第1章　Scilab について

1.1　はじめに

本資料の PDF ファイルや Scilab のサンプルプログラムのファイルは,「確率論・数値解析及び演習」のページからダウンロードできます．資料を授業で受け取れなかった場合や，紛失した場合は，各自必要に応じてダウンロードや印刷をして下さい．

http://www.isee.nagoya-u.ac.jp/~shinimada/

1.2　Scilab とは

Scilab (サイラボ) は，フランスで開発された数値計算用プログラミング言語のフリーウェアで，行列処理やグラフィック表示機能なども充実している．研究機関や企業等で広く利用されている MATLAB (　有料) と仕様が類似した言語であり，MATLAB　のフリー版とも言える．演習では Windows 版の Scilab を利用するが，その他に linux 版・Mac 版も公開されている．自宅の PC 等に Scilab をインストールするには，下記の URL を参考にするとよい．

http://www.scilab.org/download/　(Scilabのダウンロードのページ)

1.3　Scilab の起動と終了

1.3.1　起動

デスクトップ上のアイコン「scilab-5.4.1」をダブルクリックして Scilab を起動すると，図 1.1 のような Scilab ウィンドウが立ち上がる．デスクトップにアイコンが無い場合は,「スタート」ボタンから「すべてのプログラム」を選び，scilab ディレクトリ内の「scilab-5.4.1」を起動する。

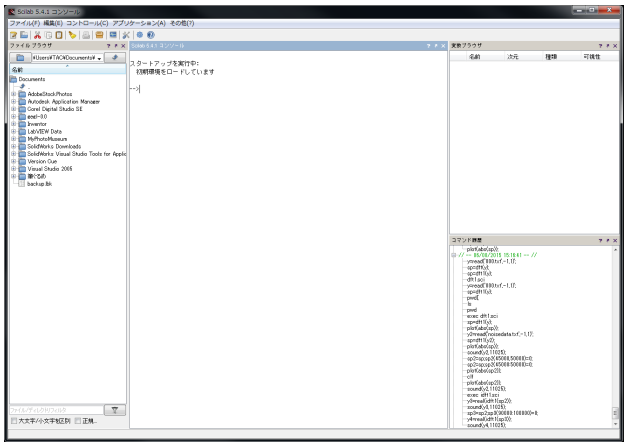

図 1.1: Scilab ウィンドウ

1.3.2　終了

Scilab を終了するには，Scilab ウィンドウのプロンプト (--> の行) で “exit” と入力するか，左上の “File” メニューから “Exit” を選択する．(原因不明のエラーが生じた場合は，一度 Scilab を終了して起動し直すとよい．)

1.4 ヘルプの使い方

```
--> help  [調べたい関数名・構文名]
```

(例)

```
--> help plot       // plot 関数の使い方を調べる
--> help if         // if 文 (if then else) の使い方を調べる
```

1.5 変数への値の代入と消去

C 言語では変数を利用する前に変数の型を宣言する必要があるが，Scilab では事前に宣言する必要はない．

```
--> a=1                  // 変数 a に 1 を代入する場合 a=1 と記述する．
 a  =
    1.                   // 代入した結果が Scilab ウィンドウ内に表示される．

--> a=1;                 // 式の最後にセミコロン (;) を付けると結果は表示されない．

--> a=1, b=2; c=3; A=4   // 同一行の複数の式はコンマ (,) かセミコロン (;) で区切る．
 a  =                    // 変数の大文字 (a) と小文字 (A) は区別される．
    1.
 A  =
    4.                   // セミコロン (;) を付けた式の結果は表示されない．

--> a                    // 定義済みの変数 a の値の表示
 a  =
    1.

--> clear                // これまでに代入された変数の値を全て消去する．

--> a                    // 再度，変数 a の値の表示してみる．
  !--error     4          // clear が実行された後なので，
undefined variable : a   // 変数 a の値は定義されていないというエラーメッセージが出る
```

1.6 四則演算

```
--> 10+3*(16-2^3)/4
 ans  =
    16.

--> ans                  // 直前の演算結果は ans という変数に代入される．
 ans  =
    16.

--> a=1; b=2; c=3; A=4; d=a+b*c-A
 d  =
    3.
```

表 1.1 に四則演算等の記号をまとめる．但し，始めにドット「.」が付く演算子「.*」「./」「.^」は，ベクトル同士や行列同士の演算において，対応する (位置が同じ) 要素間で演算を行う場合に用いる．詳しくは 1.9 節で説明する．

表 1.1: 演算記号

演算	加算	減算	乗算	除算	累乗
表記	a+b	a-b	a*b a.*b	a/b a./b	a^b a.^b

1.7 特殊変数

Scilab では，円周率 π や虚数 i などの特殊変数が用意されている．それらの多くは頭に%が付く．

```
--> x = %pi          // 円周率: %pi
 x  =
    3.1415927

--> y = 3+2*%i       // 虚数単位: %i
 y  =
    3. + 2.i
--> real(y)          // real 関数で y の実数部を表示
 ans  =
    3.
--> imag(y)          // imag 関数で y の虚数部を表示
 ans  =
    2.

--> e = %eps         // 1 + eps > 1 となる最小の値 (マシン・イプシロンと呼ばれる．テキスト p.8)
 e  =                // %eps は (1/2) の 52 乗に等しい．52 は double 型の仮数部のビット数
    2.220E-16

--> 20-15;
--> ans              // 直前の演算結果: ans (ans には%が付かない)
 ans  =
    5.
```

1.8 format 関数による数値表示の桁数の制御

Scilab で数値を表示する場合，デフォルトでは 8 桁しか表示されない．

```
--> x = 3.14159265358979323846264338327950288419716
 x  =
     3.1415927                                  // 8 桁しか表示されない
```

表示する桁を増やすには，format 関数を使う．

```
--> format(14)                                  // 指定した数-2 桁表示される
--> x
 x  =
    3.14159265359                               // 12 桁表示される
--> format(34)
--> x
 x  = 3.1415926535897931159979634685442         // 32 桁表示されるが，16 桁以上の精度はない．
```

1.9 ベクトルや行列の扱い

1.9.1 ベクトルの表現

```
--> x = 0:10              // 0 から 10 までを要素に持つベクトルを作る．
 x  =
    0.    1.    2.    3.    4.    5.    6.    7.    8.    9.    10.

--> y = 1:3:10            // 1 から 10 まで 3 刻みの要素を持つベクトルを作る．
 y  =
    1.    4.    7.    10.

--> a = y(3)              // ベクトル y の 3 番目の要素を取り出す．(添字は 1 から始まることに注意)
 a  =
    7.
```

```
--> a = y(5)            // ベクトル y の 5 番目の要素を取り出そうとする.
         !--error  21   // 5 番目の要素は存在しないので添字が不適切とのエラーメッセージが出る.
invalid index

--> a = y(0)            // ベクトル y の 0 番目の要素を表示.
          !--error  21  // ベクトルの添字は必ず 1 から始まるので, 0 以下はエラーとなる.
invalid index

--> b = y($)            // ベクトル y の最後の要素を取り出す.
 b  =
    10.

--> c = x(1:3)          // ベクトル x の最初〜3 番目の要素を取り出す.
 c  =
    0.    1.    2.

--> d = x(2:3:$)        // ベクトル x の 2 番目〜最後の要素まで 3 つ飛ばしに取り出す.
 d  =

    1.    4.    7.    10.

--> z = y'              // ベクトルの転置
 z  =
    1.
    4.
    7.
    10.

--> y+y                 // ベクトル同士の和
 ans  =
    2.    8.    14.    20.

--> sum(y)              // ベクトルの全要素の和
 ans  =
    22.

--> prod(y)             // ベクトルの全要素の積
 ans  =
    280.

--> y*y'                // y と y の転置ベクトルの積（結果はスカラー）
 ans  =
    166.

--> y'*y                // y の転置ベクトルと y の積（結果は行列）
 ans  =
    1.     4.     7.     10.
    4.     16.    28.    40.
    7.     28.    49.    70.
    10.    40.    70.    100.

--> y.*y                // y の要素同士の積（結果はベクトル）
 ans  =
    1.    16.    49.    100.
```

1.9.2 行列の表現

```
--> A = [1,2,3;4,5,6]     // 2行×3列の行列．行はセミコロン(;)で区切る．
 A  =                     // 列はコンマ(,)またはスペースで区切る．A=[1 2 3; 4 5 6] でも可．
    1.    2.    3.
    4.    5.    6.

--> A = [1,2,3            // または改行でも区切ることができる．
--> 4,5,6]
 A  =
    1.    2.    3.
    4.    5.    6.

--> x = A(2,1)            // 2行目の1列目の要素を取り出す．(行列の添字は1から始まることに注意)
 x  =
    4.

--> y = A(1,:)            // 1行目を全て取り出す．
 y  =
    1.    2.    3.

--> z = A(:,2)            // 2列目を全て取り出す．
 z  =
    2.
    5.

--> A'                    // Aの転置行列を表示する．
 ans  =
    1.    4.
    2.    5.
    3.    6.

--> B = eye(3,3)          // 3行×3列の単位行列を作る．
 B  =
    1.    0.    0.
    0.    1.    0.
    0.    0.    1.

--> C = zeros(2,3)        // 2行×3列の要素が全て0の行列を作る
 C  =
    0.    0.    0.
    0.    0.    0.

--> D = ones(2,3)         // 2行×3列の要素が全て1の行列を作る
 D  =
    1.    1.    1.
    1.    1.    1.

--> A.*A                  // 行列Aの要素同士の積
 ans  =
    1.     4.     9.
    16.    25.    36.

--> A'*A                  // Aの転置行列とAの積
 ans  =
    17.    22.    27.
    22.    29.    36.
    27.    36.    45.

--> 2*A                   // スカラーと行列の積
 ans  =
    2.    4.     6.
    8.    10.    12.
```

表 1.2: 論理演算の記述の仕方

論理演算	$a < b$	$a \leq b$	$a > b$	$a \geq b$	$a = b$	$a \neq b$	and	or
Scilab での演算表記	a<b	a<=b	a>b	a>=b	a==b	a~=b or a<>b	&	|

```
--> C = [1,2;3,4]          // 2x2 の正方行列 C を定義
 A  =
    1.    2.
    3.    4.

--> C*C                    // C*C の計算（通常の行列の積）
 ans  =
    7.    10.
    15.   22.

--> C.*C                   // C.*C の計算（行列の要素同士の積）
 ans  =                    // C*C と C.*C の結果が異なることに注意
    1.    4.
    9.    16.
```

1.10 条件文，繰り返し文

1.10.1 if 文

```
if 〔条件式〕 then          // then は 条件式の後に改行またはコンマを入れれば省略もできる
    〔命令〕
elseif 〔条件式〕 then
    〔命令〕
else
    〔命令〕
end
```

なお，条件文などで利用できる論理演算記号には，表 1.2 のようなものがある．

＜例＞

```
--> score = 70;
--> if score > 80 then
-->   disp("Well done.");
--> elseif score > 60 then
-->   disp("Good.");
--> else
-->   disp("Be serious.");
--> end
 Good.                        // 2 つ目の条件文が実行される．
```

1.10.2 for 文

```
for〔ループ変数〕=〔ループ変数の範囲〕
    〔命令〕
end
```

＜例＞

```
--> s = 0;                    // 和を代入する変数を s=0 で初期化
--> for i = 1:10              // for i=[1 2 3 ... 10] に同じ
-->   s = s + i;              // s に i を加えて行く
--> end
--> s                         // 最終的な和 s の値を表示
 s  =
    55.
```

```
--> for k=0:2:4             // for k=[0 2 4] に同じ
-->   disp(k);              // k の値を表示
--> end
    0.
    2.
    4.

--> for k=3:-1:0            // for k=[3 2 1 0] に同じ
-->   disp(k);              // k の値を表示
--> end
    3.
    2.
    1.
    0.
```

1.10.3 while 文

```
while〔条件式〕
    〔命令〕
end
```

＜例＞

```
--> a = 3;
--> while a > 0
-->   disp(a);
-->   a=a-1;
--> end
    3.
    2.
    1.
```

1.10.4 select-case 文

C 言語 switch-case 文に相当するものである．

```
--> n = ceil(4*rand())     // 0〜4 の乱数を発生し，小数を切上げた 1,2,3,4 いずれかの整数を n に代入
--> select n
--> case 1 then
-->   disp("odd number!")  // n = 1 であれば odd number! (奇数) と出力
--> case 3 then
-->   disp("odd number!")  // n = 3 の場合も odd number! (奇数) と出力
--> else
-->   disp("even number!") // その他 n = 2 or 4 の場合は even number! (偶数) と出力
--> end
 n  =                      // この実行例は，発生した乱数が 2.xxxx だった場合．切上げて n=3
    3.
 odd number!               // case 3 に当てはまるため odd number! と出力
```

1.11 スクリプトファイルの実行

複数の命令をまとめて実行するには，Scilab コンソールに一つずつ命令を入力するよりも，あらかじめその命令の列をファイルに記述し (これを Scilab スクリプトと呼ぶ)，このスクリプトファイルから実行する方が便利である．まず，実行したい命令の列を，SciNotes 等を利用してファイルに記述する．例えば，次のような内容をファイルに記述し，sample.sci という名前で保存する．保存するファイル名の拡張子は何でもよいが，分かりやすくするため，ここでは「.sci」とする．

(注) ファイルを保存する場所は「ホームディレクトリ」の下とすること．例えば,「ホームディレクトリ」の中に「n-ana」という名前のフォルダを作り，その中にスクリプトや関数のファイルを保存するとよい．(デスクトップ等にファイルを保存しても，次回以降ログインした時にファイルが消えている可能性があるため．)

```
// --- sample.sci の中身 -----------
a=10;
b=20;
x=a*b
// ---------------------------------
```

Scilab スクリプトを実行するには，まず，左上の “File” メニューから “Change Directory” を選択し，立ち上がるウィンドウの中から，スクリプトファイルを保存したフォルダを選択して，そのフォルダに移動する．次に，“File” メニューから “Exec...” を選択し，リストの中から実行したいスクリプトのファイル名 (ここでは，sample.sci) を選び，“開く” ボタンで実行すればよい．

```
--> a=10;
--> b=20;
--> x=a*b
 x  =
    200.
```

または，exec 関数を用いることもできる．

```
--> exec('sample.sci')        // ファイル名はシングルクォート' 'またはダブルクォート" "で括る.
--> a=10;
--> b=20;
--> x=a*b
 x  =
    200.
```

1.12 コメント

Scilab で，// は，コメント記号としてみなされる．つまり，Scilab のスクリプトファイルや関数ファイルにおいて，行頭や行の途中に // と記述すれば，その行の右側は全て無視される．

```
--> a=1    // a=2              ← a=2 の部分は無視される.
 a  =
    1.
```

1.13 関数の定義と使い方

関数は，スクリプトとは別のファイルに記述して保存し，それを読み込んで利用する．

1.13.1 (例 1) 2 変数の積を返す関数 multiply の定義 (引数 2 つ，返り値 1 つ)

例として，ファイル multiply.sci に 2 変数の積を返す Scilab 関数 multiply を記述する．関数名とファイル名を同じにする必要はないが，統一しておくとわかりやすい．

```
// --- multiply.sci の中身 -------------
function z = multiply(x,y)                  // 返り値 1 つ，引数 2 つの関数 multiply
z = x*y                                     // x と y の積を返す
endfunction                                 // 関数の定義の終わり (endfunction は省略可能)
// ---------------------------------
```

Scilab 関数のファイルを読み込んで関数を利用できるようにするためには，Scilab スクリプトファイルの読み込みと同様に，Scilab ウィンドウの左上の “File” メニューの “Exec...” を選択し，立ち上がるウィンドウの中から読み込みたい関数のファイル名 (ここでは，multiply.sci) を選び，“開く” ボタンをクリックすればよい．これで関数の読み込みが完了し，multiply という関数が利用できるようになる．

```
--> multiply(10,5)*2
 ans  =
    100.
```

または，exec 関数を用いて関数を読み込むこともできる．

```
--> exec('multiply.sci')
--> multiply(10,5)*2
 ans  =
    100.
```

1.13.2 (例 2) 円周と円の面積を返す関数 circle の定義 (引数 1 つ，返り値 2 つ)

通常，C 言語等では返り値が複数ある関数は定義できない (ポインタ等を利用する必要がある) が，Scilab では返り値が複数の関数も簡単に記述できる．

```
// --- circle.sci の中身 -----------
function [perimeter, area] = circle(r)    // 引数 1 つ， 返り値 2 つの関数 circle
perimeter = 2 * %pi * r;                  // 1 つ目に「円周」を返す
area = %pi * r * r;                       // 2 つ目に「円の面積」を返す
endfunction                               // 関数定義の終わり
// ---------------------------------
```

関数 circle をファイルから読み込んで利用するには，以下のようにする．

```
--> exec('circle.sci')                    // circle 関数の読み込み
--> [p,a]=circle(3)                       // 半径 3 の円周と円の面積
 a  =                                     // 返り値が複数なら，代入先もベクトル (例:[p,a]) とする
    28.274334                             // 円の面積が a
 p  =
    18.849556                             // 円周が p (p と a の表示の順番は逆となる)
```

1.13.3 (例 3) 仮数部の 0,1 の並びを返す関数 significand の定義 (引数 1 つ，返り値 1 つ)

この関数は，ある 1 以下の数字 x を倍精度の浮動小数点で表現した場合の仮数部 (52 ビット) の 0,1 の並びを文字列として返す関数である．

```
// --- significand.sci の中身 -----
function bit = significand(x)       // 0<x<1 の仮数部の 0,1 の並び (52bit 分) を返す関数
bit = '';                           // 仮数部を表す文字列 bit を空の文字列として初期化する
while x < 1                         // x が 1 以上になるまで
  x = x*2;                          // x を 2 倍する
end
x = x - 1;
for j = 1:52                        // 仮数の桁数 (52 回) 繰り返す
  if x >= 0.5                       // x が 0.5 以上であれば
    bit = bit + '1';                // 仮数部 bit の後ろに 1 を追加し
    x = x - 0.5;                    // x から 0.5 を引く (最左ビットをの 1 を除く)
  else                              // x が 0.5 未満であれば
    bit = bit + '0';                // 仮数部 bit の後ろに 0 を追加
  end
  x = x*2;                          // x を二倍して一桁左へシフトする
end
endfunction
// ---------------------------------
```

```
--> exec("significand.sci")                 // significand 関数の読み込み
--> significand(0.0328)                     // 0.0328 の仮数部
 ans  =
 0000110010110010100101011110100111100001101100001001
```

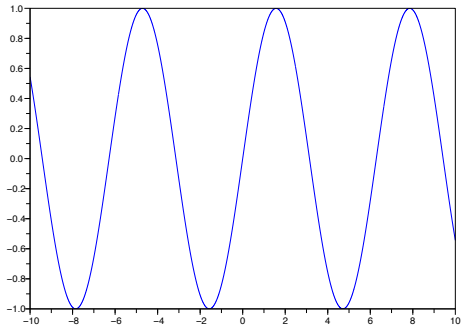

図 1.2: plot 関数の実行例 (sin(x))

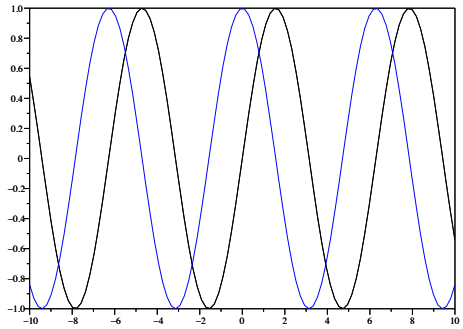

図 1.3: plot2d 関数の実行例 (sin(x), cos(x))

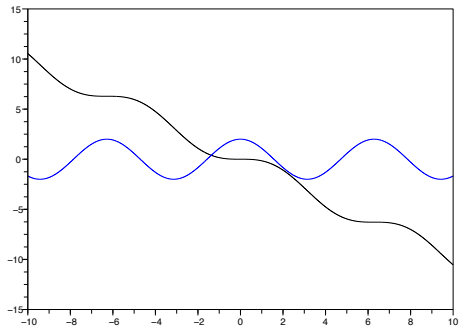

図 1.4: plot2d 関数の実行例 (sin(x)-x, 2cos(x))

1.14 グラフの描画

```
--> x=-10:0.001:10;    // x は-10〜10 まで 0.001 刻み. x=[-10 -9.999 -9.998 ... 9.999 10] に同じ
--> y=sin(x);          // y=[sin(-10) sin(-9.999) ... sin(9.999) sin(10)] に同じ
--> plot(x,y);         // sin(x) のグラフの表示 (図 1.2). plot(x,sin(x)) でも OK

--> mtlb_hold off      // MATLAB 関数の hold (mtlb_を付けることで MATLAB 関数が使える場合がある. )
                       // 現在表示されているグラフを消去して新しいグラフで上書きするモードにする
                       // 逆に mtlb_hold on とすればグラフを重ね合わせて表示できる
--> y=sin(x);
--> z=cos(x);
--> plot2d(x',[y' z']);       // sin(x), cos(x) のグラフの表示 (図 1.3)
                              // 2 つ以上のグラフは plot2d 関数で同時に描画できる
                              // 但し, plot2d では縦ベクトルで指定する (' は転置を表す)
--> mtlb_hold off
--> y=sin(x)-x;
--> z=2*cos(x);
--> plot2d(x',[y' z']);       // sin(x)-x, 2cos(x) のグラフの表示 (図 1.4)
```

1.15 グラフの保存

Scilab でプロットしたグラフをファイルとして保存するには，グラフのウィンドウの File メニューから，Export を選択する．そこで，保存する図の形式・ファイル名を指定し，Ok ボタンで保存する．

第2章　計算機上での数値表現

計算機上で数値を表現する主な方法としては，整数を表すための整数表現と，実数を表すための浮動小数点表現がある．浮動小数点表現の他に，固定小数点表現があるが，ここでは触れない．

2.1　整数表現

計算機上の整数の表現方法には，大きく分けて，0 以上の整数のみを表す符号なし (unsigned) 整数と，負の数も含めた整数を表す符号付き (signed) 整数の 2 種類がある．

2.1.1　符号なし整数表現

C 言語の unsigned char 型, unsigned short 型, unsigned int 型, unsigned long 型等がこれに当たる．多くの計算機では，unsigned char は 1 バイト (8 ビット)，unsigned short は 2 バイト (16 ビット)，unsigned int は 4 バイト (32 ビット)，unsigned long は 8 バイト (64 ビット) で，0 以上の整数を表現する．例えば，unsigned char 型であれば，8 桁の 0,1 の並びで 2 進数 00000000〜11111111 (10 進数の 0〜255) の 256 種類の整数を表現できる．

2.1.2　符号付き整数表現

C 言語の char 型, short 型, int 型, long 型等がこれに当たる．多くの計算機では，char は 1 バイト (8 ビット)，short は 2 バイト (16 ビット)，int は 4 バイト (32 ビット)，long は 8 バイト (64 ビット) で，負の数〜正の数までを表現している．例えば，char 型であれば，-128〜127 までの 256 種類の整数を表現できる．(負の数の方が，表現できる数字が 1 つ多い．)

ここで，負の数は「2 の補数表現」で表されている．ある負の数の 2 の補数表現は，その数の絶対値を 2 進数で表したものの 0 と 1 を反転させて，1 を加えたものに等しい．例えば，char 型で -100 を 2 の補数表現で表す場合，$100 = 64 + 32 + 4 = 2^6 + 2^5 + 2^2$ であるから，その 2 進数 1100100 の頭に 0 を埋めて 8 桁に拡張した 01100100 の 0 と 1 を反転させて 10011011 とし，それに 1 を加えた 10011100 が -100 の 2 の補数表現となる．

2.2　浮動小数点表現

浮動小数点表現は，次式のように表される実数を「符号部」「指数部」「仮数部」の組合せで表現する方法である．代表的なものに 4 バイト (32 ビット) の単精度，8 バイト (64 ビット) の倍精度の表現方法がある．

$$(-1)^{「符号部」} \times (1 + 0.「仮数部」) \times 2^{(「指数部」-ゲタ)}$$

2.2.1　単精度の浮動小数点表現

単精度 (4 バイト) 浮動小数点表現は，C 言語の float 型に相当する．符号部 1 ビット，指数部 8 ビット，仮数部 23 ビットの合計 32 ビット (32 桁) の 0,1 の並びで数値が表現される．符号部は，正の数の場合 0，負の数の場合 1 となる．指数部は 8 ビットで -127〜128 の範囲を表現できるが，実際は，127 の「ゲタ」が足された形で 0〜255(符号なし 8 ビット整数) で表現されている．つまり，指数部の数値から 127 を引けば，実際の指数の値となる．

但し，2.2.3 節で述べるように，指数部が 0 (全てが 0) と 255 (全てが 1) の場合は，特殊な数値に割り当てられているため，特殊な数値を除く数値 (正規化数) については，指数部が 1〜254 の範囲，つまり，指数が -126〜127 乗まで (128 乗未満) の範囲で数値を表現できる．

例えば，10 進数の -18.3125 は，

$$-18.3125 = -(16 + 2 + 0.25 + 0.0625) = -(2^4 + 2^1 + 2^{-2} + 2^{-4}) = -(10010.0101)_2 = -(1.00100101)_2 \times 2^4$$

であるから，符号部は負の数を表す 1, 指数部は 2^4 の指数 4 にゲタ 127 を足した 131 の 2 進数 10000011，仮数部は 1.00100101 の「1.」の続き 00100101 の右側に 0 埋めした 00100101000000000000000 となる．つまり，全体 32 ビットの 0,1 の並びは次の通りとなる．

11000001100100101000000000000000 ← −18.3125 の float 型での浮動小数点表現

その他の例として，−32768 を float 型で表現することを考えると，

$$-32768 = -2^{15} = -(1.0)_2 \times 2^{15}$$

であるから，符号部が 1，指数部が 2^{15} の 15 にゲタ 127 を加えた 142 の 2 進数 10001110, 仮数部は，1.0 の 1. を除いた 0，つまり 23 桁全てが 0 の並びとなる．

11000111000000000000000000000000 ← <u>−32768 の float 型での浮動小数点表現</u>

ここで，同じ数字 −32768 を int 型で表現することを考えると，

00000000000000001000000000000000 (32768 の 2 進数)
↓ 0 と 1 を反転
11111111111111110111111111111111 (−32768 の 1 の補数表現)
↓ 1 を加える
11111111111111111000000000000000 ← <u>−32768 の int 型での整数表現</u> (−32768 の 2 の補数表現)

となり，int 型・float 型共に同じ 4 バイト (32 桁) であるが，計算機上では 0,1 の並びが異なることがわかる．

2.2.2 倍精度の浮動小数点表現

倍精度 (8 バイト) 浮動小数点表現は，C 言語の double 型に相当する．符号部 1 ビット，指数部 11 ビット，仮数部 52 ビットの合計 64 ビット (64 桁) の 0,1 の並びで数値が表現され，単精度 (4 バイト) の場合の 2 倍の桁数で，より範囲の広い実数を表現できる．単精度の場合と同様に，符号部は，正の数の場合に 0，負の数の場合に 1 となる．

指数部は 11 ビットで −1023〜1024 の範囲の指数を表現できるが，実際は，1023 の「ゲタ」が足された形で 0〜2047(符号なし 11 ビット整数) で表現されている．つまり，指数部の数値から 1023 を引けば，実際の指数の値となる．

但し，後に 2.2.3 節で述べるように，指数部が 0 (全桁が 0) と 2047 (全桁が 1) の場合は，特殊な数値に割り当てられているため，特殊な数値を除く数値 (正規化数) については，指数部が 1〜2046 の範囲，つまり，指数が −1022〜1023 乗まで (1024 乗未満) の範囲で数値を表現できる．

2.2.3 浮動小数点表現における特殊な値の表現ついて

浮動小数点表現では，零，非正規化数，無限大，非数の 4 種類の特殊な数値が存在し，それらの符号部・指数部・仮数部の組合せは，表 2.1 のようになっている．

表 2.1: 浮動小数点表現における特殊な数値

特殊な数値	省略表現	符号部	指数部	仮数部	種類と例
零（zero）	Zero	0 又は 1	$00\cdots0$	$00\cdots0$	±0 の 2 種類存在
非正規化数 (denormal number, subnormal number)	Denormal	0 又は 1	$00\cdots0$	$\neq 00\cdots0$	零と正規化数との間のギャップを埋める数 (float : $2^{-149} \leq \lvert r\rvert < 2^{-126}$ の数値) (double : $2^{-1074} \leq \lvert r\rvert < 2^{-1022}$ の数値)
無限大 (infty)	Inf	0 又は 1	$11\cdots1$	$00\cdots0$	$\pm\infty$ の 2 種類存在 (例：零割りの結果等)
非数 (not a number)	NaN	0 又は 1	$11\cdots1$	$\neq 00\cdots0$	シグナル発生あり・なしの NaN が存在 (例：負の数の平方根を取った結果等)

第3章　加速法

本章では，加速法の例としてエイトケン加速のプログラムを紹介する．この例では，エイトケン加速で $\pi/4$ の値の近似値を求めている．式 (1.8) の百万番目まで和 ($k = 0$〜1000000) を計算して得られる $\pi/4$ の近似値と同精度の値を，エイトケン加速ではたった 24 回の繰り返し計算で高速に求められることを示している．

3.1　エイトケン加速のプログラムの例

3.1.1　式 (1.8) を用いて $\pi/4$ を計算する Scilab 関数 "pi4.sci" (Aitken.sci から呼び出される関数)

```
// ==================================================================
//  pi4.sci: 式 (1.8) の第 n 項目までの和を返す Scilab 関数
// ==================================================================
function Sn = pi4(n);
  Sn = 0;
  for k = n:-1:0        // k = 0:n でも OK．計算精度を考えて和の順番を逆 (n から 0) にしている．
    Sn = Sn+((-1)^k)/(2*k+1)
  end
endfunction
```

3.1.2　エイトケン加速で $\pi/4$ を求める Scilab スクリプト "Aitken.sci"

```
// ----------------------------------------------------------------------
//  式 (1.8) の百万番目までの和と同精度の π/4 の値をエイトケン加速で計算
//
// ＜変数の説明＞
//  S: 式 (1.19) の計算結果 S_n^(k) (p.10 グレー枠) の値を保存する 2 次元配列
//  m: 配列 S の縦方向の添字 (m = n + 1) n = 0,1,2,…, m = 1,2,3,…
//  h: 配列 S の横方向の添字 (h = k + 1) k = 0,1,2,…, h = 1,2,3,…
//
// ※ Scilab では，配列の添字が 1 から始まる (C 言語では 0 から始まる).
//   式 (1.19) で，n,k は 0 から始まるが，0 は配列 S の添字として使えない.
//   そこで添字用の変数 m,h を用意し，S_n^(k) の結果を配列 S(m,h) に保存
// ----------------------------------------------------------------------
clear;                  // 定義済の変数の値を全て消去 (なくても OK)
stacksize(10000000);    // 使用可能なメモリのサイズを増やす (繰り返し回数が非常多いため必要)

itr = 1000000;              // 式 (1.8) で何番目まで和を計算するか (百万番目)
S_real = %pi/4;             // 式 (1.8) の理論値 π/4=0.7853981634…
exec('pi4.sci');            // 式 (1.8) の関数のファイル pi4.sci の読み込み
S_slow = pi4(itr);          // 式 (1.8) の百万番目までの和を計算
err1 = abs(S_real - S_slow); // 式 (1.8) の百万番目まで和と理論値との絶対誤差
                            // (↑エイトケンの繰返しの停止条件として利用)
for n=0:itr
   m = n + 1;               // m は配列 S の縦方向の添字 (n は 0 から，m は 1 から始まる)
   S(m,1) = pi4(n);         // p.9 グレー枠の一番左の列の S_n^(0) の値の計算
   err2 = abs(S(m,1) - S_real);
   for k = 0:((n/2)-1)      // p.9 グレー枠内を左下から右上へ向かって計算
      if err2 <= err1       // 誤差が小さくなったら終了 (break で内側の for 文から脱出)
         break
      end
      h = k + 1;            // h は配列 S の横方向の添字 (k は 0 から，h は 1 から始まる)
```

```
31        m = m - 1;              // 計算を行う位置を一段上の行へ移動する
32        S(m,h+1) = (S(m+1,h)*S(m-1,h)-S(m,h)^2)/(S(m+1,h)+S(m-1,h)-2*S(m,h));
33        err2 = abs(S(m,h+1) - S_real);  // ↑ エイトケン加速の計算式 (1.19) そのまま
34     end
35     if err2 <= err1            // 誤差が小さければ終了 (break で外側の for 文から脱出)
36        break
37     end
38  end
39
40  format(11);                   // 9 桁まで数値を表示するように変更 (9=11-2)
41  disp(S);                      // 配列 S の中身を表示
42
43  printf( "error1 = %.16f\n", err1 );              // 式 (1.8) の結果の理論値との誤差
44  printf( "error2 = %.16f\n", err2 );              // エイトケンの結果の理論値との誤差
45  printf( "S_real = %.16f\n", S_real );            // π/4 の理論値
46  printf( "S_slow = %.16f\n", S_slow );            // 式 (1.8) の百万番目までの和
47  printf( "S(%d,%d) = %.16f\n", m-1, h, S(m,h+1)); // エイトケン加速の結果
```

3.1.3 Aitken.sci の実行結果

以上のように，関数 pi4.sci と，スクリプト Aitken.sci を記述し，Scilab の exec コマンドで，もしくは，File メニューの Exec... から実行すると，次のようになる．

```
--> exec("Aitken.sci");              // ← exec コマンドを利用する場合
!   1.           0.           0.           0.         !
!   0.66666667   0.79166667   0.           0.         !
!   0.86666667   0.78333333   0.78552632   0.         !
!   0.72380952   0.78630952   0.78536255   0.78539984 !
!   0.83492063   0.78492063   0.78541083   0.78539772 !
!   0.74401154   0.78567821   0.78539282   0.78539831 !
!   0.82093462   0.78522034   0.78540071   0.         !
!   0.75426795   0.78551795   0.           0.         !
!   0.81309148   0.           0.           0.         !

error1 = 0.0000002499997500
error2 = 0.0000001444154701
S_real = 0.7853981633974483
S_slow = 0.7853984133971983
S(5,3) = 0.7853983078129184     // n = 5, k = 3 (たった 24 回の計算) で同精度の結果が得られている
```

第4章　連立一次方程式の数値的解法

4.1　ガウスの消去法 (枢軸選択あり) の Scilab 関数の例 "Gauss_pivot.sci"

以下に，ピボット (枢軸) 選択機能付きのガウスの消去法 (テキスト p.93) の関数の例とその使用例を示す.

```
// ==============================================================================
// Gauss_pivot.sci: ガウスの消去法 (ピボット選択を行う) で連立一次方程式を解く関数
// ------------------------------------------------------------------------------
//    入力 :  A : n 行 n 列 の係数行列
//            b : n 行 1 列 の定数ベクトル
//    出力:   x : n 行 1 列 の連立一次方程式の解のベクトル
// ==============================================================================
function x = Gauss_pivot(A, b)
[n, n1] = size(A);                  // 係数行列の縦横サイズ (n × n1) を調べる.
for i = 1:n-1                       // 係数行列の 1〜n-1 列目まで繰り返す.
   [m,k] = max(abs(A(i:n,i)));      // i 列目の i 行以降で係数の最大値を持つ行を探す.
                                    // k 行目が最大値を持つ行 (実際は i+k-1 行目)
   if k > 1                         // 行を入れ替える必要がある場合
      tmp1 = A(i,:);                // A の i 行目のベクトルを tmp1 に一時的に退避する.
      tmp2 = b(i);                  // b の i 行目の要素を tmp2 に一時的に退避する.
      A(i,:) = A(i+k-1,:);          // A の i 行目を最大係数を持つ i+k-1 行目と入れ替える.
      b(i)   = b(i+k-1);            // b の i 行目を最大係数を持つ i+k-1 行目と入れ替える.
      A(i+k-1,:) = tmp1;            // A の i+k-1 行目を退避してあった i 行目に置き換える.
      b(i+k-1)   = tmp2;            // b の i+k-1 行目を退避してあった i 行目に置き換える.
   end

   for h = i+1:n                     // i+1 行目以降に対して掃き出し法を適用する.
      r = A(h,i)/A(i,i);             // h 行目係数のが対角要素の何倍かを求める.
      A(h,:) = A(h,:) - r * A(i,:); // 掃き出し法で A の h 行目を更新する. (i 列目を 0 にする)
      b(h)   = b(h)   - r * b(i);   // 掃き出し法で b の h 行目を更新する.
   end;
end;                                 // この時点で，掃き出し法の上三角行列ができ上がる.

x = zeros(n,1);                      // 解のベクトル x の要素を全て 0 で初期化する.
x(n) = b(n) / A(n,n);                // まず一番下の行 (n 行目) から n 番目の解 x(n) を計算する.

for i = n-1:-1:1                     // 上三角行列の下の方から順番に解を求めていく.
   x(i) = (b(i) - A(i,i+1:n) * x(i+1:n,1)) / A(i,i);
end                                  // ↑上三角行列 i 行目で i 番目の解 x(i) を計算する.
endfunction
```

```
--> exec("Gauss_pivot.sci");
--> A = [1 100 3; 1 3 4; 50 1 3];
--> b = [4 3 2]'
--> x = Gauss_pivot(A,b) // ← 関数 Gauss_pivot で 連立一次方程式 Ax=b を解く.
 x  =
  - 0.00462146
    0.01791463
    0.73771939

--> x = A\b       // ← Gauss_pivot で求めた解が正しいか確認. 演算子 \ で Ax=b の x を求められる.
 x  =
  - 0.00462146
    0.01791463
    0.73771939
```

4.2 連立一次方程式を反復法で解く例題

ここでは，連立一次方程式を反復法で数値的に解く例題として，次の係数行列 $\boldsymbol{A}$，定数ベクトル $\boldsymbol{b}$ で表される連立一次方程式をヤコビ法およびガウス・ザイデル法に基づいて解く．

$$\begin{cases} 5x_1 - x_2 = 9 \\ -x_1 + 5x_2 - x_3 = 4 \\ -x_2 + 5x_3 = -6 \end{cases} \qquad \boldsymbol{A} = \begin{bmatrix} 5 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 5 \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 9 \\ 4 \\ -6 \end{bmatrix}$$

但し，解の初期値は $\boldsymbol{x}^{(0)} = [0 \ \ 0 \ \ 0]'$ とする．

4.2.1 ヤコビ法による解法

連立方程式を書き換えると次のようになる．

$$\begin{cases} x_1 = x_2/5 + 9/5 \\ x_2 = x_1/5 + x_3/5 + 4/5 \\ x_3 = x_2/5 - 6/5 \end{cases} \qquad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 & 1/5 & 0 \\ 1/5 & 0 & 1/5 \\ 0 & 1/5 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix}$$

ここで，

$$\boldsymbol{C} = \begin{bmatrix} 0 & 1/5 & 0 \\ 1/5 & 0 & 1/5 \\ 0 & 1/5 & 0 \end{bmatrix}, \quad \boldsymbol{r} = \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix}$$

とおくと，ヤコビ法の反復式は次のように表せる．

$$\boldsymbol{x}^{(n)} = \boldsymbol{C}\boldsymbol{x}^{(n-1)} + \boldsymbol{r}$$

但し，$\boldsymbol{x}^{(n)}$ は n 回目の繰り返しにおける解である．この反復式に基づいて $\boldsymbol{x}^{(n)}$ を繰り返し求めればよい．

$$\begin{aligned}
\boldsymbol{x}^{(0)} &= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\
\boldsymbol{x}^{(1)} &= \begin{bmatrix} 0 & 1/5 & 0 \\ 1/5 & 0 & 1/5 \\ 0 & 1/5 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix} = \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix} = \begin{bmatrix} 1.8 \\ 0.8 \\ -1.2 \end{bmatrix} \\
\boldsymbol{x}^{(2)} &= \begin{bmatrix} 0 & 1/5 & 0 \\ 1/5 & 0 & 1/5 \\ 0 & 1/5 & 0 \end{bmatrix} \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix} + \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix} = \begin{bmatrix} 49/25 \\ 23/25 \\ -24/25 \end{bmatrix} = \begin{bmatrix} 1.96 \\ 0.92 \\ -0.96 \end{bmatrix} \\
\boldsymbol{x}^{(3)} &= \begin{bmatrix} 0 & 1/5 & 0 \\ 1/5 & 0 & 1/5 \\ 0 & 1/5 & 0 \end{bmatrix} \begin{bmatrix} 49/25 \\ 23/25 \\ -24/25 \end{bmatrix} + \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix} = \begin{bmatrix} 248/125 \\ 1 \\ -128/125 \end{bmatrix} = \begin{bmatrix} 1.984 \\ 1 \\ -1.024 \end{bmatrix} \\
\boldsymbol{x}^{(4)} &= \begin{bmatrix} 0 & 1/5 & 0 \\ 1/5 & 0 & 1/5 \\ 0 & 1/5 & 0 \end{bmatrix} \begin{bmatrix} 248/125 \\ 1 \\ -128/125 \end{bmatrix} + \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix} = \begin{bmatrix} 2 \\ 124/125 \\ -1 \end{bmatrix} = \begin{bmatrix} 2 \\ 0.992 \\ -1 \end{bmatrix} \\
&\vdots
\end{aligned}$$

繰り返しによって，連立方程式の解の理論値 $\boldsymbol{x} = [2 \ \ 1 \ \ -1]'$ に近付いて行く．

4.2.2 ガウス・ザイデル法による解法

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{c}_1 \\ \boldsymbol{c}_2 \\ \boldsymbol{c}_3 \end{bmatrix} = \begin{bmatrix} 0 & 1/5 & 0 \\ 1/5 & 0 & 1/5 \\ 0 & 1/5 & 0 \end{bmatrix}, \quad \boldsymbol{r} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 9/5 \\ 4/5 \\ -6/5 \end{bmatrix}$$

とおくと，ガウス・ザイデル法の n 回目の反復における解 $\boldsymbol{x}^{(n)} = \begin{bmatrix} x_1^{(n)} & x_2^{(n)} & x_3^{(n)} \end{bmatrix}'$ は，次式で表せる．

$$\begin{aligned} x_1^{(n)} &= \boldsymbol{c}_1 \begin{bmatrix} x_1^{(n-1)} & x_2^{(n-1)} & x_3^{(n-1)} \end{bmatrix}' + r_1 \\ x_2^{(n)} &= \boldsymbol{c}_2 \begin{bmatrix} x_1^{(n)} & x_2^{(n-1)} & x_3^{(n-1)} \end{bmatrix}' + r_2 \\ x_3^{(n)} &= \boldsymbol{c}_3 \begin{bmatrix} x_1^{(n)} & x_2^{(n)} & x_3^{(n-1)} \end{bmatrix}' + r_3 \end{aligned}$$

初期値 $\boldsymbol{x}^{(0)} = [\,0 \;\; 0 \;\; 0\,]'$ から開始して，繰り返し解を求めると次のようになる．

$$\begin{aligned} x_1^{(1)} &= [\;0 \;\; 1/5 \;\; 0\;]\,[\;0 \;\; 0 \;\; 0\;]' + 9/5 = 9/5 = 1.8 \\ x_2^{(1)} &= [\,1/5 \;\; 0 \;\; 1/5\,]\,[\,9/5 \;\; 0 \;\; 0\,]' + 4/5 = 29/25 = 1.16 \\ x_3^{(1)} &= [\;0 \;\; 1/5 \;\; 0\;]\,[\,9/5 \;\; 29/25 \;\; 0\,]' - 6/5 = -121/125 = -0.968 \end{aligned}$$

$$\begin{aligned} x_1^{(2)} &= [\;0 \;\; 1/5 \;\; 0\;]\,[\;9/5 \;\; 29/25 \;\; -121/125\,]' + 9/5 = 254/125 = 2.032 \\ x_2^{(2)} &= [\,1/5 \;\; 0 \;\; 1/5\,]\,[\,254/125 \;\; 29/25 \;\; -121/125\,]' + 4/5 = 633/625 = 1.0128 \\ x_3^{(2)} &= [\;0 \;\; 1/5 \;\; 0\;]\,[\,254/125 \;\; 633/625 \;\; -121/125\,]' - 6/5 = -3117/3125 = -0.99744 \end{aligned}$$

$$\begin{aligned} x_1^{(3)} &= [\;0 \;\; 1/5 \;\; 0\;]\,[\;254/125 \;\; 633/625 \;\; -3117/3125\,]' + 9/5 = 6258/3125 = 2.00256 \\ x_2^{(3)} &= [\,1/5 \;\; 0 \;\; 1/5\,]\,[\,6258/3125 \;\; 633/625 \;\; -3117/3125\,]' + 4/5 = 15641/15625 = 1.001024 \\ x_3^{(3)} &= [\;0 \;\; 1/5 \;\; 0\;]\,[\,6258/3125 \;\; 15641/15625 \;\; -3117/3125\,]' - 6/5 = -78109/78125 = -0.9997952 \end{aligned}$$

$$\vdots$$

ヤコビ法よりも，厳密解 $\boldsymbol{x} = [2 \;\; 1 \;\; -1]'$ に早く近付く．

4.3 ガウス・ザイデル法の Scilab プログラム

4.3.1 ガウス・ザイデル法の Scilab 関数の例 "Seidel.sci"

```
// ==============================================================================
// Seidel.sci:  ガウス・ザイデル法で連立一次方程式 Ax=b を解く関数
// ------------------------------------------------------------------------------
// 入力  A:    係数行列 A (n × n)
//       b:    右辺 b の縦ベクトル (n × 1)
//       x0:   解の初期値の縦ベクトル (n × 1)
//       tol:  ノルムの変化の 2 乗の値の許容範囲 (収束判定に利用)
//       itr:  最大繰返し回数
// 出力  x:    解 x の縦ベクトル (n × 1)
// ==============================================================================
// C: 式 (5.41) の係数 a_ij/a_ii の値を入れる行列 (右辺第 1,2 項目の係数÷対角成分)
// r: 式 (5.41) の係数 b_i/a_ii の値を入れる縦ベクトル (右辺第 3 項目の係数÷対角成分)
// S: 解の途中結果を保存するための行列

function x = Seidel(A, b, x0, tol, itr)
[n, m] = size(A);                // 行列の縦横サイズをチェック
C = -A;                          // C = -A (行列 A を右辺へ移項)
for i = 1:n
  C(i,i) = 0;                    // 行列 C の対角成分を全て 0 に
```

```
  C(i,1:n) = C(i,1:n) / A(i,i); // 行列 C の各行の要素を A の対角成分で割る
  r(i) = b(i) / A(i,i);         // ベクトル b の要素を A の対角成分で割ったもの
end

S = x0;                         // 解の初期値を行列 S の一番左の列に保存
x = x0;                         // 解の初期値を x に代入
for k = 1:itr                   // 繰返し回数の最大値 itr 回まで繰り返す
  for i = 1:n                   // i=1,2,…の順に i 番目の解 x(i) を求める
    x(i) = C(i,:) * x + r(i);   // x(i) の結果が次の解 x(i+1) に反映される
  end
  S(:,k+1) = x;                 // k 回目繰返しにおける解を行列 S の右側に追加
  if norm(S(:,k) - x)^2 < tol   // 前の解との差のノルムの 2 乗を見て収束判定
    break;
  end
end
format(7);                      // 解の途中経過を 5 桁までの表示に変更
disp(S');                       // 途中経過を表示 (S を転置して表示)

endfunction
```

4.3.2 Seidel 関数を利用して例題 5 の方程式を解く Scilab スクリプトの例 "ex5_Seidel.sci"

```
// ------------------------------------------------------------
//  Seidel 関数の使用例 (例題 5 の連立方程式 Ax=b の解:表 5.2 に同じ)
// ------------------------------------------------------------
clear;                          // 定義済みの変数を消去
exec('Seidel.sci');             // Seidel 関数の読み込み
A  = [ 10, 3, 1, 2, 1;          // Ax=b の係数行列 A の定義
        1,19, 2,-1, 5;
       -1, 1,30, 1,10;
       -2, 0, 1,20, 5;
       -3, 5, 1,-2,25 ];
b   = [-22, 27, 89,-73, 22]';   // Ax=b の縦ベクトル b の定義
x0  = [  0,  0,  0,  0,  0]';   // 解の初期値
tol = 1.0e-8;                   // x の差のノルムの 2 乗の許容量
itr = 100;                      // 最大繰り返し回数
x = Seidel( A, b, x0, tol, itr ) // Seidel 関数の実行
                                // ↑ セミコロン (;) をつけていないので解 x が表示される
```

4.3.3 Scilab スクリプト"ex5_Seidel.sci"の実行結果

```
--> exec('ex5_Seidel.sci');
    0.        0.        0.        0.        0.      // 各繰り返しにおける数値解 x の途中経過が
  - 2.2       1.5368    2.8421  - 4.0121  - 0.1260  // Seidel 関数の最終行 disp(S') で表示される
  - 2.1302    1.056     3.0362  - 3.9833  - 0.0269  // ← テキストの表 5.2 の結果に同じ
  - 2.0211    1.0053    3.0075  - 3.9957  - 0.0035
  - 2.0028    1.0005    3.0009  - 3.9994  - 0.0004
  - 2.0003    1.0001    3.0001  - 3.9999  - 5.D-05
  - 2.        1.        3.      - 4.      - 6.D-06
  - 2.        1.        3.      - 4.      - 7.D-07

 x  =                                                // 繰り返しの最終回における数値解 x の表示
  - 2.                                               // (上記の表の最後の行に同じ)
    1.
    3.
  - 4.
  - 7.D-07
```

第5章　関数近似 (ラグランジュ補間)

5.1　多項式や有理式を扱う Scilab 関数の紹介

多項式や有理式[1]を扱うための Scilab の関数に，poly, horner, derivat などがある．ここでは，次の多項式 y と有理式 z を例にして説明する．

$$
\begin{aligned}
y &= (x-1)(x-2) \\
&= x^2 - 3x + 2 \\
z &= \frac{x^2}{(x-1)(x-5)(x+3)} \\
&= \frac{x^2}{x^3 - 3x^2 - 13x + 15}
\end{aligned}
$$

5.1.1　poly 関数による多項式や有理式の定義

根 (roots) を指定して多項式や有理式を表現する方法

多項式 $y = (x-1)(x-2)$ を，その根によって定義するには，poly 関数の第 1 引数に多項式の根を並べたベクトル [1 2]，第 2 引数に多項式の変数として使う記号 (ここでは 'x')，第 3 引数に 'roots' を指定すればよい．但し，第 3 引数は，デフォルトが 'roots' であるため省略できる．

```
--> y = poly([1 2], 'x')          // 多項式 y の定義 (多項式の根を指定)
 y  =                             // y = poly([1, 2], 'x', 'roots') に同じ ('roots' は省略可)
              2
    2 - 3x + x

--> roots(y)                     // roots は多項式の根を返す関数
 ans  =
    1.                           // 1 と 2 が多項式 y の根
    2.

--> z = poly([0 0],'x') / poly([1 5 -3],'x')  // 有理式 z の定義 (分子と分母の根を指定)
 z  =
             2
            x
    -----------------
                2   3
    15 - 13x - 3x + x
```

また，定義した多項式や有理式を用いて，新たな多項式や有理式を定義することもできる．

```
--> s = y^2 - 1                    // y^2 - 1 = {(x-1)(x-2)}^2 - 1 に同じ
 s =
                2    3   4
    3 - 12x + 13x - 6x + x

--> y * z                          // 多項式 y と有理式 z の積の有理式
 ans  =
        2   3
    - 2x + x
    ----------
                2
  - 15 - 2x + x
```

[1]有理式とは，分母と分子が多項式の分数．

係数 (coefficients) を指定して多項式を表現する方法

多項式 $y = x^2 - 3x + 2$ を各次の係数を指定して表現するには，第 1 引数に多項式の係数を低い次数から順に並べたベクトル $[2\ \ -3\ \ 1]$, 第 2 引数に多項式の変数として使う記号 (ここでは 'x')，第 3 引数に 'coeff' を指定する．

```
--> y = poly([2 -3 1],'x','coeff')      // 多項式 y の定義．(注) 係数を並べる順番は低い次数から
 y  =
              2
    2 - 3x + x

--> coeff(y)                 // 多項式 y の係数を表示．  coeff は多項式の係数のベクトルを返す関数
 ans  =
    2.  - 3.    1.


--> z = poly([0 0 1],'x','coeff') / poly([15 -13 -3 1],'x','coeff')     // 有理式 z の定義
 z  =
             2
            x
    -----------------
                 2   3
    15 - 13x - 3x + x

--> coeff(numer(z))       // 有理式 z の分子の多項式の係数を表示．  numer は分子の多項式を返す関数
 ans  =
    0.    0.    1.

--> coeff(denom(z))       // 有理式 z の分母の多項式の係数を表示．  denom は分母の多項式を返す関数
 ans  =
    15.  - 13.  - 3.    1.
```

多項式を直接記述する方法

根や係数のベクトルを指定する他に，次のように多項式を直接記述することもできる．

```
--> x = poly(0, 'x');                          // x を多項式の変数として用いることを宣言
--> y = x^2 - 3*x + 2                          // 多項式 y の定義
 y  =
              2
    2 - 3x + x

--> x = poly(0, 'x');                          // x を多項式の変数として用いることを宣言
--> z = x^2 / (x^3-3*x^2-13*x+15)              // 有理式 z の定義
 z  =
             2
            x
    -----------------
                 2   3
    15 - 13x - 3x + x
```

5.1.2 horner 関数で多項式や有理式に値を代入する方法

多項式や有理式にある値を代入するには，horner(ホーナー) 関数を用いる．

```
--> horner(y, 5)                       // y = x^2-3x+2 の x に 5 を代入
 ans  =
    12.

--> horner(z,-2)                       // z = x^2/(x^3-3*x^2-13*x+15) の x に-2 を代入
 ans  =
    0.1904762
```

5.1.3 derivat 関数による多項式や有理式の微分

derivat 関数を用いて，多項式や有理式を微分することができる．

```
--> dy = (derivat(y))                          // 多項式 y の微分
 dy  =
  - 3 + 2x

--> ddy = derivat(dy)                          // 多項式 y の 2 階微分
 ddy  =
    2

--> dz = derivat(z)                            // 有理式 z の微分
 dz =
                          2   4
                30x - 13x - x
    -------------------------------------
                    2      3     4    5   6
    225 - 390x + 79x + 108x - 17x - 6x + x
```

5.2 deff 関数による関数定義と feval 関数による関数への値の代入

これまで，関数を定義するには，関数の定義を 1 つのファイルに記述して，そのファイルを exec 関数で読み込んでいたが，簡単な関数であれば，deff 関数を用いて定義するのが便利である．また，定義した関数に値を代入するための関数として feval がある．horner は多項式の変数にある値を代入した結果を返すが，これと同様に，feval も関数にある値を代入した結果を返す．

```
--> deff('y = f(x)','y = x*cos(x)');  // 関数 f の定義
--> f(2)                              // 関数 f に 2 を代入した値
 ans  =
  - 0.8322937
--> feval(2, f)                       // f(2) に同じ
 ans  =
  - 0.8322937

--> deff('y = f(x)','y = x^2-3*x+2'); // 関数 f の定義
--> a = f(5)                          // 関数 f に 5 を代入した値
 ans  =
    12.
--> a = feval(5, f)                   // f(5) に同じ
 a  =
    12.
```

5.3 ラグランジュ補間のプログラム

5.3.1 ラグランジュ補間の多項式を返す関数定義の例 "Lagrange.sci"

```
// ==================================================================
//  Lagrange.sci: ラグランジュの補間多項式を求める関数
// ------------------------------------------------------------------
//  引数     x: 補間点の x 軸の値の系列
//           y: 補間点の y 軸の値の系列
//  返り値   f: 補間多項式
// ==================================================================
function f = Lagrange(x, y)
f = 0;                                          // 多項式 f を 0 で初期化
for i = 1:length(x)                             // length(x) は x の要素数 (補間点の数)
   xi = [x(1:i-1), x(i+1:$)];                   // xi は x から i 番目の点を除いたベクトル
   f = f + poly(xi, 'x') * y(i) / prod( x(i) - xi ); // p.17 式 (2.11) の各項を順に加える
end
endfunction
```

5.3.2 関数 Lagrange を用いて例題 1 (p.20) を解くスクリプトの例 ”ex1_Lagrange.sci”

```
// ----------------------------------------------------------------------------
//  テキスト p.20 例題 1
//  sin(x) 上の 5 つの補間点から 4 次のラグランジュ補間多項式を求め,
//  sin(x) のグラフを重ねてグラフに表示するスクリプト
// ----------------------------------------------------------------------------
clear;                                  // 定義済みの変数の値を全て消去
exec('Lagrange.sci');                   // Lagrange 関数の読込み

x = [0, %pi/4, %pi/2, 3*%pi/4, %pi];    // 補間点の x 軸の値の系列
y = [0, sqrt(2)/2, 1, sqrt(2)/2, 0];    // 補間点の y 軸の値の系列
f = Lagrange(x, y)                      // Lagrange 関数でラグランジュ補間多項式を求める

xx = x(1):0.001:x($);             // 0〜π の間を 0.001 刻みで分割した系列 (グラフの x 軸の刻み)
yy = horner(f, xx);               // x 軸の系列 xx での補間多項式の値 (xx と同じ長さの系列)
y_real = sin(xx);                 // x 軸の系列 xx での理論値 sin(x) の値 (xx と同じ長さの系列)
plot2d(xx', [yy', y_real']);      // 補間多項式と sin(x) を重ねて表示 (ほぼぴったり重なる)
```

5.3.3 ”ex1_Lagrange.sci” の実行結果

```
--> exec('ex1_Lagrange.sci');
 f  =
                                2            3            4
    0.9819684x + 0.0582877x - 0.2360955x + 0.0375758x     // 得られた 4 次の補間多項式
```

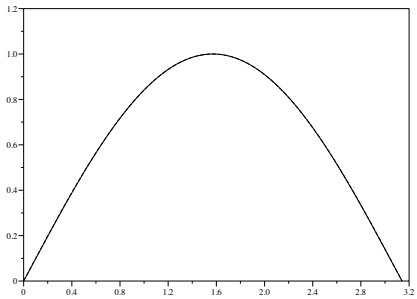

図 5.1: 例題 1 (sin 関数の Lagrange 補間) を解くスクリプト”ex1_Lagrange.sci”の実行結果のグラフ

5.3.4 関数 Lagrange を用いて例題 2 (p.24) を解くスクリプトの例 ”ex2_Lagrange.sci”

```
// ----------------------------------------------------------------------------
//  テキスト p.24 例題 2
//  4 つの補間点から 3 次のラグランジュ補間多項式を求め, 3√20 の推定値を求めるスクリプト
// ----------------------------------------------------------------------------
clear;                                      // 定義済みの変数の値を全て消去
exec('Lagrange.sci');                       // Lagrange 関数の読込み

x = [19.9466, 19.9907, 20.0349, 20.0793]; // 補間点の x 軸の値の系列
y = [  2.712,   2.714,   2.716,   2.718]; // 補間点の y 軸の値の系列
f = Lagrange(x, y)                          // Lagrange 関数でラグランジュ補間多項式 f を求める
yy = horner(f, 20);                         // 補間多項式 f に 20 を代入
printf("f(20) = %f\n", yy);                 // 3√20 の推定値 f(20) の値の表示
```

5.3.5 ”ex2_Lagrange.sci” の実行結果

```
--> exec("ex2_Lagrange.sci");
 f  =
                                    2            3
    69.883617 - 10.193976x + 0.5133640x - 0.0085794x      //  得られた 3 次の補間多項式

  f(20) = 2.714421                                        // 多項式から求めた 3√20 の推定値
```

第6章　数値積分

6.1　台形公式による数値積分

6.1.1　台形公式による数値積分の値を返す関数の例 ”trapezoid.sci”

```
// ==================================================================
//  trapezoid.sci: 台形公式による数値積分を行う関数
// ------------------------------------------------------------------
//  ＜引数＞   f: 被積分関数
//              a: 積分区間の始まり
//              b: 積分区間の終わり
//              N: 区間 [a,b] での小区間の数
//  ＜返り値＞ I: 積分値
// ==================================================================
function I = trapezoid(f, a, b, N)
h = (b - a) / N;                                  // 積分の小区間の幅
x = a:h:b;                                        // 補間点における x 軸の値
y = f(x);                                         // 補間点における y 軸の値
I = h * (y(1) + 2 * sum(y(2:N)) + y(N+1)) / 2;   // 式 (3.4) で表される積分の値
endfunction
```

6.1.2　式 (3.6) の数値積分の値を求めるスクリプトの例 ”eq36_trapezoid.sci”

```
clear;
exec('trapezoid.sci');          // 台形公式の関数ファイルの読み込み
deff('y=f(x)','y=sin(x)');      // 被積分関数 f の定義 f(x)=sin(x)
a = 0;                          // 積分区間の始まり
b = %pi/2;                      // 積分区間の終わり
N = 8;                          // 分割数 (台形の数)
format(11);                     // 結果を 9 桁まで表示
I = trapezoid(f,a,b,N)          // 台形公式による数値積分．式 (3.6) の結果に同じ
```

6.1.3　スクリプト ”eq36_trapezoid.sci” の実行結果

```
--> exec('eq36_trapezoid.sci');
 I  =
    0.99678517                      // 台形公式により求めた積分値
```

6.1.4　台形公式による数値積分を行う Scilab の標準関数

上記で定義した関数と同様に，台形公式による数値積分を行う Scilab の標準関数として inttrap がある．以下に，inttrap 関数の使用例を示す．(詳細については，help inttrap でヘルプを参照のこと．)

```
--> h = %pi/16;                     // 各区間の幅 h = π/16
--> x = 0:h:%pi/2;                  // 0〜π/2 まで π/16 刻み
--> I = inttrap(x, sin(x))          // 台形公式による数値積分
 I  =
    0.99678517                      // 上記のスクリプトと同じ結果
```

また，自分で定義した関数について積分する場合は，次のようにする．

```
--> h = %pi/16;
--> x = 0:h:%pi/2;
--> deff('y=f(x)','y=sin(x)');      // deff で関数 f(x) を定義
--> I = inttrap(x, f(x))            // 第 2 引数を関数 f(x) とする
 I  =
    0.99678517
```

6.2 ガウス・ルジャンドルの積分公式による数値積分 (ガウスの求積法)

6.2.1 ガウスの求積法による数値積分の値を返す関数の例 "GaussLegendre.sci"

```
// ==================================================================
//  GaussLegendre.sci: ガウスの求積法による数値積分を行う関数
// ------------------------------------------------------------------
//  ＜引数＞   f: 被積分関数
//              a: 積分区間の始まり
//              b: 積分区間の終わり
//              N: 補間点数 (但し, N は 2,3,4,5 の場合にのみ対応)
//  ＜返り値＞ I: 積分値
// ==================================================================
function I = GaussLegendre(f, a, b, N)
xi = [-sqrt(1/3)  -sqrt(3/5)  -0.8611363116  -0.9061798459  // p.53 の表 3.3 の補間点
       sqrt(1/3)          0    0.3399810436  -0.5384693101
              0    sqrt(3/5)   0.3399810436              0
              0           0    0.8611363116   0.5384693101
              0           0               0   0.9061798459];

wi = [        1         5/9    0.3478548451   0.2369268851  // p.53 の表 3.3 の重み
              1         8/9    0.6521451549   0.4786286705
              0         5/9    0.6521451549   0.5688888889
              0           0    0.3478548451   0.4786286705
              0           0               0   0.2369268851];

x = xi(1:N, N-1);                  // [1,-1] の区間での「補間点」の行列 xi から必要な部分を抽出
x = ((b - a) * x + b + a) / 2;     // [a,b] の区間での「補間点」に変換. p.51 の ξ の逆変数変換
y = f(x);                          // y = [f(x_1), f(x_2),..., f(x_N)]
w = wi(1:N, N-1);                  // 「重み」の行列 wi から必要な部分を抽出
I = (b - a) / 2 * (w' * y);        // 式 (3.20) を (b-a)/2 倍して [a,b] の区間に変換
endfunction
```

6.2.2 表 3.4 のガウスの求積法による積分値を求めるスクリプトの例 "tab34_GaussLegendre.sci"

```
clear;
exec('GaussLegendre.sci');                  // ガウスの求積法の関数ファイルの読み込み
deff('y=f(x)','y=(x.*(1-x)).^(3/2)');       // 関数 f の定義 f(x)= {x(1-x)}^(3/2)
a = 0;                                      // 積分区間の始まり
b = 1;                                      // 積分区間の終わり
for N = 2 : 5                               // 補間点数 N = 2,3,4,5 の場合について順に計算
   S = (128/3) * GaussLegendre(f,a,b,N);    // ガウスの求積法による数値積分
   disp(S);                                 // ↑ 表 (3.4) の上から 4 行目まで (N=2〜5) を計算
end
```

6.2.3 スクリプト "tab34_GaussLegendre.sci" の実行結果

```
--> exec('tab34_GaussLegendre.sci');

   2.903099                // N = 2 の場合  (p.54 表 3.4 の一番右の列の N=2〜5 の場合と同じ結果)
   3.1199473               // N = 3 の場合
   3.1365993               // N = 4 の場合
   3.1399284               // N = 5 の場合
```

6.2.4 ガウスの求積法による数値積分を行う Scilab の標準関数

ガウスの求積法で数値積分を行う Scilab の標準関数として integrate がある．以下に，integrate 関数の使用例を示す．(詳細については，help integrate でヘルプを参照のこと．)

```
--> deff('y=f(x)','y=(x.*(1-x)).^(3/2)');
--> S = (128/3) * integrate('f(x)', 'x', 0, 1)
 S =
   3.1415927               // p.54 表 3.4 の N = 100 以上と同等の精度
```

第7章　非線形方程式の数値的解法

7.1　ニュートン法 (ニュートン・ラフソン法) による非線形方程式の数値的解法

7.1.1　ニュートン法による方程式の解を返す関数の例 "Newton.sci"

```
// ==========================================================================
//  Newton.sci: ニュートン法 (テキスト p.72 の式 (4.16)) で非線形方程式を解く関数
// --------------------------------------------------------------------------
//  f: 関数 f(x),  df: 導関数 f'(x),  x0: 解の初期値
//  tol: 絶対誤差の許容範囲,  itr: 最大繰り返し回数
// ==========================================================================
function x = Newton(f, df, x0, tol, itr)
x(1)  = x0;                          // x の初期値 x0 を x(1) とする
y(1)  = f( x(1) );                   // x0 に対する f(x0) の値を y(1) とする
dy(1) = df( x(1) );                  // x0 に対する f'(x0) の値を dy(1) とする

for k = 1:itr                        // 最大で itr 回繰り返す
   x(k+1)  = x(k) - y(k) / dy(k);    // 式 (4.16) で x の値を更新
   y(k+1)  = f( x(k+1) );            // 更新した x に対する f(x) の値
   dy(k+1) = df( x(k+1) );           // 更新した x に対する f'(x) の値
   if ( abs( y(k+1) ) < tol ) then   // 式 (4.17) で収束判定
      break;                         // 収束したら for 文から抜ける
   end
end
format(12)                           // 値を 10 桁まで表示する
disp([ [0:k]', x, y ])               // ニュートン法の繰り返し回数と x, y の値を表示
endfunction
```

7.1.2　ニュートン法で p.72 の例題 2 の方程式を解くスクリプトの例 "ex2_Newton.sci"

方程式 $f(x) = x^3 + 6x^2 + 21x + 32 = 0$ をニュートン法で解くには，次のようなスクリプトを実行すればよい．

```
clear;                                // 定義済みの変数の値の消去
deff('y=f(x)','y=x^3+6*x^2+21*x+32'); // f(x) の定義
deff('y=df(x)','y=3*x^2+12*x+21');    // f'(x) の定義
exec('Newton.sci');                   // ニュートン法の関数の読み込み
Newton(f,df,0,1.0e-10,20);            // 初期値 0，許容誤差 1.0e-10，最大繰返し回数 20 で解く
```

7.1.3　スクリプト "ex2_Newton.sci" の実行結果

```
--> exec('ex2_Newton.sci');
    0.    0.             32.              // 初期値 x = 0, y = f(0) = 32
    1.  - 1.523809524    10.39369399      // 繰り返し 1 回目
    2.  - 2.597508059    0.409107601      // 繰り返し 2 回目
    3.  - 2.638130209  - 0.003024983      // 繰り返し 3 回目
    4.  - 2.637834269  - 0.000000168      // 繰り返し 4 回目
    5.  - 2.637834253    3.55271E-15      // 繰り返し 5 回目で収束．解は x = -2.637834253
```

7.2　方程式の解を求める Scilab の標準関数 roots

多項式 $f(x) = 0$ の解を求める Scilab の標準関数として roots がある．以下に，roots 関数の使用例を示す．

```
--> y = poly([32,21,6,1],'x','coeff');     // 多項式 f(x)=x^3+6x^2+21x+32 の定義
--> roots(y)                               // f(x)=0 の解を roots 関数で求める
 ans  =
  - 2.637834253                            // 解 1 (上記の例で求めた解に同じ)
  - 1.681082874 + 3.050430199i             // 解 2 (複素解)
  - 1.681082874 - 3.050430199i             // 解 3 (複素解)
```

第8章　微分方程式の数値的解法

8.1　4次のルンゲ・クッタ法による微分方程式の数値的解法

8.1.1　4次のルンゲ・クッタ法で微分方程式を解く関数 "RungeKutta.sci"

```
// ==================================================================
// RungeKutta.sci: 4次のルンゲ・クッタ法 (6.42) で微分方程式を解く関数
// ------------------------------------------------------------------
//  <引数>      f: 微分方程式の関数 f(x,y)
//             a,b: 区間 [a,b]
//              y0: 初期条件
//               N: 区間の分割数
// <返り値>  x,y: 微分方程式の数値解 (x:ベクトル, y:ベクトル)
// ==================================================================
function [x,y] = RungeKutta(f, a, b, y0, N)
h = (b - a)/N;                                      // h : 区間の刻み幅
x = a:h:b;                                          // x 軸の刻み (縦ベクトル)
y = y0 ;                                            // 初期条件 y=y0
for i = 1 : N                                       // N 点で解を求める
   k1 = h * f( x(i),       y(i)        );           // 式 (6.42) の k1
   k2 = h * f( x(i) + h/2, y(i) + k1/2 );           // 式 (6.42) の k2
   k3 = h * f( x(i) + h/2, y(i) + k2/2 );           // 式 (6.42) の k3
   k4 = h * f( x(i) + h,   y(i) + k3   );           // 式 (6.42) の k4
   y(i+1) = y(i) + (k1 + 2*k2 + 2*k3 + k4) / 6;     // 式 (6.42) の y(i+1)
end
endfunction
```

8.1.2　4次のルンゲ･クッタ法でp.141の例題3を解くスクリプトの例 "ex3_RungeKutta.sci"

```
// ---------------------------------------------------------------------------
//  関数 RungeKutta を用いて例題 3 (p.141) を解くスクリプト
// ---------------------------------------------------------------------------
clear;                                                      // 定義済みの変数の値の消去
exec('RungeKutta.sci');                                     // 関数の読み込み
deff('dy=f(x,y)','dy=y-y^2');                               // 例題 3 の微分方程式の関数の定義
a = 0;                                                      // 区間の始まり
b = 20;                                                     // 区間の終わり
y0 = 0.1;                                                   // 初期条件
N =  8; [x,y] = RungeKutta(f,a,b,y0,N); plot2d(x,y,1);      // N = 8  の場合
N = 10; [x,y] = RungeKutta(f,a,b,y0,N); plot2d(x,y,2);      // N = 10 の場合
N = 20; [x,y] = RungeKutta(f,a,b,y0,N); plot2d(x,y,3);      // N = 20 の場合
```

exec('ex3_RungeKutta.sci') で，上記のスクリプトを実行すると，図 8.1 に示す微分方程式の数値解が表示される．

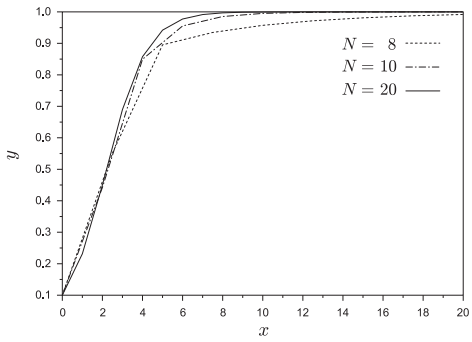

図 8.1: 例題 3 (p.141) を 4 次のルンゲ・クッタ法で解いた様子 ($N = 8, 10, 20$)

8.2 Scilab の標準関数 ode の使い方

Scilab の標準関数に，常微分方程式 (ODE; Ordinary Differential Equation) を数値的に解く関数 ode がある．以下に，ode 関数を用いた 1 階の微分方程式，n 階の微分方程式，連立 1 階微分方程式の解法の例を示す．

8.2.1 ode 関数による 1 階微分方程式の解法

ode 関数を用いて，次のような微分方程式の初期値問題を解く方法について説明する．

$$y' = f(x, y), \quad \text{初期条件}: y(x_0) = y_0 \tag{8.1}$$

1 階微分方程式 $y' = x + y,\ y(0) = 0$ の例

ここでは，例として，$f(x, y) = x + y$, $x_0 = 0$, $y_0 = 0$，すなわち次式の微分方程式を解く．

$$y' = x + y, \quad y(0) = 0 \tag{8.2}$$

ode 関数の 4 つの引数は，左から (1) y_0 の値，(2) x_0 の値，(3) 数値解を求める区間の x 軸の刻みの系列 x_0〜x_N の横ベクトル，(4) 関数名の順で，例えば，次のように利用する．

```
--> deff('yd=f(x,y)','yd=x+y');       // f(x,y)=x+y の定義
--> x0=0; y0=0;                       // 初期条件 y(0)=0 の設定
--> x=x0:0.2:1;                       // x=[0,1] の区間において刻み幅 0.2 で解く (x=[0,0.2,0.4,..,1])
--> y=ode(y0,x0,x,f)                  // ode 関数で微分方程式を解く
 y  =
   0.   0.0214028   0.0918247   0.2221188   0.4255409   0.7182819    // ode 関数による数値解
--> ye = exp(x)-x-1
 ye  =
   0.   0.0214028   0.0918247   0.2221188   0.4255409   0.7182818    // 厳密解 exp(x)-x-1
```

1 階微分方程式 $y' = x + y,\ y(0) = 2$ の例 (初期条件のみが上記の例と異なる)

同様に， 同じ微分方程式を，初期条件 $x_0 = 0$, $y_0 = 2$ で解くことを考える．

$$y' = x + y, \quad y(0) = 2 \tag{8.3}$$

初期条件のみを変更して，同様に ode 関数を用いて解けばよい．

```
--> deff('yd=f(x,y)','yd=x+y');       // f(x,y)=x+y の定義
--> x0=0; y0=2;                       // 初期条件を y(0)=2 に変更 (変更したのはこの行のみ)
--> x=x0:0.2:1;                       // x=[0,1] の区間で，刻み幅 0.2 で解く
--> y=ode(y0,x0,x,f)                  // ode 関数で微分方程式を解く
 y  =
    2.   2.4642082   3.0754742   3.8663565   4.876623    6.1548466    // ode 関数による数値解
--> ye = 3*exp(x)-x-1
 ye  =
    2.   2.4642083   3.0754741   3.8663564   4.8766228   6.1548455    // 厳密解  3*exp(x)-x-1
```

図 8.2 に，上記で得られた微分方程式 $y' = x + y$ の解 (y の値) を図示する．

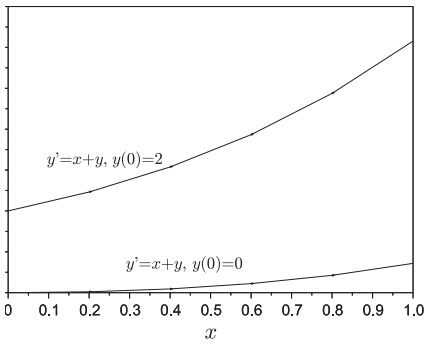

図 8.2: 1 階微分方程式 $y' = x + y$ の 2 種類の初期条件 $y(0) = 0$, および $y(0) = 2$ に対する解

8.2.2 ode 関数による高階微分方程式の解法

ここでは，次式のような n 階 (高階) 微分方程式を数値的に解く方法について説明する．

$$y^{(n)} = f\left(x, y, y', y'', \ldots, y^{(n-1)}\right), \tag{8.4}$$
$$\text{初期条件：} y(x_0) = \alpha_0,\ y'(x_0) = \alpha_1,\ y''(x_0) = \alpha_2,\ \ldots,\ y^{(n-1)}(x_0) = \alpha_{n-1}$$

2 階微分方程式の解法 ($y'' = -0.3y' - \sin(y),\ y(0) = \pi/2,\ y'(0) = 0$ の例)

例として，ode 関数を用いて，次式で与えられる 2 階微分方程式 ($n = 2$) を解く．

$$y'' = -0.3y' - \sin(y), \tag{8.5}$$
$$\text{初期条件：} y(0) = \pi/2, \quad y'(0) = 0$$

n 階の微分方程式を ode 関数で解く場合，引数は，(1) 左から初期条件の縦ベクトル $[\alpha_0, \alpha_2, \ldots, \alpha_{n-1}]'$，(2) x_0 の値，(3) 数値解を求める区間の x 軸の刻みの系列 x_0〜x_N の横ベクトル，(4) 関数名の順で，次のように記述する．但し以下の例では，$u = y$, $v = y'$ としている．

```
--> deff('ydot=f(x,y)', ['du=y(2)', 'dv=-0.3*y(2)-sin(y(1))', 'ydot=[du;dv]'])
                                    // 関数 u'=y'=f(x,u,v), v'=y"=g(x,u,v) の定義
--> x0=0; y0=[%pi/2;0];             // 初期条件の設定
--> x=0:0.1:15;                     // x=[0,15] の区間で刻み幅 0.1 で解く
--> y=ode(y0,x0,x,f);               // ode 関数で微分方程式を解く
--> plot2d(x,y(1,:));               // 微分方程式の解 y をプロット
--> plot2d(x,y(2,:));               // 微分方程式の解 y' を重ねてプロット (図 8.3)
```

ode 関数で得られた微分方程式の解 y, y' を図 8.3 に示す．n 階 ($n > 3$) の微分方程式についても，ode 関数で同様に解くことができる．この場合，ode の引数で与える初期値 y0 と，deff で定義する関数 ydot を要素が n 個の縦ベクトルとすればよい．

8.2.3 ode 関数による連立 1 階微分方程式の解法の例

連立 1 階微分方程式を数値的に解く方法についてロトカ・ボルテラの方程式の例を用いて説明する．

連立 1 階微分方程式の解法 (ロトカ・ボルテラ方程式の例)

ロトカ・ボルテラ方程式とは,「被食者」と「捕食者」の個体数の割合の変化を表す単純な生態系モデルで，被食者の個体数 y_1 と，捕食者の個体数 y_2 の関係を，連立微分方程式で次のように表したものである．

$$y_1' = y_1 - y_1 y_2 - \frac{1}{10} y_1^2 \tag{8.6}$$
$$y_2' = y_1 y_2 - y_2 - \frac{1}{20} y_2^2 \tag{8.7}$$

例えば，初期条件を

$$y_1(0) = 2,\ y_2(0) = 1 \tag{8.8}$$

として，連立微分方程式を解くと次のようになる．なお，連立微分方程式を解く場合の ode 関数の使い方は，高階微分方程式を解く場合とほぼ同じである．

```
--> deff('ydot=f(x,y)', ['dy1=y(1)-y(1)*y(2)-y(1)^2/10', 'dy2=y(1)*y(2)-y(2)-y(2)^2/20',
    'ydot=[dy1;dy2]']);             // y1', y2' の定義
--> x0=0; y0=[2;1];                 // 初期条件の設定
--> x=0:0.01:50;                    // x の刻み幅と範囲を設定
--> y=ode(y0,x0,x,f);               // 微分方程式を解く
--> plot(y(1,:),y(2,:));            // y1, y2 のプロット (図 8.4)
```

図 8.4 のように，式 (8.8) で与えられる初期条件では，捕食者の数に対して被食者の数が多いため，しばらくの間は捕食者の数が増加するが，捕食者が増えすぎると被食者の数が減るために捕食者の数も減る．このようなサイクルを繰り返し，ある均衡な値に近付いている様子がわかる．

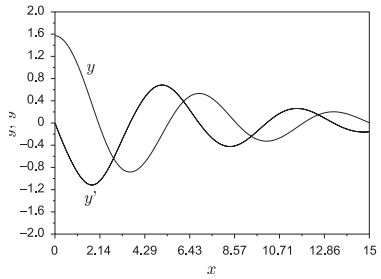

図 8.3: 例題の 2 階微分方程式の解

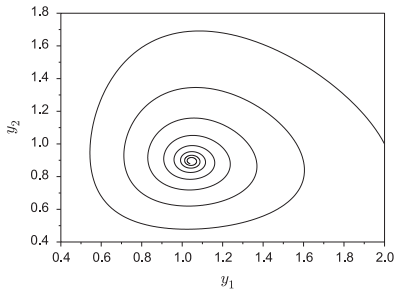

図 8.4: ロトカ・ボルテラの方程式の解

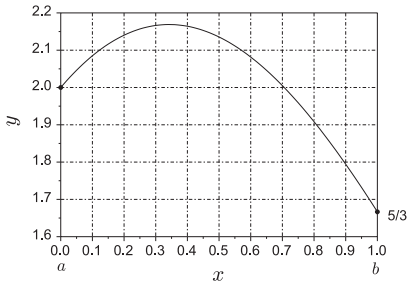

図 8.5: shooting.sci の実行結果

8.3 シューティング法 (狙い撃ち法) による境界値問題の解法の例

8.3.1 線形の境界値問題 $y'' = p(x)y' + q(x)y + r(x), \quad y(a) = y_a,\ y(b) = y_b$ の例

境界値問題とは，ある異なる 2 点における条件 (境界条件) が与えられており，その条件を満たす解を求める問題である．例として，次に示す線形の境界値問題をシューティング法を用いて解く．

$$y'' = \frac{2x}{x^2+1}y' - \frac{2}{x^2+1}y + x^2 + 1, \quad y(0) = 2, \quad y(1) = \frac{5}{3}$$

ちなみに，この線形境界値問題の厳密解は次の通りである．

$$y = \frac{x^4}{6} - \frac{3}{2}x^2 + x + \frac{x^2}{2}$$

シューティング法を適用する境界値問題の微分方程式の関数定義 ”f.sci”

```
// ---------------------------------------------------------------------------
// シューティング法を適用する境界値問題の例題の微分方程式を定義する関数
// ---------------------------------------------------------------------------
function dz = f(x,z)
dz = [ z(2)                                                  // z(1)'= u'= z(2)
       2 * x./(x^2 + 1).*z(2) - 2/(x^2 + 1).*z(1) + x^2 + 1 // z(2)'= p(x)u'+q(x)u+r(x)
       z(4)                                                  // z(3)'= v'= z(4)
       2 * x./(x^2 + 1).*z(4) - 2/(x^2 + 1).*z(3) ];         // z(4)'= p(x)v'+q(x)v
endfunction
```

シューティング法で境界値問題を解くスクリプトの例 2 ”shooting.sci”

上記の境界値問題を ode 関数を用いて解くスクリプトの例を以下に示す.

```
// ---------------------------------------------------------------------------
// シューティング法により境界値問題の例題を解くスクリプト
// ---------------------------------------------------------------------------
clear;                                          // 定義済みの変数の消去
exec('f.sci')                                   // 例題の微分方程式の関数定義の読み込み
x = 0:0.001:1;                                  // x 軸の刻み
n = length(x);                                  // x 軸の刻みの数
a  = 0;  b = 1;                                 // 区間 [a,b]=[0,1]
ya = 2; yb = 5/3;                               // x=a,b における境界条件
z0 = [ya,0,0,1]';                               // 微分方程式の解の初期値
z = ode(z0, a, x, f);                           // ode 関数を利用

y = z(1,:) + (yb - z(1,n)) * z(3,:)./ z(3,n);   // 結果: y(x) = u(x)+(yb-u(b))/v(b)*v(x)
plot2d(x,y); xgrid;                             // グラフの表示，グリッド表示
```

exec('shooting.sci') として，スクリプトを実行すると，図 8.5 が表示される．