905

- カルキング, 8
- OS
 - Debian, 3, 4
 - KNOPPIX, 3
 - KNOPPIX/Math, 2, 3
 - KNOPPIX Edu, 2
 - Linux, 2
 - MacOSX, 2
 - MS-Windows, 2
 - OpenSUSE, 4
 - RedHat, 3
 - Slackware, 3
 - Solaris, 2
 - Vine, 4
- 言語
 - Allegro Common Lisp, 66
 - CLISP, 2, 47
 - CMUCL, 66
 - FORTRAN, 46
 - GCL, 47
 - InterLisp, 46
 - KCL(Kyoto Common Lisp), 47
 - LISP, 46
 - MACLisp, 46
 - openMCL, 66
 - PostScript, 49
 - Prolog, 5
 - PS, 49
 - SBCL, 66
 - Scheme, 47
- GNUPLOT
 - 演算子
 - ~, 768
 - *, 765
 - **, 765
 - +, 765
 - , 765
 - /, 765
 - %, 765
 - !=, 768
 - !, 765, 768
 - <=, 768
 - <, 768
 - =, 768
 - >=, 768
 - >, 768
 - &&, 768
 - &, 768
 - ||, 768
 - !, 768
 - {}, 771
 - 記号
 - '-', 726
 - 数学函数
 - abs, 766
 - acos, 766
 - acosh, 766
 - arg, 766
 - asin, 766
 - asinh, 766
 - atan, 766
 - atan2, 766
 - atanh, 766
 - besj0, 767
 - besj1, 767
 - besy0, 767
 - besy1, 767
 - ceil, 767
 - cos, 766
 - cosh, 766

- erf, 767
- erfc, 767
- exp, 766
- floor, 767
- gamma, 767
- ibeta, 767
- igamma, 767
- imag, 766
- int, 767
- invert, 767
- invnorm, 767
- lambertw, 767
- log, 766
- log10, 766
- norm, 767
- rand, 766
- real, 766
- sgn, 767
- sin, 766
- sinh, 766
- sqrt, 766
- tan, 766
- tanh, 766
- A
 - arrow, 763
 - at, 729
 - auto, 741
- B
 - b, 729, 731
 - both, 738
 - bspline, 741
- C
 - cbrange, 746
 - clabel, 746
 - cntrparam, 740
 - contour, 738
 - base, 738
 - both, 738
 - surfacee, 738
 - cubicspline, 741
- H
 - hidden3d, 748
- I
 - if, 771
 - if 文の構文, 772
 - incremental, 742
 - isosamples, 754
- K
 - key, 726, 759
- L
 - label, 761
 - levels, 741
 - linear, 741
- M
 - map, 731
- P
 - palette, 734
 - gray, 735
 - negative, 735
 - positive, 735
 - rgbformulae, 737
 - plot, 725
 - dots, 729
 - impulse, 729
 - lines, 729
 - linespoints, 729
 - point, 729
 - plot 命令の構文, 725
 - pm3d, 729
 - pm3d の設定, 729

- print, 770
- R
 - replot, 735
 - reread, 771
- S
 - s, 730
 - sample, 754
 - set, 725
 - show, 725
 - splot, 728
 - splot 命令の構文, 728
 - surface, 738, 748
- T
 - term, 778
 - ticslevel, 733
 - title, 728
- U
 - unset, 725
- V
 - view, 743
- W
 - with, 728
- X
 - xrange, 756
- Y
 - yrange, 756
- Z
 - zrange, 756
- い
 - 陰線処理, 748
- か
 - 階調の調整, 746
 - 関数の定義, 768
- き
 - 曲面を面を張る, 730
 - 曲面の上面への投影, 731
 - 曲面の底面への投影, 729
 - 曲面の様式, 728
- し
 - 条件分岐, 771
 - 真理値, 770
- せ
 - 整数, 770
- と
 - 等高線, 731
 - 等高線の高度, 742
 - 等高線の設定, 738
 - 等高線の調整, 741
 - 等高線の補間式, 741
- は
 - 反復処理, 771
 - 凡例, 726
- ひ
 - 表題, 728
- ふ
 - 複素数, 771
 - 浮動小数点, 770
- へ
 - 変数, 728
- LISP
 - A
 - acos, 50
 - and, 54
 - append, 53
 - apply, 59
 - aref, 55
 - asin, 50
 - atan, 50
 - C
 - caar, 54

- cadr, 54
- car, 53
- cddr, 54
- cdr, 53
- char-code, 52
- close, 63
- code-char, 52
- concatenate, 51
- cond, 54, 60
- cons, 54
- cos, 50
- D
 - defun, 52, 60
 - deof, 61
 - dribble, 41
- E
 - eval, 57
 - exp, 50
- F
 - format, 64
- G
 - get, 61, 188
 - get-decoded-time, 675
 - get-internal-real-time, 675
 - get-universal-time, 675
 - gethash, 56
 - great, 179, 183
- I
 - :initial-element, 55
 - if, 54, 60
 - inof, 61
- L
 - lambda 式, 59
 - let, 57
 - let*, 58
 - load, 62
 - log, 50
- M
 - make-array, 55
 - make-hash-table, 56
 - mapcar, 59
 - most-negative-double-float, 169
 - most-negative-fixnum, 168
- N
 - nil, 51
- O
 - open, 63
 - or, 54
- P
 - pop, 61
 - prin1, 64
 - princ, 64
 - print, 64
 - push, 61
 - P リスト, 61
 - ut, 191
- R
 - read, 64
 - room, 673
- S
 - setf, 55, 56, 61
 - setq, 55, 56
 - sin, 50
 - string, 51
 - symbol-plist, 61
 - S 式, 50
- T
 - t, 51
 - tan, 50
- LISP の記号

*, 50
 +, 50
 -, 50
 /, 50
 あ
 原子, 48
 え
 S 式, 297
 き
 局所変数, 57
 さ
 三角関数, 50
 し
 指数関数, 50
 四則演算, 50
 述語, 54
 剰余, 50
 そ
 属性リスト, 61
 た
 対数関数, 50
 は
 配列, 55
 ハッシュ表, 56
 ふ
 複素数, 50
 も
 文字データ, 51
 文字列, 51
 り
 リスト, 50, 52
 ろ
 論理積, 54
 論理和, 54
 SINGULAR

E
 eliminate, 892
 G
 groebner, 888
 L
 LIB, 890
 M
 map, 889
 P
 preimage, 890
 S
 setring, 889
 std, 888
 surf.lib, 890
 い
 イデアル, 887
 イデアルの定義, 887
 き
 基礎環の係数体, 887
 基礎環の定義, 887
 し
 写像, 889
 商環, 887
 商環の定義, 887
 た
 多項式, 887
 多項式の定義, 887
 れ
 零イデアル, 890
 Octave
 表記
 $a(:, j)$, 917
 $a(i, :)$, 917
 $a(i, j)$, 916
 $a(i:j)$, 917

A

all, 927
any, 927

C

cd, 935
conv, 931

D

deconv, 931
diag, 928

E

e, 907
eps, 909
exec, 934
eye, 929

F

fclose, 946
fgets, 947, 948
find, 922
fopen, 946, 948
for, 907
for 文による反復処理, 914
fprintf, 955
frewind, 947, 948
fscanf, 947

G

gnulot, 936
grid, 936

H

help, 906
hold, 936

I

i, 907
is_struct, 953

L

length, 916

load, 940

load -force, 941
ls, 935

M

M-file, 906, 932

O

Octave の演算, 910

P

pi, 907
plot, 936
polyval, 931
pwd, 935

R

rand, 914

S

save, 942
save -append, 944
size, 916
sscanf, 948
sscanf の型の指定, 949
struct_elements, 953
system, 934

T

title, 936
type, 908, 933

W

while, 909
who, 908, 933, 942

X

xlabel, 936

Y

ylabel, 936

Z

zlabel, 936

Octave の記号

- \, 910
- ’, 910
- *, 910
- +, 910
- , 910
- .*, 910
- ./, 910
- .^, 910
- /, 910
- ~, 910
- !, 934
- ., 916
- :, 917, 918, 922
- ;, 906, 916
- =, 907
- お
 - オンラインヘルプ, 906
- き
 - 行列の書式, 915
 - 行列の成分, 916
 - 偽, 922
 - 行列, 915
- く
 - 組込函数, 933
- グラフ
 - グラフに網目, 936
 - グラフにラベルや表題を入れる, 936
 - グラフの重ね描き, 936
- け
 - 計算時間, 914
- こ
 - 構造体, 953
- し
 - 四則演算, 906
- 真, 922
- た
 - 多項式の商と剰余, 931
 - 多項式の積, 931
 - 多項式の変数に値を代入, 931
- な
 - 並びの照合, 922
- ふ
 - ファイルの close, 946
 - ファイルの open, 946
 - ファイルの読込, 940
 - ファイルへの保存, 942
 - 不等号, 923
- ほ
 - ポインタ, 947
- surf
 - C
 - curve, 876
 - D
 - draw_curve, 876
 - draw_surface, 876
 - E
 - epsilon, 872
 - H
 - height, 872
 - I
 - iterations, 872
 - R
 - root_finder, 872
 - rot_x, 873
 - rot_y, 873
 - rot_z, 873
 - S
 - scale_x, 873
 - scale_y, 873

scale_z, 873

surface, 876

W

width, 872