大学一年生のための MuPAD 入門

幸谷 智紀[1]　　角谷　悟[2]

March 13, 2000
Version 0.0

[1]静岡理工科大学
[2]千葉県立沼南高等学校

初めに

0.1 数式処理ソフトウェアとは？

世の中には数学を蛇蝎の如く嫌悪している人達が大勢います。運動の苦手な人が，体育祭の前日に世界の破滅を渇望するのと同様，数学が苦手な人は黒板に踊る $\sum$ や $\log$ や $\sin, \cos, y = x^3 + 2x^2 + 7$ という記号は単なる記号以上の意味を持たず，午睡を誘うものでしかありません。そういう人達にとっては数学が得意な，所謂「理系」と分類される人種は理解の範囲外であるかもしれません。

しかし，理系の人達が全て数学全般を得意とするかというと，これが案外違うのです。今では若年者人口が減少し，大学の数も雨後の竹の子，蜘蛛の子を散らすように増えましたから，昔よりも受験勉強に血道を上げるという光景は少なくなりました。それでもなお，それなりに社会的地位が高いとされる偏差値の高い大学へ進学しようとすると相応の受験勉強をし，他者を蹴落とさなければなりません。

その受験勉強全体に言えることですが，これは入学試験という極めて特殊な問題を，限られた時間内に効率的に解くための訓練に他なりません。そして，入学試験問題は文部省の指導要領が定めた「高校で習う範囲」をベースに作成されなければなりません。従って，学習した内容に対する深い理解を求めたり，範囲を逸脱するような出題は出来ませんから，多くは一定数のパターンに分類できる類型的な問題が多くなります。河合塾や代々木ゼミナール，駿台予備校などの大手受験産業ではこのパターンを分析し,「東京大学の問題はこんな感じ」という模擬試験や参考書を作成します。そして受験生はこの手の文献を参考に，限られたパターンの問題を繰り返し解く機械的な訓練を重ねることで，自らの偏差値を向上させていくわけです。

数学に関して言うと，ある程度の計算の練習必要不可欠で，このような受験勉強も「それなりに」役には立ちます。しかし，過度の受験勉強をしてきた「理系」人間には，逆に「パターンにはまらない」問題は全くお手上げになるケースが多いのです。むしろ，計算はそんなに速くは出来ないし正確さにも欠ける一見「グズ」な人の方が，自分のペースですらっと解いてしまうことがままあります。以下，全て数学に関してだけ述べさせていただきます。

「パターンにはまらない」問題というのは何でしょう？　一概に述べることは出来ませんが，漠然とした表現を使えば，

1. 問題の意図するところを正しく理解できる読解力

2. 深い推論・推察能力

を要求していることが多いと思われます。例えて言うと，有名中学校の入試問題のようなものです。小学校の算数で習うことはたかが知れていますので，当然入試問題も少ない知識があれば解けるようになっていなければなりません。決して「任意の Hermit 行列の固有値は実数であり，かつ

対角化可能であることを証明せよ」という未知の知識を問うような問題は出題できません。しかし「有名」中学ですから，受験生は殺到する，親は何とかして子供に合格して欲しいと望む[1]。中学側にしてみればその中から他より優れた子供を入学させて有名中学としての威厳と経営を保ちたい，と考えます。従って，入試問題のレベルを上げるには，限られた知識を生かして高度な推論を必要とするようにするしかありません。この推論能力を持つ人は年齢に関係なく，そんなに多くはありません。有名中学の入試問題が，大人ですら解くのが難しいのはそのためです。よって，この手の問題が解けるような小学生は凡百の大人よりもずっと頭が良いと言えます。パターンにはまらない問題の多くはこのような高度な推論を必要とする問題なのです[2]

結論から先に言うと，パターンにはまった問題しか解けないというのでは，人間としての価値は非常低いといわざるを得ません。よく言われていることですが，世間一般で求められている能力は，パターンにはまらない問題を解くことなのです。そして，計算練習などの機械的なパターンの繰り返し訓練が「それなりに役に立つ」のは，それ以外の問題を解くための基礎になるからです。同時に，それ以上の意味を持つことはあまりないのです。

「理系」に分類された人が進学する大学の学部・学科の大半は，工学部 (機械・電子・電気・情報など)，理学部（物理・化学・生物・数学など)，農学部，医学部です。近年の金融の世界は経済学＋情報＋工学というような世界ですから，これも進学先の一つに数えられるかもしれません。また，卒業後の進路もこれらの専門知識[3]を生かせる企業に就職することが多いでしょう。そういったところで数学を応用する場合，グラフを描くだとか方程式を解くといった作業は枝葉末節に過ぎません。とある現象に共通して見られる法則を表現する公式や定理，関係を見出すために用いられます。このときに必要になるのが推論能力であり，パターン通りの計算はそれを支えるためのデータを作り出すために用いられます。従って，枝葉末節は効率重視で片付けるのが望ましいわけで，もはや人間がいちいち手計算でやるものではなく，殆どはコンピュータの力を借ります。近年のコンピュータ，特にパソコンの能力の増加は著しいものがあるので，未知数が万，億の単位の連立一次方程式を解くことも可能です。現実もそれに即して，さらに膨大な計算を要求するようになってきてます。もはや，計算は人間が自力で解決レベルのものではありません。人間が行うべき仕事は，コンピュータには不可能な，手探りの推論に集中することなのです。

数学に関して言うと，この傾向はさらに強まりました。当初は電卓に出来るような四則演算とその膨大な組み合わせによる計算だけをさせていたのが，$(a+b)^2 = a^2 + 2ab + b^2$ という代数計算も含めて，単純作業は全てコンピュータにさせることができるようになりました。前者の仕事は専らプログラミング言語と使って記述されますが，後者はプログラミング言語に似た記述も可能な別のシステムによるものです。これを「数式処理ソフトウェア (Computer Algebra system)」と呼びます。これらの違いについては次の節で詳しく述べることにします。

数学における単純計算はもうコンピュータでこなしてしまう時代になりました。Internet の普及に伴い，様々な情報がやりとりされる中，コンピュータを使った計算技術も格段の進歩を遂げつつあります。これからの時代，求められるのは単なる計算技術ではなく，計算技術をこなすための思考能力なのです。

しかし，誤解して欲しくないのは，小・中・高で学んできた数学における計算そのものが不必要

[1] 子供がどう考えているかということは大抵考慮されません。

[2] 単に意表をつく問題，というものもあります。例えば「$\sin x$ の定義を述べよ (東京大学)」というようなものです。

[3] たかが大学四年ぐらい勉強したからといってそれを専門と称するのはどうかという気もしますけどね。長けりゃ良いというものでもないですけど。

になるわけではないということです。全くの白紙状態の頭脳で思考能力が身に付くはずはありません。世間的には必要がないと思われる基礎訓練を素材としてこねくり回すことによって，初めて考える力が生まれてくるのです。数式処理ソフトウェアといえども人間が作ったものですから，間違いは起こり得ます。その誤りを正すことは人間にしかできず，そのためにも算数・数学の重要性が増すことはあれ，減ることはないのです。皆さんは，そこで単なるテクニックではない，なぜそのようなテクニックを使わなければ解けないのかという思考のプロセスをしっかり学ばなければなりません。先に挙げた受験勉強における「パターンにはまった」問題だけしかできない「理系」人間では，膨大な計算処理能力を持つコンピュータの足元にも及ばないのです。

以下の章では，数式処理ソフトウェアの一種である「MuPAD」を使って，その機能の一端に触れていきます。コンピュータを自分の思考を助ける有用な道具として使いこなすためにも,「パターンにはまらない」問題を解く練習として，高校で学んだ数学を題材に人間らしい試行錯誤を重ねてみてください。多少なりとも「これは役に立ちそうだ」ということが体感できれば，著者として非常に満足です。

ご健闘をお祈りいたします。

0.2 プログラミング言語と数式処理ソフトウェア

ここでは，プログラミング言語と数式処理ソフトウェアの役割と相違点について述べます。

高校の普通科で使用される数学の教科書には，コンピュータを操作して図を描かせたり，簡易電卓でな面倒な一連の計算をさせたりする説明が載っています。コンピュータに出す指令は，人間にもある程度中身の類推がつき，かつコンピュータにも都合のよい言葉で記述されています。この言葉をプログラミング言語，プログラミング言語で記述された一連の命令のひとまとまりをプログラムと呼びます。プログラミング言語は用途別に様々なものが開発され，常に進化してきました。教科書で採用されているプログラミング言語は BASIC[4] というもので，1964 年，アメリカダートマス大学のジョンケメニーとトマスカーツによって発明されたものです [2]。以後，幾つかの亜流に分派していき，皆さんが教科書でお目にかかる BASIC は 1970 年代にビル・ゲイツとポールアレンが開発したものが元になっています [1]。ここでは便宜的に教科書 BASIC と呼ぶことにします。現在ではこの教科書 BASIC とは大分仕様の異なる BASIC が主流ですが，教科書 BASIC と文法上の違いはあれ，基本機能の違いはそれほどありません。この節では，コンピュータ言語の例として教科書 BASIC を使うことにします。

コンピュータが直接理解できる言葉は，特定の 2 進数の列，ビットパターンと呼ばれるものだけです。これを機械語と呼びます。機械語を人間が覚えて命令を書くのは大変な労力を要する上，機械語の命令一つ一つ行う作業はごく限定されたものです。キーボードから打ち込まれた一文字をディスプレイに表示させるという，人間から見れば簡単な作業も，機械語でコンピュータに命令しようとすると,「キーボードから入力された文字を格納してあるメモリを読み出し，一端 CPU に持ち込んで，それをもとに文字の字体データを別のメモリから読み出し，ディスプレイに表示されるためのまた別のメモリに読み出した字体データを書き込め。」といった具合で，一から十まで全てを事細かに指示してやらねばなりません。そこで，使用頻度は高いが機械語で記述すると長くなる

[4] Beginner's All-purpose Symbolic Instruction Code の略称です。

一連の手続きをひとまとめにして覚えやすい名前を付け，さらにこれらをまとめてライブラリとして提供したり，人間に読みやすい記述が出来るよう英語に似た文法構造を導入し，それを機械語に翻訳できるプログラム，いわゆるコンパイラやインタープリタを追加したりしました。プログラミング言語は，このライブラリやコンパイラ/インタープリタの助けを借りて，間接的にコンピュータに仕事をさせることが出来ます。直接機械語でやりとりするよりも作業工程が多くなりますが，複雑な処理を少ない入力文字数で記述できますから，プログラムの開発はこのようなプログラミング言語を使って行うのが普通です。

機械語よりは楽になっても，プログラミング言語はあくまでコンピュータに指令を出すためのものですから，プログラムの中身はコンピュータが行う作業を思い描きながら作り込んでいくことに変わりありません。複雑な処理はその複雑さに比例して分量が増えていきます。

これを例で見ていくことにしましょう。図 1 はリサージュ曲線を描く教科書 BASIC のプログラムです [8]。

```
100 H=639 : V=199 : U=160 : W=80
110 PI=3.14159
120 LINE (0,V/2)-(H,V/2)
130 LINE (H/2,0)-(H/2,V)
140 INPUT "m,n=";M,N
150 T=0 : GOSUB 1000 : PSET (P,Q)
160 FOR T=0 TO 2*PI STEP 0.02/(M+N)
170   GOSUB 1000 : LINE -(P,Q)
180 NEXT
190 END
1000 X=SIN(M*T) : Y=SIN(N*T)
1010 P=H/2+X*U : Q=V/2-Y*W
1020 RETURN
```

図 1: 教科書 BASIC プログラムの例

プログラム本体は行番号 100〜190 までで，1000〜1020 は行番号 150 の「GOSUB 1000」から呼ばれる部分です。ここでは特に説明はしませんが，リサージュ曲線を描くには面倒な手続きを記述する必要があることはご理解いただけると思います。これが 3 次元の図形であったり，x 軸，y 軸に目盛りをふるとなると更に命令を追加しなければなりません。

プログラミング言語の用途は，より使い勝手のよいプログラムを作ることにあります。これをアプリケーションプログラム，いわゆるソフトウェアと呼びます。使い勝手のよいソフトウェアとは，

- 利用者にわかりやすい画面構成であること
- 操作方法が簡単であること

が最小限求められるでしょう。現在のように，コンピュータの利用者が増大してくると，それだけ求められる機能にも幅が出てくるわけで，アプリケーションソフトウェアに対する要求も増してき

ます。数式処理ソフトウェアもその要求の一つとして機能を上げてきたものです。利用者の多くは技術に携わる人々です[5]。

技術者の仕事は千差万別で一概に分類できるものではないでしょうが，実験を重ねてその結果から一般的な法則を導き出す，というものを考えてみましょう。法則を出した後でも，検証のため再び同様の実験を行いたいことがあります。しかし，その度ごとにお金のかかる実験を繰り返すのは難しいこともあり，コンピュータによる模擬実験で済ませることも多いようです。この作業工程を簡単に示すと次のようになります。

(1) 実験 → (2) 理論構築 → (3) コンピュータ上で模擬実験 →
(4) 実験結果との比較・考察 → (5) 理論修正

コンピュータがない時代，(3) は手作業で可能な範囲でしかできませんでしたし，(1) → (2) のためには実験データを手書きでまとめ上げ，グラフを描いたりしなければなりません。統計的なデータでしたら，平均値・中央値・標準偏差・分散・ヒストグラムと全てが人力ですから手間は大変です。(1) において，計測装置の値を読み取るにも，人間の目で見，手で値を書き取らねばなりません。コンピュータの出現でこれらの作業が全てプログラム化され自動で行われるようになった訳ですが，隔靴掻痒の感は払拭されていません。

何故か? 理由は 2 つあります。

一つは，実験の形態やデータ形式が変更される度に，それを処理するプログラムを書き直さねばならなかったからです。(1) → (2) の過程のデータ整理には統計処理機能を備えた汎用的な表計算ソフトウェアの登場で自分でプログラムを組むということは少なりましたが，(3) のために作られたプログラムは以前のものが全く二次利用できないことも多いのです。

二つ目として，(1) から (5) に至る作業を一つのプログラムの中で完結させるのは難しかったという点が上げられます。特に (2) や (4), (5) のような実に人間くさい作業において，プログラミング言語のような無数の単機能の集合体を使おうとするとやりたい仕事に直接は関係のない繁雑な工程，しかもそれを経なければ肝心のことが出来ないということになりかねません。また，作り上げた代数方程式や微分方程式が正しいかどうか，実験で得られた数値をはじき出せるのか，そもそも実験環境をパラメータとして与えた時，解自体が一意に存在するのか? オブジェクト指向と考え方が導入され，他人の作成した様々なライブラリが利用できるようになった昨今でも，プログラミング言語だけでその検証を行うのは容易なことではありません。単機能のプログラムを多数使いこなすことで，一定のルーチンワークはこなせるでしょうが，そもそもルーチンワークになり難いのがこの過程なのですから，どれほど効果が上がるかは疑問です。

これら二点の隔靴掻痒感を払拭するために発展と遂げつつあるのが数式処理ソフトウェアなのです。現在の数式処理ソフトウェアは大まかにいって，次のような機能を持っているものが普通です。

- 二次元・三次元グラフ作成
- 代数計算
- 高精度計算可能な電卓
- プログラミング言語的な文法構造

[5] 数学者も利用するようになってはきましたが，利用者全体の割合を考えるとそれほど多くはないと思われます。

かなり便利に使えるようになりましたが，反面，単機能のプログラムに比べると処理速度の点で難があります。これは多機能であるが故の宿命のようなもので，必然的なものです。しかし，その便利さは多少の処理速度の低下を補って余りあるのです。

例として，教科書 BASIC で記述されたリサージュ曲線を描くプログラムを，数式処理ソフトウェアで実行してみましょう。本書で取り上げる MuPAD を使ってみます (図 2)。

```
m := input("m>> ");
n := input("n>> ");
plot2d([Mode=Curve, [sin(m*t), sin(n*t)], t=[0,2*PI], \
Grid=[ceil(2*PI/0.02*(m+n))]]);
```

図 2: 数式処理ソフトウェアでリサージュ曲線を描く

1・2 行目が行番号 140 にあたる処理で，残りの処理は全て 3 行目で行っています。3 行目がいかに沢山の処理を行っているかがご理解できるでしょう。実はもっと複雑な処理も併せて行っていますから，グラフが描かれるまで少々待たされるの致し方ないですね。しかし，プログラミング言語よりもずっと少ない記述で様々な仕事が出来ることはご理解いただけると思います。それはひとえに，プログラミング言語よりずっと豊富な，数学をこねくり回すに必要なライブラリを沢山抱え，試行錯誤にふさわしいインターフェースを備えているからなのです。

最初に述べたように，プログラミング言語はもともとアプリケーションソフトウェアを開発するために発明されました。数式処理ソフトウェアもアプリケーションソフトウェアの一種です。プログラミング言語という土台の上に築かれた，プログラミング言語の要素も兼ね備えた，統合数学ツールという位置づけになります。もともとこの二つは利用用途が違うもので，どちらがよい悪いという比較は意味がありません。処理速度だけを問題にしてプログラミング言語の優位性を説くのも，数学ツールとして機能の豊富さだけを取り上げてその優位性を持ち上げるのも間違っています。二種類の道具の長所を生かす使い方をするのが，頭脳を持った人間の正しいあり方ではないでしょうか。

著者としては，もやもやした漠然としたアイディアをこねくりまわす際には数式処理ソフトウェアを活用し，端から見れば遊んで見えるような気楽な使い方をおすすめします。この関数の概形を見たいからグラフを描いてみようとか，この範囲内に解があるか，とか，要は賢い電卓のような使い方です。慣れてくれば，プログラミング言語的なちょっと長めの処理をさせてみます。そして，どうも処理速度に不満が出てくるようになってきたら，プログラミング言語によるプログラムを試してみる，というのが一番と考えています。もっとも，このような使い方はあくまで著者が考えてるモデルケースにすぎないので，やはり，使い方は自分の頭で考えていただきたいのです。

それでは，次章から数式処理ソフトウェアの機能を見ていくことにしましょう。

目 次

第1章　数

この章では一番基本となる数の演算について見ていくことにします。操作としては簡単ですので，MuPAD に慣れる練習のつもりでお付き合い下さい。

1.1　整数・分数の計算

百聞は一見にしかずといいます。まず MuPAD を起動し[1],「3+5;」と打ち込んでみて下さい。以下のような結果が得られるはずです。

```
>> 3 + 5;
                                  8
```

画面上は入力した文字列の前に「>>」というものは表示されませんが，以下，入力した文字列が判別しやすいよう，これを行頭に付けていきます。

何を入力したか，つまり MuPAD に何をさせたかは表示を見れば明らかで,「3+5 を計算し，その結果を画面に出せ」ということです。もう少し細かく見ていくと

- 「3」と「5」は整数の 3 と 5 を表わしています[2]
- 「+」は「足し算をしろ」という命令です。通常，演算子と呼びます
- 「;」(セミコロン) は,「これで命令文終わり」という意味です。「:」(コロン) でも命令文の終わりを表わすことが出来ます。両者の違いは，計算結果を表示する (セミコロン) か，しない (コロン) かという点にあります。一行に複数の命令文を一気に書きたいときには，命令ごとにセミコロンかコロンを挿入して下さい

ということになります[3]。

整数の計算を指定する演算子には他にも次のようなものがあります。

---

[1] MuPAD そのものの入手方法，及びインストール方法については付録を参照して下さい。

[2] 何を当たり前のことを，と思われるでしょうが，コンピュータの画面に現れた文字の解釈はあくまでコンピュータがやることです。つまり,「3」という文字を 3 という整数として解釈するか，それとも単なる「3」という形をした文字として扱うかはプログラムの作り方に依存します。

[3] コンピュータ関係の習い事をしていると，テキストや講師がよく「おまじないだから何も考えずにこうしてください」という言い方をよくします。「おまじない」という言葉には「其の意味を理解するのは現段階では難しいだろうから，十分習熟してから改めてちゃんと説明します」というニュアンスがあります。しかし，論理回路と論理構造から成り立っているはずのコンピュータの世界で「おまじない」という言葉は不可解と言わざるを得ません。場合によっては「おまじない」の意味が講師にすらちゃんと分かっていないこともありますから，この言葉が出てきたら，講師の説明を遮ってでも「おなまじないってどういう意味ですか?」と積極的に質問して下さい。

```
>> 3+5; 3-5; 3*5; 3/5; 3 mod 5; 3^2;
```

```
                                                    8
                                                   -2
                                                   15
                                                  3/5
                                                    3
                                                    9
```

右から順に，加法・減法・乗法・除法・剰余演算・累乗です。このうち，除法の結果は分数になっていることがお分かりになりますでしょうか?　MuPAD では，特に指定しないと割り切れない整数の除法は分数の形で表わされます。

```
>> 4/2; 5/3;
```

```
                                                    2
                                                  5/3
```

商を求めたい場合は

```
>> trunc(3/5);
```

```
                                                    0
```

として下さい。ここで出てきた「trunc」を関数と呼びます。

MuPAD では分数の混じった計算も可能です。演算子は整数と同じものが使えます。

```
>> 1/3 + 5/6; 3/7 - 7/8; 4/3 * 15/7; (3/2) / (5/8); (7/4)^2;
```

```
                                                  7/6
                                                -25/56
                                                 20/7
                                                 12/5
                                                49/16
```

また，長い数，多倍長計算も楽にこなします。コンピュータのメモリの許す限りの桁が取れますから，通常の使い方で不自由することは少ないでしょう。

```
>> -15901395728943 * 459280589293;

                                    -7303202400970134336007299

>> 5 - 9289567 + 29762897458923523489578 / 458972480 * (4590824 / 87989) + 98475125608;

                              85400125287768671319619573337/2524033096420
```

以下の `binomial(m, n)` は ${}_mC_n$ を意味します。

```
>> binomial(3,1); binomial(3,2); binomial(3,3);

                                    3

                                    3

                                    1

>> binomial(100,10);

                              17310309456440
```

また，以下の `fact(n)` は $n!$ を表わします。

```
>> fact(10);

                                 3628800

>> fact(100);

933262154439441526816992388562667004907159682643816214685929638952175999993\
229915608941463976156518286253697920827223758251185210916864000000000000000\
0000000000
```

素因数分解の機能も充実しています。以下では 144 を素因数分解しています。

```
>> ifactor(144);

                                          [1, 2, 4, 3, 2]
```

出力結果の意味は $144 = 1 \times 2^4 \times 3^2$ ということです。もうちょっと分かりやすく結果を表示して欲しいものですね。

素因数分解が出来ますので，最大公約数 (GCD)，最小公倍数 (LCM) の計算も楽にこなせます。最初は 12，24，246 の最大公約数を，次は最小公倍数を求めています。

```
>> igcd(12, 24, 256);

                                                    4

>> ilcm(12, 24, 256);

                                                  768
```

指定された数が素数かどうかを判断する関数もあります。以下では 89797，37，867569 が素数かどうかを判断しています。

```
>> isprime(89797); isprime(37); isprime(867569);


                                                 TRUE

                                                 TRUE

                                                FALSE
```

ご覧のように，89797 と 37 が素数 (TRUE)，867569 は素数ではない (FALSE) と判断しました。MuPAD は大学の研究者が開発に携わっているせいか，ちょっと高度な関数も満載しています。

`numlib::lincongruence(a, b, m)` という関数は，線型合同式の計算を行ってくれる関数です。線型合同式とは ax を m で割った余りが b になるような x を求める方程式です。関数の前についている「`numlib::`」はドメインを表わしています。ドメインとは，特定用途の関数をひとまとめにした時に使うグループ名です。

```
>> numlib::lincongruence(13, 9, 27);

                                                 [9]

>> numlib::lincongruence(37, 82, 493);

                                               [322]
```

練習問題 1.1

最後の線型合同式が正しいかどうか，MuPAD の機能を使って確認して下さい。

1.2 無理数と小数

以下のように打ち込んで下さい。

```
>> reset();
```

この reset 関数は，今まで MuPAD に打ち込んできた履歴をきれいさっぱり洗い流してくれます。仕事が一段落した時に実行すると妙な動作に悩まされずに済むでしょう。

この節では無理数と小数を扱います。無理数の例として平方根を使ってみましょう。平方根を表わす方法は２つあります。一つは sqrt 関数を使う方法，もう一つはベキ乗を使う方法です。

```
>> sqrt(2);
```

$$2^{1/2}$$

```
>> 2^(1/2);
```

$$2^{1/2}$$

どちらも結果は同じです。計算も通常通りやってくれます。但し，特に指定しない限り，数をまとめるという作業はやってくれません。

```
>> 5^(1/2) * 6^(1/6); sqrt(5) * sqrt(6);
```

$$5^{1/2}\,6^{1/6}$$

$$5^{1/2}\,6^{1/2}$$

数をまとめたい場合は次のように simplify 関数を使います。この場合は前の結果 (%で表わします) を簡素化しています。

```
>> simplify(%);
```

$$30^{1/2}$$

逆に，大きい数は素因数分解して表示してくれます。

```
>> sqrt(1746451354687654134 7867);
```

$$\text{1/2}$$

```
                        7 356418643813806966283
```

```
>> sqrt(56389521000);
```

```
                                          1/2
                              10 563895210
```

```
>> sqrt(256*4530*10230);
```

```
                                        1/2
                              480 51491
```

```
>> sqrt(26356846465^2 * 6);
```

```
                                          1/2
                            26356846465 6
```

有理化も自動的にやってくれています。

```
>> 8 / sqrt(56);
```

```
                                     1/2
                                  2 14
                                  -------
                                     7
```

よく使う無理数として，円周率の π や自然対数の底 e があります。これらは MuPAD で `PI` と `E` として定義されています。これを有限桁の小数で表示するには `float` 関数を使います。

```
>> float(PI); float(E);
```

```
                              3.141592653
```

```
                              2.718281828
```

桁数をもっと増やしたいときには以下のように `DIGITS` というグローバル変数[4]を変更してやります。以下の例では π と e を 100 桁表示しています。

```
>> DIGITS := 100; float(PI); float(E);
```

```
                                   100
```

[4] MuPAD 実行中，常に有効な変数のこと。

```
>> 3.14159265358979323846264338327950288419716939937510582097494459230781640628\
62089986280348253421170 68
```

```
>> 2.71828182845904523536028747135266249775724709369995957496696762772407663035\
354759457138217852516642 7
```

練習問題 1.2
$\sqrt{2}$ を 100 桁求めて下さい。

1.3 複素数

reset 関数を使って，真っ新にしておきましょう。

```
>> reset();
```

MuPAD は複素数も扱えます。虚数単位 $\sqrt{-1}$ は I で表わします。

```
>> I; sqrt(-1);
```

$$I$$

$$I$$

試しに，解が複素数になる二次方程式を解いてみましょう。ここで使う solve 関数はかなり賢い機能を持っています。詳細は次の章で説明します。

```
>> solve(3 * x^2 + x + 7, x);
```

$$\{-\frac{1}{6} - \frac{I\sqrt{83}}{6}, -\frac{1}{6} + \frac{I\sqrt{83}}{6}\}$$

複素数も実数と同じ計算が可能です。実数よりも数字が長いので，いちいち打ち込むのも面倒ですから，a と b という変数にあらかじめ計算したい複素数を入れておきます。

```
>> a := 3 - 4*I; b := 8 + 9 * I;
```

$$3 - 4\,I$$

$$8 + 9\,I$$

では四則演算をさせてみましょう。

```
>> a + b; a - b; a * b; a / b;
```

```
                                  11 + 5 I

                                 - 5 - 13 I

                                  60 - 5 I

                              - 12/145 - 59/145 I
```

複素数の実部を返す関数 Re と虚部を返す関数 Im もあります。

```
>> Re(a); Im(a);
```

```
                                     3

                                    -4
```

conjugate 関数は共役複素数を返します。

```
>> conjugate(a);
```

```
                                  3 + 4 I
```

練習問題 1.3
上で求めた二次方程式の解が方程式を満足していることを確認して下さい。

1.4　集合

MuPAD の特徴の一つに，集合演算の機能があります。ここでは簡単な例でその機能を見ていきます。

まず，100 までの自然数のうち，偶数からなる集合 A を定義してみます。

```
>> A := {i * 2 $ i=1..trunc(100/2)};
```

```
{62, 16, 86, 40, 64, 18, 88, 42, 66, 20, 90, 44, 68, 22, 92, 46, 70, 24, 94,
48, 2, 72, 26, 96, 50, 4, 74, 28, 98, 52, 6, 76, 30, 100, 54, 8, 78, 32, 56,
10, 80, 34, 58, 12, 82, 36, 60, 14, 84, 38}
```

次に，3 の倍数の集合 B を定義します。

```
>> B := {i * 3 $ i=1..trunc(100/3)};
```

```
{39, 63, 87, 18, 42, 66, 90, 21, 45, 69, 93, 24, 48, 72, 3, 96, 27, 51, 75, 6,
99, 30, 54, 78, 9, 33, 57, 81, 12, 36, 60, 84, 15}
```

同様にして，5 の倍数の集合 C を定義します。

```
>> C := {i * 5 $ i=1..trunc(100/5)};
```

```
{85, 40, 65, 20, 90, 45, 70, 25, 95, 50, 5, 75, 30, 100, 55, 10, 80, 35, 60, 15}
```

ではまずこの 3 つの集合の要素数を求めてみましょう。nops 関数を使います。

```
>> nops(A); nops(B); nops(C);
```

```
                                    50

                                    33

                                    20
```

X を A, B, C の和集合として定義し，その要素数を求めてみます。和集合を求めるには union という命令を使います。

```
>> X := A union B union C; nops(X);
```

```
{39, 85, 62, 16, 63, 86, 40, 87, 64, 18, 65, 88, 42, 66, 20, 21, 90, 44, 45,
68, 22, 69, 92, 46, 93, 70, 24, 25, 94, 48, 2, 3, 95, 72, 26, 27, 96, 50, 4,
51, 5, 74, 28, 75, 98, 52, 6, 99, 76, 30, 100, 54, 8, 9, 55, 78, 32, 33, 56,
10, 57, 80, 34, 81, 35, 58, 12, 82, 36, 60, 14, 15, 84, 38}
```

```
                                    74
```

Y として，A, B, C の積集合を定義し，その要素集を求めてみます。積集合を求めるには intersect という命令を使います。この場合，Y は $2, 3, 5$ の最小公倍数の倍数からなる集合になりますね。

```
>> Y := A intersect B intersect C; nops(Y);
```

```
                                {90, 30, 60}

                                    3
```

集合からその部分集合を取り除く演算もあります。以下の例では X から Y の要素を除いて，その要素数を求めています。

```
>> Z := X minus Y; nops(Z);
```

```
{39, 85, 62, 16, 63, 86, 40, 87, 64, 18, 65, 88, 42, 66, 20, 21, 44, 45, 68,
22, 69, 92, 46, 93, 70, 24, 25, 94, 48, 2, 3, 95, 72, 26, 27, 96, 50, 4, 51,
5, 74, 28, 75, 98, 52, 6, 99, 76, 100, 54, 8, 9, 55, 78, 32, 33, 56, 10, 57,
80, 34, 81, 35, 58, 12, 82, 36, 14, 15, 84, 38}
```

```
71
```

補集合を定義してみましょう。準備として 100 までの全ての自然数からなる全体集合 G を求めます。

```
>> G := {i $ i=1..100};

{85, 62, 39, 16, 86, 63, 40, 17, 87, 64, 41, 18, 88, 65, 42, 19, 89, 66, 43,
20, 90, 67, 44, 21, 91, 68, 45, 22, 92, 69, 46, 23, 93, 70, 47, 24, 1, 94, 71,
 48, 25, 2, 95, 72, 49, 26, 3, 96, 73, 50, 27, 4, 97, 74, 51, 28, 5, 98, 75,
 52, 29, 6, 99, 76, 53, 30, 7, 100, 77, 54, 31, 8, 78, 55, 32, 9, 79, 56, 33,
 10, 80, 57, 34, 11, 81, 58, 35, 12, 82, 59, 36, 13, 83, 60, 37, 14, 84, 61,
 38, 15}
```

では補集合を求めてみましょう。

```
>> nA := G minus A; nB := G minus B; nC := G minus C;

{85, 39, 63, 17, 87, 41, 65, 19, 89, 43, 67, 21, 91, 45, 69, 23, 93, 47, 1,
71, 25, 95, 49, 3, 73, 27, 97, 51, 5, 75, 29, 99, 53, 7, 77, 31, 55, 9, 79,
33, 57, 11, 81, 35, 59, 13, 83, 37, 61, 15}

{85, 62, 16, 86, 40, 17, 64, 41, 88, 65, 19, 89, 43, 20, 67, 44, 91, 68, 22,
92, 46, 23, 70, 47, 1, 94, 71, 25, 2, 95, 49, 26, 73, 50, 4, 97, 74, 28, 5,
98, 52, 29, 76, 53, 7, 100, 77, 31, 8, 55, 32, 79, 56, 10, 80, 34, 11, 58, 35,
82, 59, 13, 83, 37, 14, 61, 38}

{62, 39, 16, 86, 63, 17, 87, 64, 41, 18, 88, 42, 19, 89, 66, 43, 67, 44, 21,
91, 68, 22, 92, 69, 46, 23, 93, 47, 24, 1, 94, 71, 48, 2, 72, 49, 26, 3, 96,
73, 27, 4, 97, 74, 51, 28, 98, 52, 29, 6, 99, 76, 53, 7, 77, 54, 31, 8, 78,
32, 9, 79, 56, 33, 57, 34, 11, 81, 58, 12, 82, 59, 36, 13, 83, 37, 14, 84, 61,
 38}
```

次の例は何を計算しているでしょう?　ちょっと考えてみて下さい。

```
>> (G minus (nA intersect nB)) minus (A union B);

                                                {}
```

練習問題 1.4
$\overline{A \cup B} = \overline{A} \cap \overline{B}$ であることを MuPAD で確認して下さい。同様に $\overline{A \cap B} = \overline{A} \cup \overline{B}$ も確認してみて下さい。

第2章　多項式・方程式

この節では多項式の計算，方程式の扱いについて見ていきます。多項式計算は数式処理の一番の要とも言うべき部分です。

2.1　多項式の計算

とりあえず，今までの履歴は消去しておきましょう。

```
>> reset();
```

まず，一次式どうしの乗算を実行してみましょう。単に式をそのまま入力するだけでは展開してくれませんので，展開させたいときには expand 関数を使います。

```
>> (x+2) * (2*x-3);
```

```
                          (x + 2) (2 x - 3)
```

```
>> expand(%);
```

```
                                  2
                            x + 2 x  - 6
```

除算も同様で，約した結果が必要であれば simplify 関数を使います。

```
>> % / (x+2);
```

```
                                  2
                            x + 2 x  - 6
                            ------------
                               x + 2
```

```
>> simplify(%);
```

```
                               2 x - 3
```

因数分解もできます。というか，これが出来なければ他への応用も難しいでしょう。

因数分解には 2 つの関数が用意されています。一つは factor 関数，もう一つは Factor 関数です。前者は因数分解した要素をリストとして返しますが，後者は因数分解された多項式として結果を返してくれます。

```
>> factor(x^3+3*x^2+3*x+1);

                              [1, x + 1, 3]

>> Factor(x^3+3*x^2+3*x+1);

                                      3
                                (x + 1)
```

変数が増えても上の計算は同様に可能です。正確には，どの文字が変数であるかを指定しておくと良いのですが，以下の計算には支障がないのでやっていません。

```
>> p := (3*x+y^2+z^3) * (-x^2+4*y-2) * (x + y + z);

                                  2    3          2
                (x + y + z) (3 x + y  + z ) (4 y - x  - 2)

>> expand(p);

                            2      4      3      4      4         2
12 x y z - 6 x y - 6 x z - 6 x  - 3 x  - 2 y  + 4 y  - 2 z  + 10 x y  +

      2          3      3          3      3        2          3      3
   12 x  y + 4 x y  - 3 x  y - 2 x z  - 3 x  z - 2 y  z - 2 y z  + 4 y  z +

      4          3    2  3    3  2    2  4    3  3      2  3    2  2
   4 y z  + 4 x y z  - x  y  - x  y  - x  z  - x  z  + 4 y  z  - x  y  z -

    2    3
   x  y z

>> Factor(%);

                            2                    2    3
              - (x + y + z) (x  - 4 y + 2) (3 x + y  + z )
```

最後に divide 関数を使ってみましょう。まず，モトになる多項式 q を定義し，同時に展開しておきます。

```
>> q := expand((x+y)*(y+z)*(a+b));
```

$$a\,x\,y + a\,x\,z + b\,x\,y + a\,y\,z + b\,x\,z + b\,y\,z + a\,y^2 + b\,y^2$$

divide 関数は 第一引数/第二引数 の結果を返す関数です。この関数の第 4 引数が Quo であれば商を，Rem であれば剰余を返します。第 3 引数では変数の指定を行っています。

```
>> divide(q, x+y, [x], Quo);
```

$$a\,y + a\,z + b\,y + b\,z$$

```
>> divide(q, x+y, [x], Rem);
```

$$a\,y\,z + b\,y\,z + a\,y^2 + b\,y^2 + y\,(-a\,y - a\,z - b\,y - b\,z)$$

```
>> qq := divide(q, x+y, [x], Quo):
   qr := divide(q, x+y, [x], Rem):
   Factor(qq * (x+y)+qr);
```

$$(a + b)\,(x + y)\,(y + z)$$

練習問題 2.1
$(x + y + z + u + v + w)^5$ の展開式を求めて下さい。

2.2 方程式の計算

お約束になりました。一区切り付いたら reset 関数を使う癖をつけておくと何かと便利です。

```
>> reset();
```

ここでは方程式，不等式を解くための solve 関数を使ってみます。かなり守備範囲の広い関数ですから，一次方程式から三次方程式までずらっと試してみます。使い方の基本は

```
solve(方程式, 変数);
```  (2.1)

です。

```
>> solve(3*x+2=1, x);
```

```
{-1/3}
```

```
>> solve(x^2-3*x+2=0, x);

                                  {1, 2}

>> solve(x^3+3*x^2+3*x+1=0, x);

                                  {-1}
```

連立方程式も解くことが出来ます。方程式をまとめて {} で括ってやり，変数も同様に {} で括って複数指定します。

```
>> solve({x+y=1, 2*x-y=2}, {x, y});

                             {{x = 1, y = 0}}
```

方程式の解を小数で求めたいときには，float 関数を使います。

```
>> solve(3*x^2-3*x-5=0, x);

                      {          1/2    1/2       }
                      {       69     69           }
                      { 1/2 - -----, ----- + 1/2 }
                      {          6      6         }

>> float(%);

                       {1.88443731, -0.8844373105}
```

桁数を増やしたいときや，予め小数としての解だけが必要なときには，今までやった方法を次のように組み合わせてやります。以下の例は 50 桁求めています[1]。

```
>> DIGITS := 50; float(solve(4*x^4-3*x + 2=0, x));

                                    50

{

0.62009832451987948927958388996275595590316905058 95 - 0.286619136992013323\
67611609121052135333257838566015 I
```

[1] 小数の場合，表示された全ての桁が正しいとは限りません。末尾の桁に行くほど信頼度は下がっていくことを常識として知っておきましょう。

```
   ,

0.62009832451987948927958388996275595590316905058 95 + 0.286619136992013323\
67611609121052135333257838566015 I


   ,

- 0.62009832451987948927958388996275595590316905058 95 - 0.8287902837598164\
38660543626615964771007490590496 66 I


   ,

- 0.62009832451987948927958388996275595590316905058 95 + 0.8287902837598164\
38660543626615964771007490590496 66 I


   }
```

solve 関数は不等式も解くことが出来ます。以下で infinity とは無限大のことです。

```
>> solve(3*x+1 <= 0);

                        {[-infinity <= x, x <= -1/3]}

>> solve({x^2+2*x-8 <= 0, 3*x+1 <= 0});

                            {[-4 <= x, x <= -1/3]}
```

以下の不等式は線型計画法と呼ばれている問題です。まず問題となる不等式を p という変数に定義し，それが線型計画法としての解を持つかどうかを linopt::feasible 関数でチェックしています。

```
>> p := {2*x+2*y <= 30, 2*x+3*y <= 40, 4*x+2*y <= 48, x >= 0, y >= 0};

    {0 <= x, 2 x + 3 y <= 40, 0 <= y, 4 x + 2 y <= 48, 2 x + 2 y <= 30}

>> linopt::feasible(p);

                                    TRUE
```

線型計画法とは，p のような制約条件下で一次式を最大・最小にする一意的に決まる解を求める問題です。例えば，上の制約下で 1 次式 $10x + 8y$ を最大化する解は以下のようにして求められます。

```
>> linopt::maximize(10*x+8*y, p);
```

```
{y = 6, x = 9}
```

同様に，最小化する問題も解くことが出来ます。

```
>> q := {2*u+2*v +4*w >= 10, 2*u+3*v+2*w >= 8, u >= 0, v >= 0, w >= 0};
```

```
{8 <= 2 u + 3 v + 2 w, 10 <= 2 u + 2 v + 4 w, 0 <= u, 0 <= v, 0 <= w}
```

```
>> linopt::minimize(30*u+40*v+48*w, q);
```

```
{v = 0, u = 3, w = 1}
```

上の 2 つの線型計画法の問題は，双対問題と呼ばれている関係にあります。

練習問題 2.2
次の方程式不等式を解いて下さい。

1. $(x+3)^3(2x-3)^2 = (x-4)^2$
2. $(4x-2)^2(x-\frac{1}{3})^2 > x$

2.3　数列

お約束です。

```
>> reset();
```

solve 関数の応用として，数列の一般項を求めてみましょう。rec 関数は，引数として与えられた式が再帰的 (recursive) であることを表わすための関数です。つまり

```
rec(漸化式，数列の項，初期値)
```　(2.2)

というように使います。最後の初期値は必要がなければ省略することもできます。こうして数列を定義してやると，これは solve 関数で一般項を導くことが出来るようになります。

```
>> solve(rec(a(n+1)=a(n)+3, a(n), a(0)=1));
```

```
{3 n + 1}
```

もう少し複雑な例として，有名な Fibonacci 数列の一般項を求めてみます。

```
>> solve(rec(f(n+2)=f(n+1)+f(n), f(n), {f(0)=1, f(1)=1}));
```

```
   { /        1/2 \ /        1/2 \n   /  1/2       \ /  1/2       \n }
   { |       5    | |       5    |    | 5          | | 5          |  }
   { | 1/2 - ---- | | 1/2 - ---- |  + | ---- + 1/2 | | ---- + 1/2 |  }
   { \        10  / \        2   /    \  10        / \  2         /  }
```

初期値を指定しないと 0 番目の項を初期値の代わりに使用します。

```
>> solve(rec(b(n+1)=b(n)*4, b(n)));
```

```
                                  n
                               {b(0) 4 }
```

一般項を囲っている {} を外すには op 関数を使います。

```
>> op(%);
```

```
                                  n
                               b(0) 4
```

今までやったことを組み合わせて，漸化式から数列の一般項を求め，更にその和も求めてみます。

```
>> a := op(solve(rec(a(n+1)=a(n)+3, a(n), a(0)=1)));
```

```
                                3 n + 1
```

```
>> sum(a, n=1..m);
```

```
                                        2
                                  5 m   3 m
                                  --- + ----
                                   2     2
```

最後に，数学的帰納法の証明で行う計算を MuPAD にやらせてみましょう。何をやっているか，わかりますか？

```
>> s := n * (n+1)/2;
   sk1 := subs(s, n=k-1);
   sk := subs(s, n=k);
   simplify(sk - sk1);
```

```
                                n (n + 1)
                                ---------
```

```
    2

k (k - 1)
---------
    2

k (k + 1)
---------
    2

    k
```

練習問題 2.3
等比数列 $a_n := a_0 r^n$ の和 $\sum_{i=0}^{n} a_i$ を求めて下さい。

第3章　関数とグラフ

前にも述べましたが，最近の数式処理ソフトウェアで，グラフが描けないものは殆ど存在しません。これは，パソコンの処理能力が上がり，少々複雑なグラフでも短時間で描けるようになったこと，そして，グラフと数式との対応が想像できるようになることが数学の理解には欠かせない訓練であることが認識されてきたからです。本章では 2 次元（平面）のグラフを描くための関数とその使い方を見ていくことにします。

3.1　グラフを描く

前にも述べましたが，パソコン[1]上で動作する数式処理ソフトウェアで，グラフを描けないものは殆ど存在しません。これはパソコンの処理能力が上がり，少々複雑な図形でも短時間で描画できるようになったこと，そして数学には数式と図形との対応付けが必要不可欠であること[2]が認識されてきたからでしょう。

ここではまず多項式関数と分数関数[3]のグラフを描いてみます。

ここで扱う関数は

$$y = f(x) \tag{3.1}$$

という形のものに限ります。このうち (3.1) の左辺の関数が

$$a_n x^n + a_{n-1} x^{n-1} + \cdots a_1 x + a_0 \tag{3.2}$$

という形をしているものを多項式関数と呼びます。ここで，a_n, ... a_0 は全て実数です。皆さんが今までに取り扱ってきた関数の大多数は $n = 1, 2$ の場合だけだったわけです。まず，この 2 つの関数を肴に，MuPAD のグラフ機能を使ってみることにします。

(3.1) のような 2 次元の関数グラフを，x の範囲を $[\alpha, \beta]$ として描くには

$$\texttt{plotfunc(}f(x)\texttt{, x=}\alpha\texttt{..}\beta\texttt{);}$$

と指定します。例えば

$$y = 3x + 5 \tag{3.3}$$

のグラフを，$[-5, 5]$ の範囲で描こうとすると

```
>> plotfunc(3*x+5, x=-5..5);
```

と指定します (図 3.1)。同様に二次関数 $y = 3x^2 + 7x + 4$ のグラフを $[-10, 10]$ の範囲で描くには

[1] いわゆる PC，Macintosh のこと。
[2] 分野にもよる。
[3] 両者まとめて有理型関数とも言う。

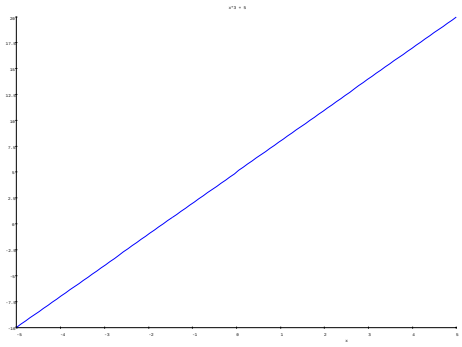

図 3.1: plotfunc(3*x+5, x=-5..5)

```
>> plotfunc(3*x^2+7*x+4, x=-10..10);
```

とすればいいわけです (図 3.2。この放物線の x 座標は

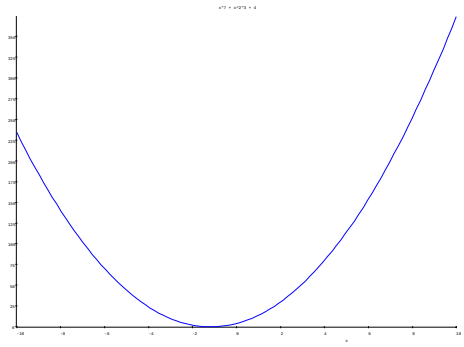

図 3.2: plotfunc(3*x^2+7*x+4, x=-10..10)

```
>> solve(3*x^2+7*x+4 = 0);
                                                    {-1, -4/3}
>> x := (-1+(-4/3))/2;
                                                        -7/6
>> subs(3*x^2+7*x+4, x = -7/6);
                                                       -1/12
```

ですから，$(-\frac{7}{6}, -\frac{1}{12})$ となります。

二つ以上の関数と一つの座標にまとめて描くこともできます。先の直線と放物線のグラフを重ねてみましょう (図 3.3)。

```
>> unasigned(x);
   plotfunc(3*x+5, 3*x^2+7*x+4, x=-5..5);
```

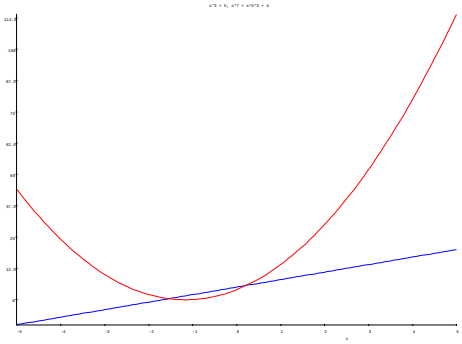

図 3.3: `plotfunc(3*x+5, 3*x^2+7*x+4, x=-5..5)`

よく出題される問題として,「二つの関数 $y = f(x)$ と $y = g(x)$ との交点を求めよ」というものがあります。MuPAD のグラフ作成機能だけを利用して，これを解いてみましょう。関数としては先ほどの $f(x) = 3x + 5$ と $g(x) = 3x^2 + 7x + 4)$ を使います。

図 3.3 より，交点は二つあることがわかります。一つは $[-2, -1]$ に，もう一つは $[0, 1]$ の範囲にありますね。もっと細かく見るには，それぞれの区間を拡大してグラフを描き直してやればよいわけです。実際，$[-2, -1]$ の方を拡大すると，交点の x 座標値は $-1.5\ldots$ であることが読み取れます (図 3.4 の左)。y 座標の値は 0.25 から 0.5 の範囲にあることしか分かりませんので，もう一段拡大します (図 3.4 の右)。ここまでくると $y = 0.3\ldots$ であることがはっきりします。この調子でずんずん拡大していけば，必要な桁数だけ交点の座標値が正確に，しかも計算せずに求めることが出来ます。

```
>> plotfunc(3*x+5, 3*x^2+7*x+4, x=-2..-1);
>> plotfunc(3*x+5, 3*x^2+7*x+4, x=-1.6..-1.5);
```

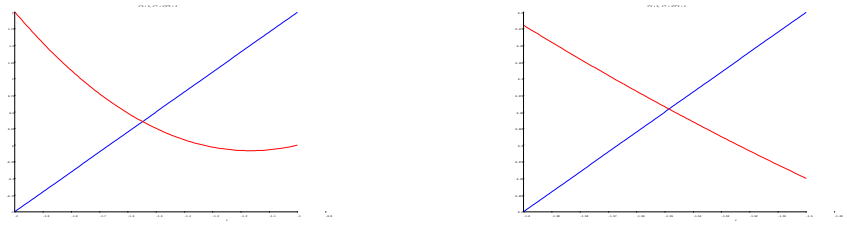

図 3.4: `plotfunc(3*x+5, 3*x^2+7*x+4,...`

勿論，`solve` 関数を使えばより正確な交点の座標値が求まるわけですが，このようなグラフを使ったアプローチは，解の存在する範囲を対極的に俯瞰できることが出来る点，非常に優れた方法

の一つです。数式の計数や指数にミスがあった場合，グラフの形から「どうも変だ」という直感が働くこともあるからです。

分数関数も，不連続点のあるなしにかかわらず，きちんとグラフを描くことが出来ます。例えば

$$y = \frac{4x+3}{2x-5} \tag{3.4}$$

のグラフを描いてみましょう。

```
>> plotfunc((4*x+3)/(2*x-5), x=-10..10);
```

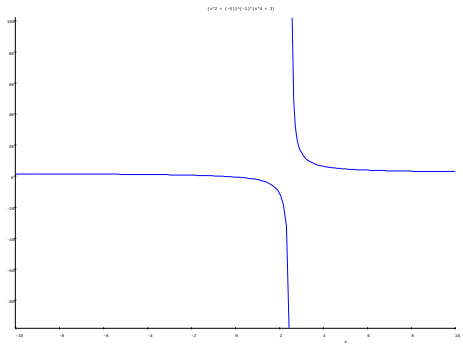

図 3.5: `plotfunc((4*x+3)/(2*x-5), x=-10..10)`

ご覧のように $x = \frac{5}{2}$ で不連続点を持ちますが，MuPAD はうまくかわして描いています。

練習問題 3.1

$y = \frac{4x+3}{2x-5}$ と $y = 3x + 5$ の交点を，solve 関数と使った方法とここで紹介したグラフを描く方法とでそれぞれ求めて下さい。

3.2　三角関数

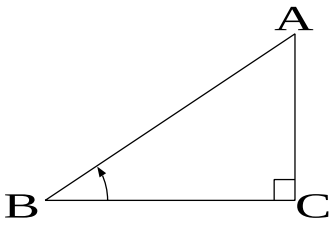

図 3.6: 直角三角形

図 3.6 において，$\angle ABC = \theta$ によって変動する次の関数群を三角関数と呼びます。

$$\sin\theta = \frac{CA}{AB}$$

$$\begin{aligned}\cos\theta &= \frac{BC}{AB}\\ \tan\theta &= \frac{CA}{BC} = \frac{\sin\theta}{\cos\theta}\end{aligned}$$

中学校で習ったピタゴラスの定理から，次のことが分かります。

$$\sin^2\theta + \cos^2\theta = 1$$

MuPAD でこれを確認します。

```
>> simplify(sin(x)^2+cos(x)^2);
```

```
                                  1
```

昔は次の関数も三角関数として教えられていました。

$$\begin{aligned}\cot\theta &= \frac{1}{\tan\theta}\\ \sec\theta &= \frac{1}{\cos\theta}\\ \mathrm{cosec}\,\theta &= \frac{1}{\sin\theta}\end{aligned}$$

これらも MuPAD で使用することは可能です。が，別段なくても sin, cos, tan で代用できますから，以降はこの三つだけ使うことにします。

MuPAD に限らず，三角関数の使える関数電卓なのでも角度の単位は「度」だけではなく，むしろ「ラジアン (radian)」という単位を使うことが普通です。これは 360 度を 2π とするものです。よく使われる角度とラジアンとの対応を書くと次のようになります。

度	→	ラジアン
$0\ ^\circ$	→	0
$30\ ^\circ$	→	$\frac{\pi}{6}$
$45\ ^\circ$	→	$\frac{\pi}{4}$
$60\ ^\circ$	→	$\frac{\pi}{3}$
$90\ ^\circ$	→	$\frac{\pi}{2}$
$180\ ^\circ$	→	π
$360\ ^\circ$	→	2π

MuPAD の三角関数は全てラジアンを使わなければなりません。従って，三角関数のグラフを $[0, 360\ ^\circ]$ の範囲で描くには $[0, 2\pi]$ としなければなりません (図 3.7)。

```
>> plotfunc(sin(x), cos(x), tan(x), x=0..2*PI, YRange=-2..2);
```

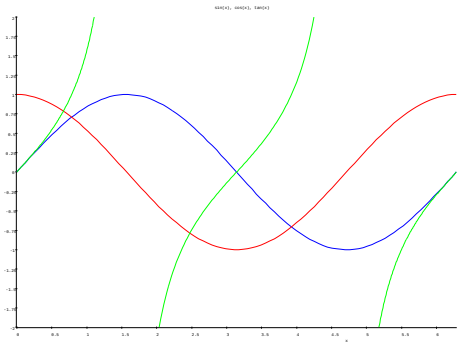

図 3.7: `plotfunc(sin(x), cos(x), tan(x), x=0..2*PI, YRange=-2..2)`

ここで `YRange=-2..2` とは，グラフに描く y 軸の範囲を $[-2, 2]$ に限定するために使っています。その理由は，tan 関数は $x = \frac{(2n+1)\pi}{2}$ で発散するからです。それに引きずられて y 軸の範囲が大幅に広くなり，相対的に sin, cos 関数のグラフが小さくなってしまうのです。

グラフから sin，cos 関数がきれいな波形をしており，$\sin\left(x + \frac{\pi}{2}\right) = \cos x$ であることも確認できます。この波形を正弦波と呼びます。

自然界に現れる波形は，一見複雑な形状をしていますが，細かく見ていくとこの正弦波を縦方向に伸縮させたり，周期を増減させたりしたものを加え合わせたもので表わすことが出来ます。具体的に言いますと，$b_n \sin nx$ として，b_n →大 (小) ならば波形が縦方向に延び (縮み)，n →大 (小) ならば周期が増え (減り) ます。実用上は cos 関数も取り入れた

$$\frac{a_0}{2} + \sum^{\infty} (a_n \cos nx + b_n \sin nx) \tag{3.5}$$

という無限級数を使って，変動に一定の周期を持つ関数にいくらでも近づけることが出来るようになります。これを Fourier 級数と呼びます。このあたりを詳しく説明すると量が増えますので，次のような例でその感覚だけつかんで下さい。

$$f(x) = \begin{cases} 1 & (0 < x < \pi) \\ 0 & (x = 0, \pi, 2\pi) \\ -1 & (\pi < x < 2\pi) \end{cases} \tag{3.6}$$

という関数 $f(x)$ を考えます。これに対する Fourier 級数は

$$\frac{4}{\pi}\left(\sin x + \frac{\sin 3x}{3} + \frac{\sin 5x}{5} + \frac{\sin 7x}{7} + \cdots\right) \tag{3.7}$$

となります。この級数の n 次項まで計算する関数を MuPAD で定義してみましょう。

```
>> ff := proc(n, x)
local ans;
begin
ans := 0;
```

```
for i from 0 to n do
ans := ans + sin((2*i+1)*x) / (2*i+1);
end_for;
ans * (4/PI);
end_proc:
```

これを使って項数が増えるに従い，$f(x)$ のグラフに近づいていくことを確認しましょう (図 3.8)。

```
>> plotfunc(ff(0, x), ff(1, x), ff(2, x), ff(3, x), x=0..2*PI);
```

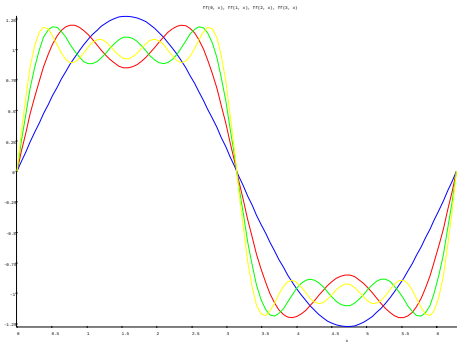

図 3.8: `plotfunc(ff(0, x), ff(1, x), ff(2, x), ff(3, x), x=0..2*PI)`

練習問題 3.2
$f(x) = |x|(-\pi \leq x \leq \pi)$ の時，この関数 $f(x)$ の Fourier 展開は $\frac{\pi}{2} - \frac{4}{\pi}\sum_{n=1}^{\infty}\frac{\cos(2n-1)x}{(2n-1)^2}$ です。この級数を n 項まで計算する関数 fg を作り，$n = 1, 2, 3$ の時のグラフをそれぞれ描いてみて下さい。

3.3 指数関数・対数関数

指数関数とは，ある実定数 a に対して

$$y = a^x \tag{3.8}$$

という形の関数の総称です。すぐに分かることですが，$a > 1$ ならば右上がりのグラフに，$0 < a < 1$ ならば右下がりのグラフになります。また，任意の a に対して，y 軸との交点が 1 であることも明らかです。

ではまず $a = 3, 2, 1, \frac{1}{2}, \frac{1}{3}$ と変えてグラフを描いてみましょう。

```
>> plotfunc(3^x, 2^x, 1^x, (1/2)^x, (1/3)^x, x=-2..2);
```

$a = e = 2.71828\ldots$ の場合は，MuPAD では特別な関数 `exp(x)` として定義されています。高校の教科書では通常の指数関数と同様，e^x と書いているものです。この e は

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e$$

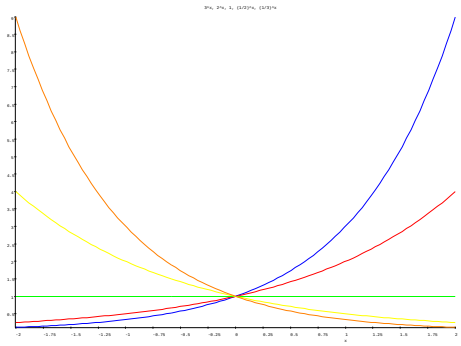

図 3.9: `plotfunc(3^x, 2^x, 1^x, (1/2)^x, (1/3)^x, x=-2..2)`

という極限値になっています。

e^x を $y = x$ の直線を軸として対称なグラフを描く関数が e を底とする対数関数，$\log x$ です。MuPAD では `ln(x)` と定義されています。実際にグラフを描くとよく分かるでしょう。

```
>> plotfunc(x, exp(x), ln(x), x=0..2);
```

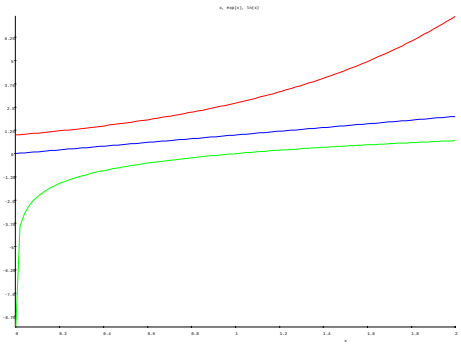

図 3.10: `plotfunc(x, exp(x), ln(x), x=0..2)`

任意の正数 a を底とする一般の対数関数 $\log_a x$ は MuPAD では定義されていません。しかし，ないと困るので自分で作ってみましょう。MuPAD にあるのは e を底とする対数関数しかないので，底の変換公式

$$\log_a b = \frac{\log b}{\log a} \tag{3.9}$$

を利用します。

```
>> log := proc(a, b)
local ans;
begin
ans := ln(b) / ln(a);
ans;
```

```
end_proc:

>> float(log(10,2)), float(log(10, 100));

                                        0.3010299956, 2.0
```

練習問題 3.3
次の方程式を解いてみましょう[4]。

- $4^x - 5 = 5$
- $(\log_3 x)^2 = 3\log_3 x - 2$

3.4 媒介変数を使った関数

私たちが二次元のグラフを描くときに使う $X-Y$ 座標では，一つの点 P を座標値 (x, y) という二つの実数でその場所を特定します。この x 座標，y 座標の値がそれぞれ

$$\begin{aligned} x &= f(t) \\ y &= g(t) \end{aligned}$$

というように，共通の一変数を持つ関数でそれぞれ定義されるとき，t を媒介変数，そしてこのような表現を媒介変数表示と呼びます。媒介変数表示の利点は，必ずしも $y = f(x)$ というように，片方の変数がもう一方の変数で陽的に表現できる必要がなく，そのためより複雑なグラフでも関数表示に出来ることです。

媒介変数表示の例として，円を見ることにします。通常，(a, b) を中心とする半径 $r \quad > 0$ の円は

$$(x-a)^2 + (y-b)^2 = r^2 \tag{3.10}$$

と表わします。これを中心角 θ を媒介変数とする媒介変数表示をすると

$$\begin{aligned} x &= r\cos\theta + a \\ y &= r\sin\theta + b \end{aligned}$$

となります。どちらも同じ円であることを MuPAD で確認しましょう。値が不定ではグラフを描くのに不都合なので，$a = -1, b = 3$ としておきます (図 3.11)。

```
>> a := -1: b := -3: r := 5:
>> plotlib::implicitplot(func((x-a)^2+(y-b)^2-r^2, x, y), -10..10, -10..10);
>> plot2d([Mode=Curve, [r*cos(t)+a, r*sin(t)+b], t=[0, 2*PI]]);
```

どちらも円というにはガタガタです。特に `implicitplot` 関数を使ったグラフは随分円周が太くなってしまっています。こうなってしまった理由は，円の方程式を満足する (x, y) が存在する範囲

[4] MuPAD 1.4.2 Pro だと，上の方程式は解けますが下の方程式は解けませんでした。

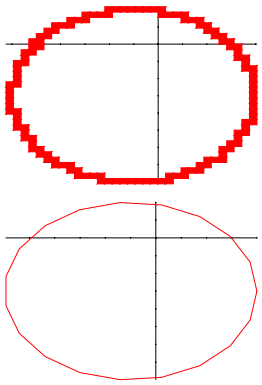

図 3.11: plotlib::implicitplot(..., & plot2d([Mode=Curve

の正方形を組み合わせて出力しているからで，その正方形の大きさの調整がうまくいっていないからだと思われます。

媒介変数表示を使った例では，”Grid=[100]”のように刻みを細かくしてやることで滑らかな円周になります。より滑らかにしたければ 100 より大きい数を指定して下さい。

媒介変数表示は計算時間の面からも有効です。もっと複雑な図形になると，グラフを描画する時間は implicitplot 関数の方が長くなるはずです。

最後に，媒介変数表示を使って渦巻きを描いてみましょう。渦巻きにも何種類かありますが，ここではバームクーヘンの巻き方を再現することにしましょう。バームクーヘンは一枚のスポンジ板をくるくると巻いて作られます。このスポンジ板の厚さを $d = 1$ とし，10 回巻いたとします。このときのバームクーヘンの断面に当たるのが図 3.12 です。

```
>> a = -1: b=-3: r=5: d := 1:
   plot2d([Mode=Curve, [(r+d*t/(2*PI))*cos(t)+a, (r+d*t/(2*PI))*sin(t)+b],
    t=[0, 20*PI], Grid=[300]]);
```

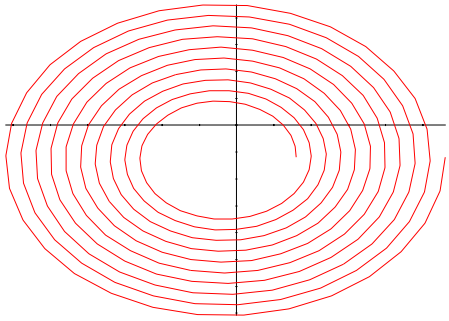

図 3.12: バームクーヘンの断面図

巻く回数が増えると Grid の数も増やす必要が出てきます。適当に調整して下さい。

練習問題 3.4
上の渦巻きを逆に巻くように改良して下さい。

第4章　極限・微分・積分

極限値を求める計算や，微分・積分の計算はかなりの部分，機械的な計算に終始します。従って，この手の分野も数式処理ソフトウェアにとっては得意な分野です。

4.1　極限

極限の計算には limit 関数を使います。これは次のようにして使います。

```
>> limit(1/n, n=infinity);
```

```
0
```

以上のように，limit(式，極限の指定) という形で呼び出します。無限大 $(+\infty)$ は infinity と書きます。当然，$-\infty$ は-infinity と書きます。

```
>> limit(n^2, n=infinity);
   limit(n^2, n=-infinity);
```

```
infinity
```

```
infinity
```

ではずらっとサンプルを並べてみましょう。

1.
$$\lim_{n\to\infty} \frac{n^2-2n+3}{-3n^2+1}$$

2.
$$\lim_{n\to\infty} \frac{1-n}{2+\sqrt{n}}$$

3.
$$\lim_{n\to\infty} \left(\sqrt{n+1}-\sqrt{n}\right)$$

以上の値を求めると，次のような結果を得ます。

```
>> limit((n^2-2*n+3)/(-3*n^2+1), n=infinity);
```

```
                                  -1/3
```

```
>> limit( (1-n) / (2+n^(1/2)) , n=infinity);
```

```
                                -infinity
```

```
>> limit((n+1)^(1/2) - n^(1/2), n=infinity);
```

```
                                   0
```

自然対数の底 $e = 2.71828...$ は

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e$$

と極限値で定義されていました。MuPAD はこの定義式も折り込み済みです。たとえば

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^{3n}$$

を計算してみましょう。

```
>> limit((1+1/n)^(3*n), n=infinity);
```

```
                                 exp(3)
```

このように，きちんと e^3 を返してくれます。

以上は数列の極限値計算でしたが，関数の極限も全く同様に求められます。関数 $f(x)$ を

$$f(x) = \frac{x^4 - 2x^5}{3x^3 - 4x^4 + 5x^5}$$

と定義して，次の値を求めてみましょう。

1.
$$\lim_{x\to 0} f(x)$$

2.
$$\lim_{x\to\infty} f(x)$$

3.
$$\lim_{x\to-\infty} f(x)$$

```
>> f := (x^4 - 2*x^5)/(3*x^3 - 4*x^4 + 5*x^5);

                                  4      5
                                 x  - 2 x
                               -----------------
                                  3      4      5
                               3 x  - 4 x  + 5 x

>> limit(f, x = 0);

                                     0

>> limit(f, x=infinity);

                                   -2/5

>> limit(f, x=-infinity);

                                   -2/5
```

練習問題 4.1
次の値を求めて下さい。

1. $$\lim_{x \to 0} \frac{x}{\sqrt{x+1}-1}$$
2. $$\lim_{x \to 0} \frac{1-\cos x}{x \sin x}$$
3. $$\lim_{x \to 0} \frac{\log(1+x+x^2)}{3x}$$
4. $$\lim_{x \to 0} (1+ax)^{\frac{1}{x}} \quad (a \neq 0)$$

4.2 微分

一変数関数を微分するには `diff` 関数を使います。文字式でも微分できるよう，通常はどれが変数か指定するため，`diff`(式，変数名) のように呼び出します。

$$3x^3 + 4x - 2$$

```
>> diff(3*x^3+4*x-2,x);
>> diff(a*x^3 + b*x^2 + c*x + d, x);
```

```
            2
9 x  + 4

                    2
c + 2 b x + 3 a x
```

いわゆる初等関数 (三角関数，対数関数，指数関数，多項式関数, etc) の組み合わせでできあがっている関数であればほとんど問題なく導関数を求めることができます。次の例で見ていくことにします。

1.
$$5\sin x + 7\cos x - e^x + 3\log x$$

2.
$$(2x^2 + 7x + 1)^2$$

3.
$$\cos x \sin x$$

4.
$$e^x \cos x + x^3 \log x$$

5.
$$\frac{1}{3x+7}$$

6.
$$e^{-x}$$

7.
$$\frac{e^x + 3x^2}{\log x}$$

8.
$$e^{-x} \sin x \tan x$$

9.
$$\frac{\log x + x^2 \sin x}{(2x+7)^3}$$

以下はその実行結果です。

```
>> diff(5*sin(x)+7*cos(x)-exp(x)+3*ln(x), x);

                                                3
                  5 cos(x) - 7 sin(x) - exp(x) + -

>> diff((2*x^2+7*x+1)^2,x);

                                       2
                    2 (4 x + 7) (7 x + 2 x  + 1)

>> diff(cos(x)*sin(x), x);

                                 2         2
                           cos(x)  - sin(x)

>> diff(exp(x)*cos(x)+x^3*ln(x), x);

                                          2      2
             cos(x) exp(x) - sin(x) exp(x) + x  + 3 x  ln(x)

>> diff(1/(3*x+7), x);

                                    3
                               - ----------
                                          2
                                 (3 x + 7)

>> diff(exp(-x),x);

                                 -exp(-x)

>> diff((exp(x)+3*x^2)/ln(x), x);

                                              2
                      6 x + exp(x)   exp(x) + 3 x
                      ------------ - -------------
                         ln(x)               2
                                      x ln(x)
```

```
>> diff(exp(-x)*sin(x)*tan(x), x);

tan(x) cos(x) exp(-x) - tan(x) sin(x) exp(-x) +

                         2
   sin(x) exp(-x) (tan(x)  + 1)

>> diff((ln(x)+x^2*sin(x))/(2*x+7)^3, x);

                         1    2
            2 x sin(x) + - + x  cos(x)              2
                         x               6 (ln(x) + x  sin(x))
            ------------------------- - ---------------------
                           3                        4
                  (2 x + 7)                (2 x + 7)
```

以上のように，ちょっと複雑な関数でも導関数を難なく求めることができます。

diff 関数は変数を明示する必要がありましたが，変数を別の方法で明示しておけば' 演算子でも代用できます。

```
>> y := poly(x^2,[x]);
   y'(x);
   z := func(exp(sin(x)), x);
   z'(x);

                                          2
                                    poly(x , [x])

                                         2 x

                                 func(exp(sin(x)), x)

                                  cos(x) exp(sin(x))
```

上の例では, まず y という変数に x を変数とする多項式 x^2 を代入してあります。ここで poly を使って変数が x であることを明示してあるわけです。また次の z も同様に，x が変数である関数 $\exp(\sin x)$ を明示しています。

練習問題 4.2

次の関数の導関数を求めて下さい。

1. $e^{x^{x^x}}$
2. $\log(2\log x^3)$

4.3 積分

積分には int 関数を使います。まず，多項式関数 $f(x) = 4x^2 + 3x^2 - 6x + 1$ を定義して，これの不定積分を求めてみます。

```
>> f := 4*x^2 + 3*x^2  - 6*x + 1;
```

```
                                    2
                                 7 x  - 6 x + 1
```

```
>> int(f, x);
```

```
                                          3
                                     2   7 x
                                 x - 3 x  + ----
                                          3
```

定積分の場合は，変数 x に積分区間を指定します。

```
>> int(f, x=0..5);
```

```
                                    665/3
```

では，積分の線型性を示す以下の公式を MuPAD で確認してみましょう。そのために適当なもう一つの関数 $g(x)$ を定義しておきます。

$$\begin{aligned}\int (f(x) + g(x))\, dx &= \int f(x)dx + \int g(x)dx \\ \int a f(x)dx &= a \int f(x)dx\end{aligned}$$

```
>> g := exp(x) + x^3;
```

```
                                      3
                                 exp(x) + x
```

```
>> int(f+g, x);
   int(f, x)+int(g, x);
   int(a*f, x);
   simplify(a*int(f, x));
```

```
                                        3    4
                                   2   7 x    x
```

```
                                  3     4
x + exp(x) - 3 x  + ---- + --
                    3     4

                      3    4
                  2   7 x    x
x + exp(x) - 3 x  + ---- + --
                    3     4

                        3
                  2   7 k x
    k x - 3 k x  + ------
                    3

                        3
                  2   7 k x
    k x - 3 k x  + ------
                    3
```

次に，微分積分の基本定理と呼ばれる,「微分と積分は逆演算の関係にある」ことを確認してみましょう。

```
>> F := int(f, x); G := int(g, x);
   dF := diff(F, x); dG := diff(G, x);
   int(dF, x); int(dG, x);
```

```
                      3
                 2   7 x
x - 3 x  + ----
                 3

                  4
                 x
  exp(x) + --
                 4

   2
7 x  - 6 x + 1

                  3
  exp(x) + x
```

```
                    3
          2    7 x
x - 3 x   + ----
               3


               4
              x
  exp(x) + --
              4
```

二つの関数に囲まれた領域の積分を求めてみます。まず関数を定義し，囲まれた領域を図示してみましょう (図 4.1)。

```
>> f := x^2-2*x:
   g := -x^2+4:
   plotfunc(f, g, x=-10..10);
   plotfunc(f, g, x=-2..3);
```

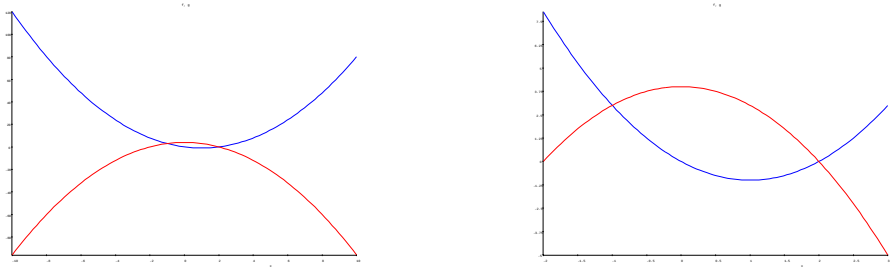

図 4.1: plotfunc(f, g, x=-10..10) & plotfunc(f, g, x=-2..3)

囲まれた領域の端点の x 座標を正確に求めてみます。

```
>> solve(f-g, x);
```

```
                                {-1, 2}
```

従って，この領域の面積は以下のように計算すればいいですね。

```
>> int(g-f, x=-1..2);
```

```
                                 9
```

球の体積を求めてみましょう。

```
>> 2*PI*int((r^2-x^2), x=-r..0);
```

$$\frac{4\,r^3\,PI}{3}$$

最後は円周の長さを求めています。何をやっているか分かりますか?

```
>> f := r*cos(t); g := r*sin(t);
   int(sqrt(diff(f, t)^2+diff(g, t)^2), t=0..2*PI);
```

$$r\cos(t)$$

$$r\sin(t)$$

$$2\,PI\,(r^2)^{1/2}$$

練習問題 4.3
第 3 章の最後に描いた渦巻きの長さを求めて下さい。

4.4　常微分方程式

この節では `solve` 関数を常微分方程式に応用します。常微分方程式の定義には `ode` 関数を使います。引数は `rec` 関数と同様です。

```
>> solve(ode(y'(x) = y(x), y(x)) );
```

$$\{C1\,\exp(x)\}$$

ここでは，まず $y' = y$ という式が常微分方程式であることを `ode` 関数で指定し，それを `solve` 関数に解かせています。

上の場合は一般の解析解を求めていますが，初期値を与えることで，これを一意に定めることが出来ます。

```
>> solve(ode({y'(x) = y(x), y(0) = 1}, y(x)) );
```

$$\{\exp(x)\}$$

多次元のものでも解くことが出来ます。以下は 2 次元 1 階線型常微分方程式の例です。

```
>> solve( ode({y1'(x) = 2*y2(x), y2'(x) = 3*y1(x)}, {y1(x), y2(x)}) );
```

```
                                                    y

{ {              1/2        1/2         1/2          1/2
{ {          C4 6    exp(x 6   )   C3 6    exp(- x 6   )
{ { y1(x) = ------------------- - ---------------------,
{ {                   3                      3

                                             } }
                   1/2                 1/2   } }
y2(x) = C4 exp(x 6   ) + C3 exp(- x 6   ) } }
                                             } }
```

ただし，積分と同様，全ての常微分方程式が解析的に解けるわけではありません。その時には別の方法を考えねばなりませんが，ここでは触れないでおきます。興味がある人は MuPAD のヘルプで `numeric::odesolve` 関数を参照して下さい。

練習問題 4.4
上の解が常微分方程式を満足することを確認して下さい。

終わりに

皆さんが今まで習ってきた高校の数学を題材として，統合数学ツールとしての数式処理ソフトウェアの機能の一端を見てきました。どのような感想を持たれたでしょうか?　以前よりも機能はアップしているとはいえ，所詮は融通の利かないコンピュータですから，イライラすることもあったでしょうが，今まで苦労してきた計算の大部分を肩代わりすることができる，ということはご理解いただけたと思います。

では，このような数式処理ソフトウェアの出現で，人間の仕事は楽になるのでしょうか?

そんなことはありません。

前書きで述べましたが，これからの人間の役割は考えることです。コンピュータのなかった時代は，予測はできてもそれを検証するだけの莫大な計算を行うことはできませんでした。そのたががはずれた今，可能性は広がり，仕事は増える一方なのです。科学技術が発達しているということは，とりもなおさず，それだけ未知の事柄が存在しているという証にすぎないのです。

ちょっと心配なのは，数式処理ソフトウェアに頼るあまり，自身の計算能力だけではなく，推論能力まで衰えてしまうのではないかということです。確実にいえるのは，計算能力は格段に落ちるということです。しかし，推論能力まで落ちるかどうかは，使い方次第でしょう。このあたりは人によって議論の分かれるところですが，著者らは問題ないという立場です。日常の買い物のお釣り計算まで計算機がなければできないという極端な依存症になってしまえば別ですが，日頃から自分の頭をちょっとでもいいから使うことを心がけている人ならば，という条件付きではありますが。

数式処理ソフトウェアに限らず，コンピュータ関係の書籍や記事などでは，このような一種の未知への恐れか，ろくに検証もしていない肯定派の識者の意見をそのまま並べたような手放しの礼賛の両極端であることが多いようです。新技術が人間に及ぼす影響は，千差万別です。結果から統計的な分類はできるでしょうが，未来予測は簡単ではありません。個々人が心がけることは，自分にとってどういう役に立つか，冷静に考えることだけです。自分にとって，その新技術がどのように役立ち，その結果どのような悪影響があるか，長所と短所のバランスを見極めた上で，どのようにつき合っていくかを自分自身で導く他はないのです。

数式処理ソフトウェアがあなたにとってどのような意味を持つのか?　これに対する回答はご自身で見つけてください。この文書がその判断の材料になれば，著者としてこれに勝る喜びはありません。

関連図書

[1] http://www.inetshonai.or.jp/ seawave/history/index.html, seawave@inetshonai.or.jp

[2] http://www.truebasic.com/index.html, webmaster@truebasic.com

[3] 藤田宏・前原昭二編, 数学 I, 東京書籍, 1998.

[4] 藤田宏・前原昭二編, 数学 II, 東京書籍, 1998.

[5] 藤田宏・前原昭二編, 数学 III, 東京書籍, 1998.

[6] 藤田宏・前原昭二編, 数学 A, 東京書籍, 1998.

[7] 藤田宏・前原昭二編, 数学 B, 東京書籍, 1998.

[8] 藤田宏・前原昭二編, 数学 C, 東京書籍, 1998.

付 録A　MuPADの入手とインストール方法

MuPAD についての情報は `http://www.mupad.de/` から辿れます。

A.1　MuPAD とは?

MuPAD は，ドイツ・Paderborn 大学[1]の B.Fuchssteiner 教授の監修により，MuPAD 研究グループが作成した数式処理ソフトウェアです。1989 年に開発が開始され，SciFace 社が加わって現在の様々な OS で動作する MuPAD が生み出されてきました。サポート OS は以下の通りです[2]。

サポートなし・無料

- Amiga/NetBSD
- HP-UX
- Convex
- DEC/Compaq Alpha/OSF
- DECstation/Ultrix
- IBM RS6000
- Machintosh 68K 系統
- OS/2
- Linux on PPC & MKLinux
- Solaris/NeXT/FreeBSD/OpenBSD on PC
- SGI/Irix 5.x/6.x

教育機関の職員・学生なら無料

- (MuPAD Light のみ)Windows 95/98/NT
- Macintosh/PPC
- Linux on PC
- Sun Sparc/Solaris & SunOS

[1] `http://www.uni-paderborn.de/`
[2] 1999 年 9 月 2 日現在。

A.2 インストール方法 (Light 版)

1. `http://www.mupad.de/` の”Download”から辿り，必要事項を記入して，ダウンロードサイトを選択します。

2. あるいは直接ダウンロードサイトから ftp で入手する手もあります。日本国内では

   愛媛大学 ftp://archives.cs.ehime-u.ac.jp/pub/MuPAD/distrib/windows/

   長崎大学 ftp://kankoromochi.econ.nagasaki-u.ac.jp/pub/MuPAD/

   がミラーサイトになってますので，ここから落とすと良いでしょう。ただ，どのみちライセンスは直接本家に連絡して取得する必要がありますので，1. の手順をお勧めします。

3. mupad_light_141.exe を適当なフォルダにダウンロードします。

4. ダウンロード終了後，この実行ファイルをダブルクリックします。途中，ライセンス番号を聞いてくるはずですので，未取得の場合は無視，取得済みの場合はそれを入力して下さい。ライセンス番号はインストール後でも入力できます。

5. 「MuPAD Light」がスタートメニューにあることを確認して下さい。これでインストール作業は終了です。

付 録B　その他の数式処理ソフトウェア

全て **1999 年 9 月 2 日現在** の情報です。

Mathematica

制作者 Wolfram Research Inc.

最新バージョン Version 4

参照 URL

日本法人 `http://www.wolfram.co.jp/`

米国本社 `http://www.wolfram.com/`

Macsyma

製作者 Macsyma Inc.

最新バージョン 2.4

参照 URL `http://www.macsyma.com/`

Reduce

制作者 Anthony Hearn

最新バージョン 3.7

参照 URL `http://www.rrz.uni-koeln.de/REDUCE/`

Maple

制作者 Waterloo Maple Inc.

最新バージョン V

参照 URL `http://www.maplesoft.com/`

DERIVE

製作者 David Stoutemyer/Soft Warehouse Inc.

最新バージョン 4.11 (Windows 版)

参照 URL `http://www.derive.com/`

UBASIC

製作者 木田祐司

最新バージョン 8.8c

参照 URL http://www.rkmath.rikkyo.ac.jp/~kida/ubasic.htm