C 言語で数値計算プログラミング

桂田 祐史

2003 年 6 月 9 日

はじめに

この文書は (主に UNIX 環境において) C 言語を使って数値計算する際の know how を解説したものである。

昔はよく「数値計算ならば FORTRAN」と言われたものだが、本当はいつの時代もユーザーは手近で利用できる便利な言語を使ってプログラムを書いて計算してきたものである。実際、私は修士論文の計算は MS-DOS パソコン上の TURBO Pascal で行なった。これを書いている現在ならば、C か C++ あるいは人によっては Java, Visual BASIC などを使うかもしれない。今時数値計算するからと言って FORTRAN と限ったものでもないだろう、というわけである。

とは言っても FORTRAN 文化も馬鹿にできない。よく勘違いされるのだが、一つの言語とその処理系だけを使って仕事をするのではない。さまざまなツール、ライブラリィ、テキスト、さらには蓄積されたノウハウ、その他もろもろのものを使うことになる。

特に数値計算の本やライブラリィでは FORTRAN による遺産が多く残されているが、同じものの C 言語版を得ようとすると、「途方に暮れる」というのが正直な感想ではないだろうか。

そういうわけで、書きたくもないけれど、学生指導の都合上しかたなく書いたプリント類が結構たまってきたので、それをまとめたのがこの文書である。

C の勉強の仕方

普通 C 言語を勉強する場合、C 言語の開発者自身が書いた

> カーニハン & リッチー, プログラミング言語 C, 共立出版 (19??)
> 原著

(以下 K&R と略記する) を持っておいた方が良いと言われている (ちなみに翻訳でなく、原著を読むべきだという説も根強い — 確かに翻訳では細かいところがわかりづらく (不正確？)、英語を読むことにそれほど抵抗がない人には原著が勧められると思う)。

しかし、それまでプログラミングの経験がない人にとって、いきなり K&R を読むのは敷居が高いというのも良く言われていることである。これまで学生に与えて好評だったのは

> 柴田望洋, 明快 C 言語入門編, ソフトバンク (1999)

である。この本に書いてあるプログラムを端から順にコンピューターに打ち込んでコンパイル&実行して試すというのが、自習法として勧められる。

良く言われることだが、上級者の書くプログラムを読んでエッセンスを吸収するという学び方も有効である。それもあって、C 言語のプログラミングにある程度慣れてきたら、目的をはっきりさせて勉強すべきである (自分の目的にあった、上級者の書いたプログラム例を探す、ということになる)。そのためには C 言語の入門書よりは、直接数値計算の本を手に取るべきだと思う。ところが、C 言語で書かれた数値計算プログラムの良い本は案外と少ないものである。ここでは

> 戸川隼人, UNIX ワークステーションによる科学技術計算ハンドブック, サイエンス社 (1992)

をあげておく。

どういうプログラムに C 言語を用いるべきか

筆者が初めてプログラムを書き出した頃は (1980 前後)、数値計算プログラムは Fortran で書けと言われたものである。それが就職した頃 (1990) には、既存の数値計算ライブラリィを利用しない場合には、C でも良いと考える人が増えていた。

一方、今では MATLAB のようなソフトウェアが普及していて、多くの数値計算プログラムが MATLAB で書く方がずっと簡単に作成でき、かつ十分効率的に実行できるようになってしまった (しばしばユーザーが C で書くよりも高速に動作する)。

また、長年大きな問題となっていて、グラフィックスや GUI に関して、近年登場した Java は大きな優位性を持っている。

そこで、今では次のように考えている。

- 数値計算をする学生に勧められる (マスターすべき) プログラミング言語は、Java, C (C++), MATLAB の 3 つである。
- とにかく早く結果が得たい場合、まずは MATLAB の利用を考える。
- MATLAB では不十分な場合は、グラフィックスのインターフェイスが必要かどうかで、Java と C のいずれかを選ぶか考える。GUI が重要な場合は Java を採用すべきである。

例えば、最近学生に書かせたプログラムで説明すると、

1. 『区間演算をサポートしたプログラミング言語処理系』... この手の、システムの細かいことをいじる必要があり、効率が要求されるプログラミングは C 言語を採用するのが良いと判断した。yacc (bison) のような C 言語用のこなれた構文解析器があったことも要因である。
2. 『円盤領域における Laplacian の固有値問題』... こういう線形計算がらみのは断然 MATLAB である。C や Java で固有値問題のライブラリィを使えるようにするのは少し面倒だし (特に Windows 環境)、たとえライブラリィがあっても、プログラミングの手間が違う。
3. 『渦糸の力学系』,『BZ 反応のシミュレーション』... こういうのは常微分方程式であり、実行効率はさほど問題にならない。ユーザーインターフェイスの部分が使い勝手 (実験の作業効率) を大きく左右する。こういうのは Java の一人勝ちだ。

目 次

第 1 章　コンパイル、実行の仕方 **8**

1.1 はじめに . . . 8
1.2 とにかくコンパイル . . . 8
1.3 実行形式の名前の指定 (-o オプション) . . . 9
1.4 システムのライブラリィをリンクする (-l オプション) . . . 9
1.5 複数のソース・ファイルからなるプログラムのコンパイル&リンク . . . 10
1.6 cco コマンド . . . 12
1.7 -I, -L, -R . . . 12
1.8 便利なオプション — 最適化 -O, 警告レベルをあげる -W -Wall . . . 12
1.9 -g . . . 13
1.10 -pg . . . 13
1.11 -E . . . 13
1.12 -S . . . 14
1.13 GNU Emacs の利用 . . . 16
1.13.1 原始的なプログラム書き . . . 16
1.13.2 compile コマンド . . . 16
1.13.3 GNU Emacs とタグ・ジャンプ . . . 16

第 2 章　数値計算のための C プログラミング入門 **18**

2.1 イントロ — サンプル・プログラムによる勉強 . . . 18
2.1.1 最初のプログラム . . . 18
2.1.2 if 文と goto 文 . . . 19
2.2 数列 . . . 21
2.2.1 for ループと数列 . . . 22
2.2.2 極限の推定 . . . 24
2.3 級数の和 . . . 25
2.3.1 簡単な例 . . . 25
2.3.2 Taylor 級数の計算 . . . 27
2.4 方程式を解く . . . 30
2.4.1 二分法 . . . 30
2.4.2 Newton 法 . . . 31
2.5 常微分方程式の初期値問題 . . . 32
2.5.1 Euler 法 . . . 33
2.5.2 Runge-Ketta 法 . . . 35

第 3 章　グラフを描こう (1) **37**
3.1 fplot ライブラリィ, ccx コマンド, f77x コマンド . . . 37
3.2 関数・サブルーチン一覧 . . . 37
3.3 図の印刷法 . . . 39
3.4 例題による解説 . . . 39
3.4.1 1 変数関数のグラフ . . . 39
3.4.2 簡単な動画 . . . 42
3.5 サンプル・プログラム (解説抜き) . . . 45
3.5.1 定数係数線型常微分方程式の相図 . . . 45
3.5.2 熱方程式の初期値境界値問題の可視化 . . . 49
3.6 ccx, f77x の正体 . . . 56
3.7 TEX への取り込み . . . 56
3.8 ccg の関数 fsymbol() の実現法 . . . 57

第 4 章　数値計算プログラミング **58**

第 5 章　グラフを描こう (2) GLSC **59**
5.1 はじめに — GLSC とは . . . 59
5.2 6701 号室では？ . . . 59
5.3 GLSC のサンプル・プログラム . . . 62
5.3.1 何をするプログラムか . . . 62
5.3.2 ソースプログラム `draw-graph.c` . . . 62
5.3.3 読んでみよう . . . 64
5.4 GLSC+ について . . . 67
5.4.1 その目的 . . . 67
5.4.2 新しく導入した関数 . . . 67
5.4.3 サンプル・プログラム・ソース . . . 67
.1 よく使う関数の説明 . . . 70
.2 PostScript への変換 (g_out に関するノウハウ) . . . 73
.3 桂田研学生向け . . . 74
.4 Cygwin+XFree86 環境での利用 . . . 76

付 録 A 常微分方程式 **77**

付 録 B 常微分方程式の初期値問題 — とにかく始めてみよう **78**
B.1 はじめに . . . 78
B.2 目的とする問題 . . . 78
B.3 離散変数法 . . . 79
B.4 Euler 法 . . . 79
B.5 Runge-Kutta 法 . . . 80
B.6 漸化式のプログラミング . . . 80
B.6.1 F が実数値の場合 . . . 80
B.6.2 F がベクトル値の場合 . . . 82

B.7 プログラミング課題 84

付 録 C 常微分方程式の初期値問題の数値解法 **85**
C.1 はじめに 85
C.1.1 常微分方程式って何だったっけ — 復習 85
C.1.2 これからの目標 86
C.2 数値解法 (1) 86
C.2.1 問題の設定と数値解法の基本原理 86
C.2.2 Euler(オイラー) 法の紹介 87
C.2.3 プログラミングの仕方 87
C.2.4 例題 88
C.2.5 Euler 法の収束の速さ 89
C.3 Runge–Kutta (ルンゲ–クッタ) 法 91

付 録 D 定数係数線形常微分方程式 **94**
D.1 問題の説明 — 定数係数線形常微分方程式 94
D.2 例題プログラムによる実験 95
D.2.1 例題プログラムの使い方 95
D.2.2 解説 97
D.3 補足 — 紙と鉛筆で解く方法 99
D.3.1 定数係数線形常微分方程式の解の公式, 行列の指数関数 99
D.3.2 $N = 2$ の場合の e^{tA}, $e^{tA}\vec{x}_0$ 99

付 録 E 力学系とリミット・サイクル **101**
E.1 力学系と Poincaré のリミット・サイクル 101
E.1.1 力学系 101
E.1.2 平衡点と線形化 101
E.1.3 リミット・サイクル 104
E.2 追加の問題 105
E.3 プログラム 106
E.3.1 dynamics.f 106
E.3.2 dynamics.c 109
E.4 参考文献 112

付 録 A ヒント集 **114**
A.1 インデンテーション 114
A.1.1 GNU indent 114
A.2 参考になる文書 114

付 録 B がらくた箱 **115**
B.1 参考書 115
B.2 C 言語の種類 115
B.3 UNIX 上の C コンパイラーの使い方 116

B.4 数値データの種類、変数宣言 . . . 117
B.5 数の型とのおつきあい . . . 121
B.5.1 定数にも型がある . . . 121
B.5.2 式の中の型変換について . . . 122
B.5.3 int の割算に注意 . . . 122
B.6 数学関数の使い方 . . . 123
B.7 配列の使い方 . . . 123
B.7.1 添字は 0 から . . . 123
B.7.2 配列の大きさは整定数 . . . 123
B.8 ANSI C のプロトタイプ宣言とは？ . . . 125
B.9 コンパイラーなどのエラーメッセージを読むための単語帳 . . . 125
B.10 “Segmentation fault”, “Bus error” って何ですか？ . . . 126
B.11 数学定数どうしよう？ . . . 127
B.12 C で負の添字を使う方法 . . . 128
B.13 scanf() を使うのは邪道だと言われちゃった . . . 129
B.14 ポインターについてどれだけ知ればいいですか？ . . . 132
B.15 行列はどうする？ . . . 132
B.15.1 はじめに . . . 132
B.15.2 1 次元配列の方法 . . . 132
B.15.3 ポインター配列の方法 . . . 133
B.15.4 二つの方法の優劣 . . . 134
B.15.5 サンプル・プログラム . . . 134
B.16 FORTRAN プログラムを C に書き換える . . . 136
B.16.1 プログラムの構成 . . . 136
B.16.2 定数・変数の宣言 . . . 137
B.16.3 式 . . . 140
B.17 NaN って何ですか . . . 144
B.18 コンパイルとは何ですか . . . 144
B.19 NEmacs 虎の巻 @(#) Oct 4 1992 . . . 144
B.19.1 NEmacs 入門編 . . . 144
B.19.2 NEmacs 初級編 . . . 144

付 録C Fortran と一緒に使う **146**
C.1 UNIX における C と Fortran の併用 . . . 146
C.2 プログラムの書き方 . . . 146
C.2.1 副プログラムの名前について . . . 146
C.2.2 引数について . . . 146
C.3 配列 . . . 146
C.3.1 プログラム例 . . . 147
C.4 コンパイルの仕方 . . . 147
C.5 LAPACK . . . 148

付 録D C++ を使ってみよう **149**
D.1 C++ について私が考えること . . . 149
D.1.1 数学屋にとって面白く有望である . . . 149
D.1.2 変化が速すぎる . . . 149
D.1.3 printf() ファミリーの代替物がないぞ！ . . . 149
D.2 簡単な入門 . . . 149
D.2.1 printf() の代りに cout と insertion 演算子 << . . . 149
D.2.2 `cout.width()` とインサーター `setw()` . . . 151
D.2.3 左詰め, 右詰め . . . 152
D.2.4 フラッシュ . . . 153
D.2.5 8 進数, 10 進数, 16 進数 . . . 153
D.2.6 浮動小数点数の書式指定 (1) . . . 156
D.2.7 浮動小数点数の書式指定 (2) 安直な `form()` . . . 167
D.2.8 文字列 . . . 167
D.2.9 sprintf() の代りに . . . 169
D.2.10 sscanf() の代りに . . . 170
D.3 vector . . . 171
D.4 ファイル入出力 . . . 172
D.5 数学関数 . . . 175
D.6 複素数の扱い . . . 175
D.7 new と delete ... 動的メモリー確保 . . . 176
D.8 C で書かれたライブラリィとのリンク . . . 176
D.9 書式付き出力 . . . 176
D.10 クラス定義についての注意 . . . 179
D.10.1 デフォールト・コンストラクター . . . 179
D.10.2 コピー・コンストラクター . . . 180
D.10.3 代入演算子 . . . 181
D.11 valarray . . . 183
D.12 注目しているソフトウェア . . . 186
D.12.1 FreeFEM . . . 186
D.12.2 Blitz . . . 186
D.12.3 MATPACK++ . . . 186
D.13 URL . . . 186
D.14 参考文献 . . . 187

付 録E Java もやってみよう **189**

第1章　コンパイル、実行の仕方

(ここでは、UNIX 環境で GCC などのソフトを利用して C のプログラム開発をする方法を説明する。Windows でも Cygwin を利用する場合にはほぼそのまま当てはまる。)

1.1　はじめに

この節では GCC[1] の C Compiler である gcc を使って、C プログラムをコンパイルする方法を説明する。

http://www.sra.co.jp/wingnut/gcc/
『GCC マニュアル日本語訳』

1.2　とにかくコンパイル

一つのソースプログラムからなる C プログラムをコンパイルするには、オプションなしで gcc ファイル名 とすれば良い。すると a.out という名前の実行形式が生成される。それを実行するには ./a.out とする。

サンプル・プログラム myprog.c

```
/* myprog.c */
#include <stdio.h>

int main()
{
    printf("Hello, world.\n");
    return 0;
}
```

単一のソース・プログラムのコンパイル

```
oyabun% ls              ← どういうファイルがあるか
myprog.c                → myprog.c というファイルがある
oyabun% gcc myprog.c    ← myprog.c をコンパイル
oyabun% ls
a.out*      myprog.c    → a.out が出来ている
```

[1]GNU Project を推進する FSF (Free Software Foundation, Richard Stallman (rms) が主宰している) により開発されたコンパイラー・システムで、現在は Cygnus Solutions がメンテナンスをしている。最初は GNU C Compiler の略とされていたが、現在では GNU Compiler Collection の略だとされている。

実行

```
oyabun% ./a.out   ← a.out を実行
Hello, world.
oyabun%
```

1.3 実行形式の名前の指定 (-o オプション)

-o 名前 で名前を指定してコンパイル

```
oyabun% ls                          ← どういうファイルがあるか
myprog.c                            → myprog.c というファイルがある
oyabun% gcc -o myprog myprog.c      ← myprog.c をコンパイル
oyabun% ls
myprog*      myprog.c               → myprog が出来ている
```

実行

```
oyabun% ./myprog   ← myprog を実行
Hello, world.
oyabun%
```

一口メモ

後で説明する make を使うと、この場合は (Makefile を書かなくても) 単に

```
oyabun% make myprog
gcc myprog.c  -o myprog
oyabun%
```

でコンパイルできる[2]。

1.4 システムのライブラリィをリンクする (-l オプション)

例えば、三角関数のような数学でよく使われる関数は、ライブラリィ関数として「数学関数ライブラリィ」の中に用意されている。

そのライブラリィ・アーカイブのパス名は通常 /usr/lib/libm.a であり、それをリンクするには -lm というオプションを指定する。一般に libxyz.a というライブラリィをリンクするには-lxyz を指定する。

[2]数学科の WS の設定では、環境変数 CC の値は gcc にセットされている。この設定がされていないときは、gcc でなく cc を使ってコンパイルされることになる。

サンプル・プログラム sintable.c

```
#include <stdio.h>
#include <math.h>

int main()
{
    int i, n;
    double pi, x, dx;
    pi = 4.0 * atan(1.0);
    n = 90;
    dx = 0.5 * pi / n;
    for (i = 0; i <= n; i++) {
        x = i * dx;
        printf("%2d %g\n", i, sin(x));
    }
    return 0;
}
```

-l 名前 でライブラリィをリンク

```
oyabun% gcc -o sintable sintable.c -lm
```

1.5 複数のソース・ファイルからなるプログラムのコンパイル &リンク

ここまでの例では、プログラムがごく簡単だったこともあって、一つのソース・ファイルをコンパイルすることで実行可能なプログラムが出来上がったが、普通は複数のソース・ファイルから一つのプログラムが構成されている。

サンプル・プログラム main.c

```
#include <stdio.h>

int main()
{
  double a, b, sum(double, double), product(double, double);
  printf("input two numbers: ");
  scanf("%lf%lf", &a, &b);
  printf("sum=%g\n", sum(a, b));
  printf("product=%g\n", product(a, b));
  return 0;
}
```

サンプル・プログラム sub1.c

```
double sum(double a, double b)
{
  return a + b;
}
```

サンプル・プログラム sub2.c

```
double product(double a, double b)
{
  return a * b;
}
```

C コンパイラー gcc は複数のソース・ファイルを一括してコンパイル& リンクすることが可能である。

一括してコンパイル

```
oyabun% gcc -o sum_product main.c sub1.c sub2.c
oyabun% ./sum_product
input two numbers:  2 3
sum=5
product=6
oyabun%
```

分割コンパイル

```
oyabun% gcc -c main.c
oyabun% gcc -c sub1.c
oyabun% gcc -c sub2.c
oyabun% gcc -o sum_product main.o sub1.o sub2.o
```

Makefile

```
sum_product: main.o sub1.o sub2.o
        gcc -o sum_product main.o sub1.o sub2.o
```

make

```
oyabun% make
gcc -O -pipe -c main.c
gcc -O -pipe -c sub1.c
gcc -O -pipe -c sub2.c
gcc -o sum_product main.o sub1.o sub2.o
oyabun% make
make:  'sum_product' is up to date.        → 既にコンパイルしてあるので、何もしない。
oyabun% mule -nw main.c                    ← main.c を少し書き換えてみる。
oyabun% make
gcc -O -pipe -c main.c                     → main.c だけはコンパイルし直す必要がある。
gcc -o sum_product main.o sub1.o sub2.o    ← 他は元からあるものを使ってリンクする。
oyabun%
```

make について一から説明を始めると長くなるので、本やインターネット上の WWW ページで勉強すること。

(1) 「Makefile の書き方 , make の使い方」 (by 名古屋大学 朝倉宏一氏)
http://www.watanabe.nuie.nagoya-u.ac.jp/member/staff/asakura/lectures/make/

(2) 「make の使い方」 (by 京都大学 安永数明氏)
http://www-clim.kugi.kyoto-u.ac.jp/yasunaga/make.html

(3) 「make コマンド」
http://www.kek.jp/ftp/kek/usg/public_html/unix_doc/node88.html

1.6 cco コマンド

私の先輩が教えてくれた csh 用の別名定義 (もちろん tcsh でもそのまま使用可) を紹介しておく。

別名定義による cco コマンド — ~/.cshrc に書いておく

```
alias cco 'gcc -O -o \!^:r \!* -lm'
```

これを用いて cco myprog.c sub1.o sub2.o とすると、gcc -O -o myprog myprog.c sub1.o sub2.o -lm を実行することになる。

1.7 -I, -L, -R

インクルード・ファイルやライブラリィ・アーカイブ・ファイルなどをシステム標準のディレクトリィから読む場合、そのディレクトリィを明示する必要がある。インクルード・ファイルを探索するディレクトリィを指定するには -I ディレクトリィ とする。ライブラリィ・ファイルを探索するディレクトリィを指定するには -L ディレクトリィ とする。

例えば、数学科 WS の環境で、GLSC を使うプログラムをコンパイルする場合、/usr/local/include にある glsc.h と、/usr/local/lib にある libglscd.a と、/usr/local/X11R6.3/lib にある libX11.a が必要になるので、例えば以下のようにする。

```
oyabun% gcc -I/usr/local/include -o graph graph.c -L/usr/local/lib
          -L/usr/local/X11R6.3/lib -lglscd -lX11 -lm
```

数学科の WS では、環境変数を

```
setenv LD_LIBRARY_PATH /usr/local/X11R6.3/lib:/usr/local/lib:/usr/lib
```

のように設定してあるので、これで良いが、そうでなければ、-R/usr/local/lib -R/usr/local/X11R6.3/lib も指定しなければならない。ここまで来ると make を利用しないと面倒かもしれない。

Makefile

```
CC      = gcc
CFLAGS  = -I/usr/local/include
LDFLAGS = -L/usr/local/lib -R/usr/local/lib -L/usr/local/X11R6.3/lib -R/usr/local/X11R6.3/lib
graph: graph.c
        $(CC) $(CFLAGS) -o graph graph.c $(LDFLAGS)
```

1.8 便利なオプション — 最適化 -O, 警告レベルをあげる -W -Wall

(説明を準備すべき)

最適化と警告

```
oyabun% gcc -O -W -Wall -o sintable sintable.c -lm
```

Makefile を利用する場合は、CFLAGS というマクロにこれらの値を設定すると良い。また Makefile なしの自動 make を行う場合は環境変数 CFLAGS に値を設定しておく。

CFLAGS を使った Makefile

```
CC      =       gcc
CFLAGS  =       -O -W -Wall
sintable: sintable.c
        $(CC) $(CFLAGS) -o sintable sintable.c -lm
```

make によるコンパイル

```
oyabun% make
gcc -O -W -Wall -o sintable sintable.c -lm
oyabun%
```

1.9 -g

C プログラムのデバッグ (debugging) には、いわゆる “printf デバッグ” と呼ばれる方法と、デバッガー (debugger) と呼ばれる専用のプログラムを使う方法がある。

gcc -g とすると、デバッガー用の情報を出力する。伝統的な cc (Portable C Compiler) では、-O と -g が併用できなかったが、gcc では可能になっている。

(暇を見て加筆する。)

1.10 -pg

プログラムを効率的なものに改良する際の鉄則は「ボトルネックを探し、そこの最適化に集中せよ」というものである。ボトルネックを探すには、プログラムのどの部分の実行に時間がかかるか、を調べるのが基本である。そのために行なわれるのがプロファイリング (profiling[3]) である。

gcc -p とすると prof コマンド用のプロファイリング情報が、gcc -pg とすると gprof コマンド用のプロファイリング情報が出力される。

(暇を見て加筆する。)

1.11 -E

「C 言語のソースプログラム中で行頭が # で始まるのは C 言語の命令ではなく、cpp (C preprocessor) の命令であり、マクロと呼ぶ」というような話を聞いたか読んだことがある (と期待する) が、cpp は実際にどういうことをしているのだろうか？ gcc -E とすると、cpp だけを行なうように指示することになる。

[3]「羊達の沈黙」のせいで妙な覚えられ方をしているが...

サンプル・プログラム macro.c

```
#include <stdio.h>
#include <math.h>

#ifdef M_PI /* システムが M_PI を定義してあれば、それを使う */
#define PI M_PI
#else       /* システムが M_PI を定義していなければ、自分で定義 */
#define PI 3.141592653589793
#endif

#define max(a,b) (((a)>(b)) ? (a) : (b))

int main()
{
  printf("max(π^2,10)=%g\n",
         max(PI*PI,10.0));
  return 0;
}
```

gcc -E の結果

```
oyabun% gcc -E macro.c
(途中省略)

int main()
{
  printf("max(π^2,10)=%g\n",
         ((( 3.141592653589793 * 3.141592653589793  )>( 10.0 )) ? ( 3.141592653589793 * 3.14159265358979
  return 0;
}
```

1.12 -S

gcc -S とすると、アセンブリー言語のソース・プログラム (拡張子は .s) が出力される。これを読むと、どういう機械語命令に変換されているのかが分かる。場合によっては自分で書き換えて最適化することもできる。

サンプル・プログラム wa.c

```
/*
 * wa.c
 */

#include <stdio.h>

int main()
{
    double a, b, c;
    printf("two numbers:");
    scanf("%lf%lf", &a, &b);
    c = a + b;
    printf("sum=%g\n", c);
    return 0;
}
```

サンプル・プログラム wa.s

```
        .file   "wa.c"
gcc2_compiled.:
.section        ".rodata"
        .align 8
.LLC0:
        .asciz  "two numbers:"
        .align 8
.LLC1:
        .asciz  "%lf%lf"
        .align 8
.LLC2:
        .asciz  "sum=%g\n"
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        !#PROLOGUE# 0
        save    %sp, -128, %sp
        !#PROLOGUE# 1
        sethi   %hi(.LLC0), %o0
        call    printf, 0
        or      %o0, %lo(.LLC0), %o0
        sethi   %hi(.LLC1), %o0
        or      %o0, %lo(.LLC1), %o0
        add     %fp, -24, %o1
        call    scanf, 0
        add     %fp, -32, %o2
        ldd     [%fp-24], %f4
        ldd     [%fp-32], %f2
        faddd   %f4, %f2, %f4
        sethi   %hi(.LLC2), %o0
        or      %o0, %lo(.LLC2), %o0
        std     %f4, [%fp-16]
        ldd     [%fp-16], %o4
        mov     %o4, %o1
        call    printf, 0
        mov     %o5, %o2
        ret
        restore %g0, 0, %o0
.LLfe1:
        .size    main,.LLfe1-main
        .ident  "GCC: (GNU) 2.95.2 19991024 (release)"
```

wa.s のアセンブル＆リンクにも gcc が使える

```
oyabun% gcc -o wa wa.s
oyabun% ls wa
wa*
oyabun% ./wa
```

(以下普通に実行できる — 省略)

1.13 GNU Emacs の利用

1.13.1 原始的なプログラム書き

GNU Emacs はテキスト・エディターと呼ぶのがはばかれるくらい機能豊富なプログラムである。通常の意味でソース・プログラム・ファイルを入力・編集するのには十二分である。

拡張子が .c であるファイルを編集するときは、c-mode というモードになって、C 言語プログラムの編集に便利なようになるが、あまり意識しなくてもよいだろう。

原始的なプログラミング＆デバッグでは、Emacs と kterm のような端末エミュレーターを同時に開いて[4]、Emacs ではソース・プログラムの入力＆編集、端末エミュレーターではプログラムのコンパイル＆実行をする。その場合には最低限以下のことを知っていれば何とかなる[5]。

- コンパイル時のエラー個所 (通常行番号で指示される) にジャンプするのに M-x goto-line コマンド (数学科 WS の設定では C-c g) がある。
- 関数や変数の宣言・定義・使用されている場所を探すのに grep や Emacs の検索コマンド (C-s あるいは C-r) が使える。

1.13.2 compile コマンド

M-x compile コマンドを使うと、"`Compile command:  make -k`" と出て来る。Makefile を適切に記述してあるのならこのままリターン。Makefile がなかったり (単一のソースからなる単純なプログラムの場合など)、あってもターゲットを変えたい場合は、ターゲットを添えて、例えば "`make -k myprog`" のようにしてリターン・キーを押す。すると Emacs は make をサブプロセスとして実行し、エラー記録をバッファーに取り込む。C-x ‘ とすると[6]、エラー位置にジャンプする。

M-x grep なんてのもある (説明準備中)。

1.13.3 GNU Emacs とタグ・ジャンプ

大きなプログラムを開発しているとき、関数や構造体をどこで定義したか、探すのは結構大変なものである。原始的には grep であたりをつけて、ファイルを開いて検索か行番号ジャンプをすれば良いわけだが、単純な作業も回数が多いとわずらわしい。そのため古来タグ・ジャンプというものが使われてきた。

最初に準備として etags コマンドを使ってタグファイル (デフォールトのファイル名は `TAGS`) と呼ばれるものを作る。

---

[4] ちなみに Window システムのない時代はどうしていたかと言うと、csh の job control 機能 (C-z と fg) を使うのが相場だった。ひょっとすると、その方が便利だったような気がしないでもない。今でも気が付くとそうしていたりする。

[5] 学生を見ていると、おぼろげな記憶と矢印キーでの移動、スクリーン上での行番号の数え、ですませているのが結構いる...

[6] single quote ’ ではなくて、back quote ‘ であることに注意せよ。

タグファイル TAGS を作る

```
oyabun% etags *.c *.h
oyabun% ls TAGS
TAGS
oyabun%
```

プログラムを開発中はタグファイルをしばしば更新する必要があるので、Makefile に書いてしまうのが普通です。

Makefile にこれを書き足しておこう

```
tags:
        etags *.c *.h
```

TAG ファイルを作成しておいてから Emacs を起動して、関数や構造体を探したいときに M-. とする。特にカーソルが探したい名前の上にあるとき M-. とすると、その名前が検索名のデフォールトになり、名前を打つ手間が省ける。

第2章　数値計算のための C プログラミング入門

2.1　イントロ — サンプル・プログラムによる勉強

C 言語の文法はかなり大きなものですが、そのごく一部を覚えただけで結構色々なことが出来ます。C 言語の文法入門書を一冊選んで、最初から順番にプログラムを入力&コンパイル&実行して試していく、というのもお勧めの勉強法ですが、ここでは文法を網羅することはあきらめて、とにかく目標を遂行するサンプル・プログラムをあげてみました。

2.1.1　最初のプログラム

まず最初に覚えるべきこととして、printf による画面への出力[1]（表示）、scanf によるキーボードからのデータの入力[2]（変数への設定）、数式による計算、代入文による変数への値の設定、を取り上げます。

example0.c

```
/* example0.c -- 華氏温度を摂氏温度に変換する。 */

#include <stdio.h>

int main()
{
    double c, f;

    printf("華氏温度を摂氏温度に変換します。\n");
    printf("華氏温度を入力してください。\n");
    scanf("%lf", &f);

    c = (f - 32) * 5.0 / 9.0;
    printf("華氏 %f 度は摂氏 %f 度です。\n", f, c);
    return 0;
}
```

このプログラムで華氏 100 度が摂氏で何度か計算してみた結果は次のようになります。

[1]正確に言えば画面ではなく “標準出力” への出力です。
[2]やはり、これも正確に言えば “標準入力” からの入力です。

実行結果

```
oyabun% ./example0
華氏温度を摂氏温度に変換します。
華氏温度を入力してください。
100
華氏 100.000000 度は摂氏 37.777778 度です。
oyabun%
```

文法メモ: 「double 使用のススメ」 – 実数データを扱うために C には、float, double, long double という 3 つのデータ型があります (これらは「浮動小数点数 (floating point numbers)」と呼ばれます)。プログラムを書く際にどれを使うべきか迷ったら、とりあえず double を使いましょう。printf で出力のフォーマット (書式) を指定する場合は、“%f”, “%e”, “%g” のいずれかを使います。scanf で入力のフォーマット (書式) を指定する場合は、アルファベットの ‘l’ (“long” の頭文字) を添えた “%lf”, “%le”, “%lg” を使います。

2.1.2 if 文と goto 文

上記のプログラムでは main() 関数内の命令が書かれた順番に実行されて行きますが、これだけでは簡単なことしか出来ません。ここでは条件判定の結果によって次にどの命令を実行するか決定する if 文 を使ってみましょう。

以下のプログラムは与えられた実係数 2 次方程式を解くものです。2 次方程式を解くためには判別式を計算してから、その符号を調べることで、その後の処理を選択する必要がありますが、こういうことをするために if はぴったりです。

example1.c

```
/* example1.c -- 2 次方程式を解く */

#include <stdio.h>
#include <math.h>

int main()
{
    double a,b,c,D;

    printf("実係数の 2 次方程式 a * x**2 + b * x + c = 0 を解く\n");
    printf("a,b,c=?\n");
    scanf("%lg %lg %lg", &a, &b, &c);
    if (a == 0.0) {
        printf("a = 0 ではいけません\n");
        return 0;
    }

    D = b * b - 4 * a * c;
    if (D > 0.0) {
        printf("相異なる 2 つの実根を持ちます\n");
        printf("%g\n", (- b + sqrt(D)) / (2 * a));
        printf("%g\n", (- b - sqrt(D)) / (2 * a));
    }
    else if (D == 0.0) {
        printf("重根を持ちます\n");
        printf("%g\n", - b / (2 * a));
    }
    else {
        printf("相異なる 2 つの虚根を持ちます\n");
        printf("実部 = %g\n", - b / (2 * a));
        printf("虚部 = %g\n", sqrt(- D) / (2 * a));
    }
    return 0;
}
```

実際に三つの 2 次方程式 $x^2 - 5x + 6 = 0$, $x^2 + 2x + 1 = 0$, $x^2 + 2x + 3 = 0$ を解いてみます。

実行結果

```
oyabun% ./example1
実係数の 2 次方程式 a * x**2 + b * x + c = 0 を解く
a,b,c=?
1 -5 6
相異なる 2 つの実根を持ちます
3
2
oyabun% ./example1
実係数の 2 次方程式 a * x**2 + b * x + c = 0 を解く
a,b,c=?
1 2 1
重根を持ちます
-1
oyabun% ./example1
実係数の 2 次方程式 a * x**2 + b * x + c = 0 を解く
a,b,c=?
1 2 3
相異なる 2 つの虚根を持ちます
実部 = -1
虚部 = 1.41421
oyabun%
```

文法メモ:　上のプログラムでは double データの書式指定として “%g” を用いました。ごく大雑把な (かなり乱暴です) 説明をすると、これは「そこそこ詳しく、かつなるべく手短な表現をする」ためのものです。これは C 言語の入門書の例ではあまり使われていませんが、なかなか便利です。興味のある人は “%f”, “%e” 等に変更して比べてみてください。

文法メモ:　上のプログラムでは平方根を計算するために “sqrt()” という関数を使っています。C 言語では数学で使う代表的な初等関数は大体使えるようになっています。その際にはいくつか注意すべきことがあります。

1. “#inclue <math.h>” として数学関数を定義したヘッダー・ファイルをインクルードする。
2. コンパイルする際に数学ライブラリィをリンクする。UNIX ワークステーション上の C コンパイラーでは “-lm” をコマンドに添える。例えば “cc example1.c -lm” のようにする。
3. 大抵の関数の引数 (かっこ “()” の中身のこと) は double 型の式にする。(例えば $\sqrt{2}$ を計算したい場合、“sqrt(2)” では間違いで、“sqrt(2.0)” か、あるいは “sqrt((double) 2)” のようにする。

2.2　数列

コンピューターで計算をすることのメリットのうち一番大きなものは、大量の計算も高速に実行できることです。この観点からプログラミング言語の「繰り返し (ループ)」構文は重要です。実際 **for** 文を使う頻度は非常に高く、これに習熟すればずいぶん色々な計算が出来るようになります。ここでは数列に関係したプログラミングに使ってみます。

2.2.1 for ループと数列

数列をコンピューターで調べるには、一連の番号に対する数列を次々に計算して、値を調べるのが有効です。

example2.c

```
/* example2.c -- for loop の利用 */

#include <stdio.h>
#include <math.h>

int main()
{
    int n;
    int An, Bn, Cn;
    for (n = 1; n <= 32; n++) {
        An = n * n;
        Bn = n * n * n;
        Cn = pow(2.0, (double)n);
        printf("n=%2d, An=%4d, Bn=%5d, Cn=%12d\n", n, An, Bn, Cn);
    }
    return 0;
}
```

このプログラムを実行すると、$n = 1, \cdots, 32$ に対して n^2, n^3, 2^n を計算します。ただし以下にあげる実行結果では、$n = 31, 32$ に対する 2^n の計算値は正しくありません。システムによっては、$n = 31$ に対してはプログラムが異常終了することもあります。

実行結果

```
oyabun% ./example2
n= 1, An=   1, Bn=    1, Cn=           2
n= 2, An=   4, Bn=    8, Cn=           4
n= 3, An=   9, Bn=   27, Cn=           8
n= 4, An=  16, Bn=   64, Cn=          16
n= 5, An=  25, Bn=  125, Cn=          32
n= 6, An=  36, Bn=  216, Cn=          64
n= 7, An=  49, Bn=  343, Cn=         128
n= 8, An=  64, Bn=  512, Cn=         256
n= 9, An=  81, Bn=  729, Cn=         512
n=10, An= 100, Bn= 1000, Cn=        1024
n=11, An= 121, Bn= 1331, Cn=        2048
n=12, An= 144, Bn= 1728, Cn=        4096
n=13, An= 169, Bn= 2197, Cn=        8192
n=14, An= 196, Bn= 2744, Cn=       16384
n=15, An= 225, Bn= 3375, Cn=       32768
n=16, An= 256, Bn= 4096, Cn=       65536
n=17, An= 289, Bn= 4913, Cn=      131072
n=18, An= 324, Bn= 5832, Cn=      262144
n=19, An= 361, Bn= 6859, Cn=      524288
n=20, An= 400, Bn= 8000, Cn=     1048576
n=21, An= 441, Bn= 9261, Cn=     2097152
n=22, An= 484, Bn=10648, Cn=     4194304
n=23, An= 529, Bn=12167, Cn=     8388608
n=24, An= 576, Bn=13824, Cn=    16777216
n=25, An= 625, Bn=15625, Cn=    33554432
n=26, An= 676, Bn=17576, Cn=    67108864
n=27, An= 729, Bn=19683, Cn=   134217728
n=28, An= 784, Bn=21952, Cn=   268435456
n=29, An= 841, Bn=24389, Cn=   536870912
n=30, An= 900, Bn=27000, Cn=  1073741824
n=31, An= 961, Bn=29791, Cn=  2147483647
n=32, An=1024, Bn=32768, Cn=  2147483647
oyabun%
```

文法メモ:　上のプログラムでは 2^n を計算するために pow() という関数を使っています[3]。a,b を double 型のデータとする時、pow(a,b) は a^b という値を返します。

蛇足:　四則演算などと比較して、pow() のような初等超越関数の計算は普通かなり長い時間がかかります。プログラムの効率を重視する必要がある場合には、上のプログラムの書き方はあまり良いものではありません。後述するプログラム example6.c のような工夫をするべきでしょう。ただし、この程度のプログラムでは効率よりも、読み易さ・書き易さを尊重して、上のような書き方も許されるでしょう。

[3]べき乗のことを英語で "power" といいます。例えば a^b は "a to the power b" あるいは "a to the b-th power と読みます。

2.2.2 極限の推定

数列 $\{a_n\}_{n\in\mathbf{N}}$ について、極限 $\lim_{n\to\infty} a_n$ について知りたい場合 (極限が存在するかどうか？その値は？) 、実際に数列の多くの項を次々に計算していくことで、ある程度の推定が出来ることがあります。

関数の極限 $\lim_{x\to\alpha} f(x)$ についても、$x_n \to \alpha$ となる数列 $\{x_n\}_{n\in\mathbf{N}}$ を適当に選んで、数列 $\{f(x_n)\}_{n\in\mathbf{N}}$ の極限を調べることで、それなりの情報が得られます。

次のプログラムでは $\lim_{x\to 0} \dfrac{\sin x}{x} = 1$ を「見る」ためのものです。

example3.c

```
/* example3.c -- 数列の極限を推測する */

#include <stdio.h>
#include <math.h>

int main()
{
    int k, n;
    double x;

    printf("x を 0 に近付ける時の sin(x)/x の値を調べます\n");
    printf("n=");
    scanf("%d", &n);

    printf("     x         sin(x)/x\n");
    for (k = 0; k < n; k++) {
        /* x = 2^{-k} */
        x = pow(0.5, (double)k);
        printf("%12e %12e\n", x, sin(x) / x);
    }
    return 0;
}
```

ここでは $x = x_n = \left(\dfrac{1}{2}\right)^n$, $n = 1, 2, \cdots, 12$ に対して $\dfrac{\sin x}{x}$ を計算しています。

実行結果

```
oyabun% ./example3
x を 0 に近付ける時の sin(x)/x の値を調べます
n=12
     x        sin(x)/x
1.000000e+00 8.414710e-01
5.000000e-01 9.588511e-01
2.500000e-01 9.896158e-01
1.250000e-01 9.973979e-01
6.250000e-02 9.993491e-01
3.125000e-02 9.998372e-01
1.562500e-02 9.999593e-01
7.812500e-03 9.999898e-01
3.906250e-03 9.999975e-01
1.953125e-03 9.999994e-01
9.765625e-04 9.999998e-01
4.882812e-04 1.000000e+00
oyabun%
```

2.3 級数の和

2.3.1 簡単な例

example5.c

```
/* example5.c -- 級数の和 */

#include <stdio.h>

int main()
{
    int k,n;
    double Sk,Ak;

    printf("数列の和を計算します。\n");
    printf("何項までの和を計算しますか ?\n");
    scanf("%d", &n);

    Sk = 0.0;
    for (k = 1; k <= n; k++) {
        Ak = k;
        Sk = Sk + Ak;
        printf("k=%3d, Ak=%e, Sk=%e\n", k, Ak, Sk);
    }
    return 0;
}
```

このプログラムで $\sum_{k=1}^{100} k$ を計算してみる。

実行結果

```
oyabun% ./example5
数列の和を計算します。
何項までの和を計算しますか ?
100
k=  1, Ak=1.000000e+00, Sk=1.000000e+00
k=  2, Ak=2.000000e+00, Sk=3.000000e+00
k=  3, Ak=3.000000e+00, Sk=6.000000e+00
k=  4, Ak=4.000000e+00, Sk=1.000000e+01
k=  5, Ak=5.000000e+00, Sk=1.500000e+01
k=  6, Ak=6.000000e+00, Sk=2.100000e+01
k=  7, Ak=7.000000e+00, Sk=2.800000e+01
k=  8, Ak=8.000000e+00, Sk=3.600000e+01
k=  9, Ak=9.000000e+00, Sk=4.500000e+01
k= 10, Ak=1.000000e+01, Sk=5.500000e+01
中略
k= 99, Ak=9.900000e+01, Sk=4.950000e+03
k=100, Ak=1.000000e+02, Sk=5.050000e+03
oyabun%
```

参考までに配列を使った (同じ計算をする) プログラムを掲げておく (こちらの方がもとの数式の表現に近いが、メモリーを余分に消費することになる)。

example5a.c

```
/* example5a.c -- 級数の和 ( 配列の利用 ) */

#include <stdio.h>

#define MAXN 2000

int main()
{
    int k, n;
    double S[MAXN+1], A[MAXN+1];

    printf("数列の和を計算します。\n");
    printf("何項まで計算しますか ? ");
    scanf("%d", &n);
    if (n > MAXN) exit(0);

    for (k = 1; k <= n; k++)
        A[k] = k;

    S[1] = A[1];
    for (k = 2; k <= n; k++)
        S[k] = S[k - 1] + A[k];

    for (k = 1; k <= n; k++)
        printf("k=%3d, Ak=%e, Sk=%e\n", k, A[k], S[k]);

    return 0;
}
```

2.3.2 Taylor 級数の計算

Taylor 級数の和を計算する場合、一般項を計算するのに漸化式を使うと便利なことが多い。

e の近似計算

$n = 1, \cdots, 20$ に対して $s_n = \sum_{k=0}^{n} \frac{1}{k!}$ を計算する。

example6.c

```
/* example6.c -- 自然対数の底 e を級数で計算する。 */

#include <stdio.h>

int main()
{
    int k, n;
    double Sk,Ak;

    printf("自然対数の底 e を計算します。\n");
    printf("何項まで計算しますか ? ");
    scanf("%d", &n);

    /* k = 0 のとき */
    Ak = 1.0;
    Sk = Ak;

    for (k = 1; k <= n; k++) {
        Ak = Ak / k;
        Sk = Sk + Ak;
        printf("k=%3d, Ak=%e, Sk=%20.15e\n", k, Ak, Sk);
    }
    return 0;
}
```

実行結果

```
oyabun% ./example6
自然対数の底 e を計算します。
何項まで計算しますか ? 20
k=  1, Ak=1.000000e+00, Sk=2.000000000000000e+00
k=  2, Ak=5.000000e-01, Sk=2.500000000000000e+00
k=  3, Ak=1.666667e-01, Sk=2.666666666666667e+00
k=  4, Ak=4.166667e-02, Sk=2.708333333333333e+00
k=  5, Ak=8.333333e-03, Sk=2.716666666666666e+00
k=  6, Ak=1.388889e-03, Sk=2.718055555555555e+00
k=  7, Ak=1.984127e-04, Sk=2.718253968253968e+00
k=  8, Ak=2.480159e-05, Sk=2.718278769841270e+00
k=  9, Ak=2.755732e-06, Sk=2.718281525573192e+00
k= 10, Ak=2.755732e-07, Sk=2.718281801146385e+00
k= 11, Ak=2.505211e-08, Sk=2.718281826198493e+00
k= 12, Ak=2.087676e-09, Sk=2.718281828286169e+00
k= 13, Ak=1.605904e-10, Sk=2.718281828446759e+00
k= 14, Ak=1.147075e-11, Sk=2.718281828458230e+00
k= 15, Ak=7.647164e-13, Sk=2.718281828458995e+00
k= 16, Ak=4.779477e-14, Sk=2.718281828459043e+00
k= 17, Ak=2.811457e-15, Sk=2.718281828459046e+00
k= 18, Ak=1.561921e-16, Sk=2.718281828459046e+00
k= 19, Ak=8.220635e-18, Sk=2.718281828459046e+00
k= 20, Ak=4.110318e-19, Sk=2.718281828459046e+00
oyabun%
```

$\sin 1$ の近似計算

$x - \dfrac{x^3}{3!} + \dfrac{x^5}{5!} - \cdots + \dfrac{(-1)^{k+1}x^{2k-1}}{(2k-1)!} + \cdots$ を最初の 10 項まで (x^{19} の項まで) 打ち切った級数で $\sin 1$ を計算する。

example7.c

```
/* example7.c -- Taylor 展開による sin x の計算 */

#include <stdio.h>
#include <math.h>

int main()
{
    int k, n;
    double x,Sk,Ak;

    printf("sin x を原点での Taylor 展開を用いて計算します。\n");
    printf("項数を入力して下さい ");
    scanf("%d", &n);

    printf("x = ");
    scanf("%lf", &x);

    /* k = 1 の場合 */
    Ak = x;
    Sk = Ak;

    for (k = 2; k <= n; k++) {
        Ak = - Ak * (x * x) / ((2 * k - 1) * (2 * k - 2));
        Sk = Sk + Ak;
        printf("k=%3d, Ak= %15e, Sk= %20.15e\n", k, Ak, Sk);
    }
    printf("C の組み込み関数によると sin(%g) = %20.15e\n", x, sin(x));
    return 0;
}
```

実行結果

```
oyabun% ./example7
sin x を原点での Taylor 展開を用いて計算します。
項数を入力して下さい 10
x = 1
k=  2, Ak=   -1.666667e-01, Sk= 8.333333333333334e-01
k=  3, Ak=    8.333333e-03, Sk= 8.416666666666667e-01
k=  4, Ak=   -1.984127e-04, Sk= 8.414682539682540e-01
k=  5, Ak=    2.755732e-06, Sk= 8.414710097001764e-01
k=  6, Ak=   -2.505211e-08, Sk= 8.414709846480680e-01
k=  7, Ak=    1.605904e-10, Sk= 8.414709848086585e-01
k=  8, Ak=   -7.647164e-13, Sk= 8.414709848078937e-01
k=  9, Ak=    2.811457e-15, Sk= 8.414709848078965e-01
k= 10, Ak=   -8.220635e-18, Sk= 8.414709848078965e-01
C の組み込み関数によると sin(1) = 8.414709848078965e-01
oyabun%
```

2.4 方程式を解く

2.4.1 二分法

bisection.c

```
/*
 * bisection.c -- 二分法 (bisection method) で方程式 f(x)=0 を解く
 *   コンパイルは gcc -o bisection bisection.c -lm
 *   cco コマンドがあれば cco bisection.c
 *   いずれも bisection という実行ファイルができる。
 */

#include <stdio.h>
#include <math.h>

int main()
{
  int i, maxitr = 100;
  double alpha, beta, a, b, c, eps;
  double fa, fb, fc;
  double f(double);

  printf(" 探す区間の左端α , 右端β , 許容精度ε =");
  scanf("%lf %lf %lf", &alpha, &beta, &eps);

  a = alpha; b = beta;

  fa = f(a); fb = f(b);
  if (fa * fb > 0.0) {
    printf(" f(α ) f(β ) > 0 なのであきらめます。\n");
    exit(0);
  }
  else {
    for (i = 0; i < maxitr; i++) {
      c = (a + b) / 2; fc = f(c);
      if (fc == 0.0)
        break;
      else if (fa * fc <= 0.0) {
        /* 左側 [a,c] に根がある */
        b = c; fb = fc;
      }
      else {
        /* 左側 [a,c] には根がないかもしれない。[c,b] にあるはず */
        a = c; fa = fc;
      }
      printf("f(%20.16f)=%9.2e, f(%20.16f)=%9.2e\n", a, fa, b, fb);
      if ((b - a) <= eps)
        break;
    }
    printf("f(%20.16f)=%9.2e\n", c, fc);
  }
  return 0;
}

double f(double x)
{
  return cos(x) - x;
}
```

2.4.2 Newton 法

ここでは非線形方程式を解くための代表的な方法である Newton 法を試してみましょう。$f : \mathbf{R} \to \mathbf{R}$ という関数があった時に、x についての方程式

$$f(x) = 0$$

を解くために、適当な初期値 x_0 を定めて漸化式

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

で数列 $\{x_n\}$ を定め、その極限 $x = \lim_{n\to\infty} x_n$ を見出して、それを解として採用する、というものです。

example4.c

```
/* example4.c -- Newton 法で正の数 d の平方根を求める */

#include <stdio.h>
#include <math.h>

int main()
{
    int n;
    double eps = 1.0e-15;
    double d, Xn, Xnp1;

    printf("Newton 法で正の数 d の平方根を求めます。\n");
    printf("d=");
    scanf("%lf", &d);
    if (d < 0.0) return 0;

    Xn = 1.0;

    for (n = 0; n < 100; n++) {
        Xnp1 = (Xn + d / Xn) / 2;
        printf("%20.15e\n", Xnp1);
        /* 次の繰り返しのため */
        Xn = Xnp1;
        /* 十分よい近似値が得られたかどうかチェックして、満足行く精度に
        なっていたら繰り返しを打ち切って END に飛ぶ */
        /* if (Xn == Xnp1) goto END; */
        /* if (fabs(Xn - Xnp1) < eps * fabs(Xn)) goto END; */
        if (fabs(Xn * Xn - d) / d < eps) break;
    }
    printf("Square of %20.15e = %20.15e\n", Xn,  Xn * Xn);
    return 0;
}
```

最後に求めた計算値を自乗してチェックするようになっています。このプログラムで $\sqrt{2}$ を計算すると次のようになります。

example4 の実行結果

```
oyabun% ./example4
Newton 法で正の数 d の平方根を求めます。
d=2
1.500000000000000e+00
1.416666666666667e+00
1.414215686274510e+00
1.414213562374690e+00
1.414213562373095e+00
Square of 1.414213562373095e+00 = 2.000000000000000e+00
oyabun%
```

上記のプログラム example4.c の for ループの部分は Xnp1 という変数を使わないで

```
for (n = 0; n < 100; n++) {
  Xn = (Xn + d / Xn) / 2;
  printf("%20.15e\n", Xnp);
  if (fabs(Xn * Xn - d) / d < eps) goto END;
}
```

のように書くことも出来ます (こちらの書き方が普通です)。

2.5 常微分方程式の初期値問題

(解説はさぼっています。後の章を見た方が良いかも。)

2.5.1 Euler 法

euler.c

```
/* euler.c */

#include <stdio.h>
#include <math.h>

int main()
{
    int i, N;
    double a = 0.0, b = 1.0;
    double x0;
    double t, x, h;
    double f(double, double);

    printf(" x0="); scanf("%lf", &x0);
    printf(" N="); scanf("%d", &N);

    h = (b - a) / N;

    t = a; x = x0;
    printf("%g %g\n", t, x);

    for (i = 0; i < N; i++) {
        x = x + h * f(t,x);
        t = t + h;
        printf("%g %g\n", t, x);
    }
    printf("%g %20.15e\n", t, x);
    return 0;
}

double f(double t, double x)
{
    return x;
}
```

実行結果

```
mathpc00% ./euler
 x0=1
 N=100
0 1
0.01 1.01
0.02 1.0201
0.03 1.0303
0.04 1.0406
0.05 1.05101
0.06 1.06152
0.07 1.07214
0.08 1.08286
0.09 1.09369
中略
0.91 2.47312
0.92 2.49785
0.93 2.52283
0.94 2.54806
0.95 2.57354
0.96 2.59927
0.97 2.62527
0.98 2.65152
0.99 2.67803
1 2.70481
1 2.704813829421526e+00
mathpc00%
```

2.5.2 Runge-Ketta 法

runge.c

```
/* runge.c */

#include <stdio.h>
#include <math.h>

int main()
{
    double a = 0.0, b = 1.0;
    double x0;
    double t, x, h, f(double, double);
    double k1,k2,k3,k4;
    int i, N;

    printf(" x0="); scanf("%lf", &x0);
    printf(" N="); scanf("%d", &N);

    h = (b - a) / N;

    t = a; x = x0;
    printf("%g %g\n", t, x);

    for (i = 0; i < N; i++) {
        k1 = h * f(t,x);
        k2 = h * f(t + h / 2, x + k1 / 2);
        k3 = h * f(t + h / 2, x + k2 / 2);
        k4 = h * f(t + h, x + k3);
        t = t + h;
        x = x + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
        printf("%g %g\n", t, x);
    }
    printf("%g %20.15e\n", t, x);
    return 0;
}

double f(double t, double x)
{
    return x;
}
```

実行結果

```
mathpc00% ./runge
 x0=1
 N=10
0 1
0.1 1.10517
0.2 1.2214
0.3 1.34986
0.4 1.49182
0.5 1.64872
0.6 1.82212
0.7 2.01375
0.8 2.22554
0.9 2.4596
1 2.71828
1 2.718279744135166e+00
mathpc00%
```

第3章　グラフを描こう (1)

この章の内容は少し古いです。今では fplot の利用は学生にあまり勧めていません。

fplot のソースは『公開プログラムのページ』[1] にあります。

3.1　fplot ライブラリィ, ccx コマンド, f77x コマンド

FORTRAN プログラムや C プログラムから利用できる、図形描画用のサブルーチン・ライブラリィ **fplot**[2]を紹介する。

6701 号室のワークステーションでは、C プログラムに対しては "**ccx**", FORTRAN プログラムに対しては "**f77x**" というコマンドを使うことで、この fplot ライブラリィがリンクされる[3]

例えば `heat.c` という C プログラムをコンパイルするには

```
oyabun% ccx heat.c
```

とする。`heat.c` に誤りがなければ `heat` という名前の実行形式のプログラムが出来る。

例えば `reidai1.f` というプログラムをコンパイルするには

```
oyabun% f77x reidai1.f
```

とする。エラーがなければ `reidai1` という名前の実行形式のプログラムが出来る。

3.2　関数・サブルーチン一覧

fplot には以下の関数・サブルーチンが含まれている。以下の引数 (`x0`, `y0`, `r` 等) は文字列である `s` を除いて単精度実数型 (C の `float`, FORTRAN の `real`) です[4]。

openpl() fplot ライブラリィの初期化をする。

closepl() fplot ライブラリィの後始末をする。

erase() 画面をクリアする。

[1] `http://www.math.meiji.ac.jp/~mk/program/`

[2] この名前の頭文字は、最初にプログラムを書いた (桂田の後輩の) 藤尾秀洋君の姓、座標のデータ型 (floating point numbers) という 2 つの "f" にかけてある。

[3] 東海林先生の講義では `f77g`, `ccg` というコマンドが紹介されているようだが、`f77x`, `ccx` はその兄弟分にあたる。ほぼ互換性があると考えて良い。

[4] 伝統的な C 言語との互換性を考慮して、実際には引数は `float` ではなく、`double` 型で渡すようにしてある。ANSI 規格の C 処理系を使う場合は注意が必要である。

fspace(x0,y0,x1,y1) 左下端が (x0, y0), 右上端が (x1, y1) である長方形が描けるように座標を割り当てる。この際、縦横の拡大比が等しくなるような調節が行なわれる。 x0<x1, y0<y1 でなければならない。

fspace2(x0,y0,x1,y1) スクリーンの左下端を (x0, y0), 右上端を (x1,y1) とするように座標を割り当てる (fspace() のような調節は行なわない)。この関数を利用した場合は fcircle() や farc() は使えない。 x0<x1, y0<y1 でなければならない。

label(s) 現在点に文字列を表示する。s は文字列。s は "Hungry?", "Cup noodle!" などの文字列 (残念ながら日本語は使えません)。

linemod(s) 線分のパターンを指定する。s として指定できるのは"dotted", "solid", "longdashed", "shortdashed" と"dotdashed" である[5]。

fline(x1,y1,x2,y2) 点 (x1,y1) から点 (x2,y2) までの線分を描く。現在点は (x2,y2) となる。

fcircle(x,y,r) 点 (x,y) を中心とする半径 r の円を描く。現在点は (x,y) となる。

farc(x,y,x0,y0,x1,y1) 中心が (x,y) の円弧 (始点 (x0, y0), 終点 (x1,y1)) を描く。描画は反時計回りに行なわれる。

fmove(x,y) 現在点を (x,y) に変更する。

fcont(x,y) 現在点から点 (x,y) まで線分を描く。現在点は (x,y) になる。

fpoint(x,y) 点 (x,y) に点を描き、そこを現在点とする。

座標を指定するのに倍精度実数型 (C では double, FORTRAN では double precision あるいは real*8 と宣言する) を使うことも出来るように、先頭の文字が "d" のサブルーチン (dspace, dline, dcircle, darc, dmove, dcont, dpoint) も用意してある。使い方は、先頭の文字が "f" であるものに準じる。

以下にあげるサブルーチンは、少し特殊なものなので、最初のうちは無視してしまっても構わない (最初の二つのうちのどちらかは、お世話になるかも知れない)。

xflush() それまで発行した描画命令を実際に X サーバーに送り出す (X のリクエスト・バッファーをフラッシュする)。アニメーションで切りの良いところで実行すると動きが滑らかになる。

xsync() それまで発行した描画命令を完全に実行し終わる (実際にウィンドウに図形が表示される) まで待つ。アニメーションで切りの良いところで実行すると動きが滑らかになる (少し遅くなるけど)。

fmark(x,y) 点 (x,y) にマーカーを描き、そこを現在点とする。

xor() 今後二重に描いたところは消すようにする。図形の部分消去を実現できる。元に戻すには call set とする。

[5]今のところ solid 以外は全部ただの点線になってしまう。そのうちきちんと描き分けるつもり、、、

3.3 図の印刷法

fplot には次の関数もある。

mkplot(s) 現在までに描いた図を plot(5) 形式でファイルに出力する。ファイルの名前は文字列 s で指定する。

つまり mgraph コマンドが出力するような形式でデータが出力されるので、xplot や plot2ps を利用して、画面に図を最表示したり、プリンターで印刷したり出来る。

例えば call mkplot("heat.plot") として出力したファイル heat.plot の内容を画面に表示するには

```
oyabun% cat heat.plot | xplot
```

プリンターに印刷するには

```
oyabun% cat heat.plot | plot2ps | lpr
```

とする。

複数の図を一枚の紙に印刷するには、直接印刷しないで

```
oyabun% cat heat.plot | plot2ps > heat.ps
```

のようにして heat.ps のような PostScript データを作っておいてから multi コマンドを利用する (次の例では一枚の紙に 4 つの図を収めるようにしている。heat5.ps は 2 枚目の紙に出力される。):

```
oyabun% multi -m 4 heat.ps heat2.ps heat3.ps heat4.ps heat5.ps | lpr
```

3.4 例題による解説

3.4.1 1 変数関数のグラフ

例題 1: $y = \sin x + \sin 3x + \sin 5x$ のグラフを描きなさい。

Fortran の場合

```
* reidai1.f -- サブルーチン呼び出しによるグラフ描き
      program reidai1
*      変数の宣言
*      a: 区間の左端の座標, b: 区間の右端の座標, margin: 余白の大きさ
      real Pi,a,b,margin
      integer N,i
      real h,x,f
      external f
*
```

```
      Pi = 4.0 * atan(1.0)
      a = 0.0
      b = 2.0 * Pi
      margin = (b - a) / 20
*      x 軸の分割の仕方の決定
      write(*,*) ' x 軸の区間の分割数を入力して下さい'
      read(*,*) N
      h = (b - a) / N
*      グラフィックスの初期化
      call openpl
      call fspace2(a - margin, -3.0, b + margin, 3.0)
*       始点のセット
      call fmove(a, f(a))
*       グラフ上の点を順に結ぶ
      do i = 1, N
        x = a + i * h
        call fcont(x, f(x))
      end do
*       図形を記録する
      call mkplot("reidai1.plot")
*       グラフィックスの後始末
      call closepl
      end
************************************************
      real function f(x)
      real x
      f = sin(x) + sin(3.0 * x) + sin(5.0 * x)
      end
```

C の場合

```
/*
 * reidai1.c -- サブルーチン呼び出しによるグラフ描き
 *   How to compile: ccx reidai1.c
 */

#include <stdio.h>
#include <math.h>
#include <fplot.h>

int main()
{
    /* 変数の宣言
     *  a: 区間の左端の座標, b: 区間の右端の座標, margin: 余白の大きさ
     */
    double Pi, a, b, margin;
    int N, i;
    double h, x, f(double);

    Pi = M_PI;
    a = 0.0;
    b = 2.0 * Pi;
    margin = (b - a) / 20;
    /* x 軸の分割の仕方の決定 */
    printf("x 軸の区間の分割数を入力して下さい: ");
    scanf("%d", &N);
    h = (b - a) / N;
```

```
    /* グラフィックスの初期化 */
    openpl();
    fspace2(a - margin, -3.0, b + margin, 3.0);
    /* 始点のセット */
    fmove(a, f(a));
    /* グラフ上の点を順に結ぶ */
    for (i = 1; i <= N; i++) {
        x = a + i * h;
        fcont(x, f(x));
    }
    /* 図形を記録する */
    mkplot("reidai1.plot");
    /* グラフィックスの後始末 */
    closepl();
    return 0;
}

double f(double x)
{
    return sin(x) + sin(3.0 * x) + sin(5.0 * x);
}
```

全体が、主プログラム (Fortran では、program reidai1, C では関数 main()) と関数副プログラム f の 2 つの部分からなっている。主プログラムの中で呼び出している openpl, fspace2, fmove, fcont, closepl が fplot の命令である。

openpl() は他のすべてのグラフィックス命令に先立って呼び出す。これによって図を描くためのウィンドウがオープンされる。

fspace2() はウィンドウに座標を割り振るために用いる。

Fortran の場合	`call fspace2(x0,y0,x1,y1)`
C の場合	`fspace2(x0,y0,x1,y1);`

とするとウィンドウの左下隅が (x0, y0)、右上隅が (x1,y1) になる (当然 x0<x1, y0<y1 でなければならない)。扱う問題に応じて適当な値を見い出して呼び出す必要がある。次の fmove(), fcont() は後回しにして、最後の closepl() を先に解説しよう。これは openpl() と対になる命令で、図形描画を終了させて後始末をするものである。実行がここに到達すると、プログラムは利用者がマウスでウィンドウをクリックするのを待つ状態になる。

ここまで解説した openpl(), fspace2(), closepl() は実際には画面に何も描かないが、何時でも必要である。つまりプログラムは必ず

Fortran の場合

```
.........
call openpl
.........
call fspace2(x0,y0,x1,y1)
.........
実際の描画命令の列
.........
call closepl
.........
```

C の場合

```
.........
openpl();
.........
fspace2(x0,y0,x1,y1);
.........
実際の描画命令の列
.........
closepl();
.........
```

という形になる。

fmove() は「ペンを移動する (=move)」, fcont() は「線を引きながら (つなぎながら=continue) ペンを動かす」という命令である。つまり、これらの命令では仮想のペンを考えて、それを移動することにより図形を描画する[6]。ではコンパイルしてから実行してみよう。

Fortran の場合

```
oyabun% f77x reidai1.f
reidai1.f:
 MAIN reidai1:
        f:
oyabun% reidai1
  x 軸の区間の分割数を入力して下さい
100
oyabun%
```

C の場合

```
oyabun% ccx reidai1.c
oyabun% reidai1
x 軸の区間の分割数を入力して下さい:  100
oyabun%
```

問題 1: reidai1.f や reidai1.c を修正して、軸や目盛を描くようにしなさい。軸を点線で描くなど工夫してみなさい。

3.4.2 簡単な動画

ここでは時間発展する系を表示するための簡単な動画 (アニメーション) に利用してみよう。

微分方程式を数値シミュレーションして、微分方程式の解を理解するには、動画にして見るのが有効なことが多く、ここで紹介するテクニック (というほど大したものではないが) はあちこちで役に立つ。

例題 2: 時刻 t, 位置 $x \in [0, 2\pi]$ における変移が $f(x,t) = \sin x \cos t + \sin 3x \cos 3t + \sin 5x \cos 5t$ で与えられる弦の振動を描け[7]。

現代では常識になっていることだが、実際に連続的に図を変化させなくても、十分素早く図を切り変えれば「連続的に動いている」ように見える。適当な時間間隔 Δt を定めて、時刻 $t_j = j\Delta t$ $(j = 0, 1, 2, \cdots)$ における x の関数 $f(x, t_j)$ のグラフを次々に描けばよいであろう。以下では、古いグラフを消すために erase() を用いている。

[6]実は XY プロッターという、そのものズバリ、ペンを紙の上で動かして図形を描く装置があって、fplot ライブラリィはその動作原理をモデルにして作られた。これは UNIX に昔からある plot ライブラリィ関数がお手本になっている。

[7]これは両端を固定された弦の、ある 3 つの固有振動の和である。特に $t = 0$ の瞬間には例題 1 の関数になっている。

```
* reidai2.f -- 簡単な動画（紙芝居？）
      program rei2
*      変数の宣言
      integer i,N,j,Jmax
      real Pi,a,b,margin
      real Time,h,dt,t,x,f
      external f
*      Pi=円周率，a=区間の左端，b=区間の右端，margin=余白
      Pi = 4.0 * atan(1.0)
      a = 0.0
      b = 2.0 * Pi
      margin = (b - a) / 20
*      x 軸の区間の分割数、追跡時間の入力
      write(*,*) ' x 軸上の区間の分割数='
      read(*,*) N
      write(*,*) ' 追跡時間='
      read(*,*) Time
*      区間の刻み幅
      h = (b - a) / N
*      グラフィックスの初期化
      call openpl
      call fspace2(a - margin, -2.0, b + margin, 2.0)
*      時間間隔を決め、ループの回数を計算する
      dt = 1.0 / 16.0
      Jmax = Time / dt + 0.5
******** メイン・ループ ********
      do j=0,Jmax
        t = j * dt
        call erase
        call fmove(a, f(a,t))
        do i = 1, N
          x = a + i * h
          call fcont(x, f(x,t))
        end do
        call xflush
      end do
****** メイン・ループ終り ******
*      グラフィックスの後始末
      call closepl
      end
****************************************
*      両端を固定された弦のある 3 つの固有振動の和
*      時刻 t=0 では、例題 1 に現われる関数に一致する
      real function f(x,t)
      real x,t
      f = sin(x)*cos(t)+sin(3*x)*cos(3*t)+sin(5*x)*cos(5*t)
      end

/*
 * reidai2.c -- 簡単な動画（紙芝居？）
 *  How to compile: ccx reidai2.c
 */

#include <stdio.h>
#include <math.h>
#include <fplot.h>

int main()
```

```
{
    /* 変数の宣言 */
    int i, N, j, Jmax;
    double Pi, a, b, margin;
    double Time, h, dt, t, x, f(double, double);
    /* Pi=円周率, a=区間の左端, b=区間の右端, margin=余白 */
    Pi = M_PI;
    a = 0.0;
    b = 2.0 * Pi;
    margin = (b - a) / 20;
    /* x 軸の区間の分割数、追跡時間の入力 */
    printf("x 軸上の区間の分割数=");
    scanf("%d", &N);
    printf("追跡時間=");
    scanf("%lf", &Time);
    /* 区間の刻み幅 */
    h = (b - a) / N;
    /*  グラフィックスの初期化 */
    openpl();
    fspace2(a - margin, -2.0, b + margin, 2.0);
    /* 時間間隔を決め、ループの回数を計算する */
    dt = 1.0 / 16.0;
    Jmax = Time / dt + 0.5;
    /* ******* メイン・ループ ******* */
    for (j = 0; j <= Jmax; j++) {
        t = j * dt;
        erase();
        fmove(a, f(a, t));
        for (i = 1; i <= N; i++) {
            x = a + i * h;
            fcont(x, f(x, t));
        }
        xflush();
    }
    /* ***** メイン・ループ終り ***** */
    /* グラフィックスの後始末 */
    closepl();
    return 0;
}

/*
 * 両端を固定された弦のある 3 つの固有振動の和
 * 時刻 t=0 では、例題 1 に現われる関数に一致する
 */

double f(double x, double t)
{
    return sin(x) * cos(t) + sin(3 * x) * cos(3 * t) + sin(5 * x) * cos(5 * t);
}
```

これもコンパイルして実行してみよう。

Fortran の場合

```
oyabun% f77x reidai2.f
reidai2.f:
 MAIN rei2:
        f:
oyabun% reidai2
  x 軸上の区間の分割数=
100
  追跡時間=
6.28                    ← (一周期分)
oyabun%
```

C の場合

```
oyabun% ccx reidai2.c
oyabun% reidai2
x 軸上の区間の分割数=100
追跡時間=6.28
oyabun%
```

長い追跡時間を指定した場合は、停止させたくなったときに C-C (コントロール C) を打てば止めることができる。

問題 **2:** 自分で何か動画を作ってみよ。例えば

(a) 振り子の運動

(b) はずむボール

3.5 サンプル・プログラム (解説抜き)

3.5.1 定数係数線型常微分方程式の相図

連立常微分方程式の初期値問題

$$\begin{cases} \dfrac{dx}{dt} &= ax + by \\ \dfrac{dy}{dt} &= cx + dy \end{cases} \quad (t \in \mathbf{R}), \qquad \begin{cases} x(0) &= x_0 \\ y(0) &= y_0 \end{cases}$$

(a, b, c, d, x_0, y_0 は既知定数) を Runge-Kutta 法で解いて、相図を描く。

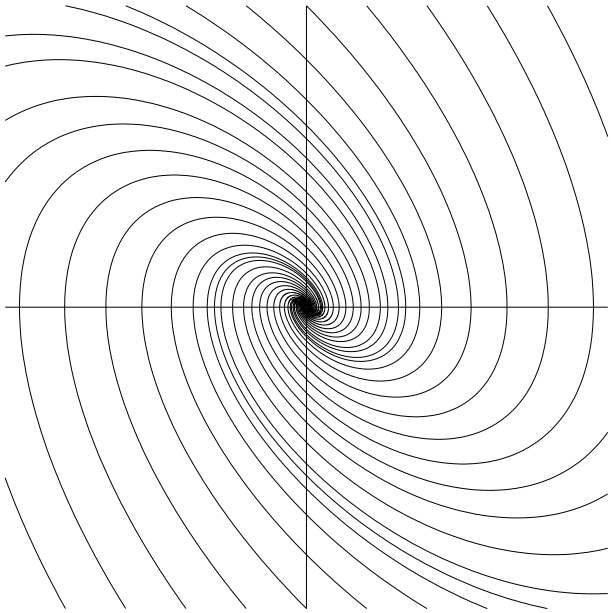
```
/*
 * dynamics.c
 */

/**********************************************************************
 **********************************************************************
 *
```

```
 *   2 次元の定数係数線形常微分方程式
 *       x'(t) = a x + b y
 *       y'(t) = c x + d y
 *  に初期条件
 *       x(0)=x0
 *       y(0)=y0
 *  を課した常微分方程式の初期値問題を解いて、相図を描く。
 *
 *  このプログラムは次の 4 つの部分から出来ている。
 *     main()
 *       主プログラム
 *       行列の係数の入力、ウィンドウのオープン等の初期化をした後に、
 *       ユーザーにメニュー形式でコマンドを入力してもらう。
 *       実際の計算のほとんどは他の副プログラムに任せている。
 *     draworbit(x0,y0,h,tlimit)
 *       (x0,y0) を初期値とする解の軌道を描く。
 *       刻み幅を h、追跡時間を tlimit とする。
 *       近似解の計算には Runge-Kutta 法を用いる。
 *     fx(x,y)
 *       微分方程式の右辺の x 成分
 *     fy(x,y)
 *       微分方程式の右辺の y 成分
 ************************************************************************/

#include <stdio.h>
#include <math.h>
#include <fplot.h>

/* 係数行列 A の成分 ( common 文により function fx,fy と共有する ) */
double a, b, c, d;
/* ウィンドウに表示する範囲 ( common 文により draworbit と共有する ) */
double xleft, xright, ybottom, ytop;

void draworbit(double, double, double, double);

int main()
{
    /* 初期値 */
    double x0, y0;
    /* 時間の刻み幅、追跡時間 */
    double h, tlimit, newh, newT;
    /* メニューに対して入力されるコマンドの番号 */
    int cmd;
    /* マウスのボタンの状態 */
    int lbut, mbut, rbut;
    /* ウィンドウに表示する範囲の設定 */
    xleft = -1.0;
    xright = 1.0;
    ybottom = -1.0;
    ytop = 1.0;
    /* 時間刻み幅、追跡時間 ( とりあえず設定 ) */
    h = 0.01;
    tlimit = 10.0;
    /* 行列の成分を入力 */
    printf(" a,b,c,d=");
    scanf("%lf %lf %lf %lf", &a, &b, &c, &d);
    /* ウィンドウを開く */
    openpl();
```

```
    fspace2(xleft, ybottom, xright, ytop);
    /* x 軸、y 軸を描く */
    fline(xleft, 0.0, xright, 0.0);
    fline(0.0, ybottom, 0.0, ytop);
    xsync();
    /* メイン・ループの入口 */
    while (1) {
        /* メニューを表示して、何をするか、番号で選択してもらう */
        printf("したいことを番号で選んで下さい。\n");
        printf(" -1:メニュー終了, 0:初期値のキー入力, 1:初期値のマウス入力,");
        printf("  2:刻み幅 h, 追跡時間 T 変更 (現在 h=%7.4f, T=%7.4f)\n",
            h, tlimit);
        scanf("%d", &cmd);
        /* 番号 cmd に応じて、指示された仕事をする */
        if (cmd == 0) {
            /* 初期値の入力 */
            printf(" 初期値 x0,y0=");
            scanf("%lf %lf", &x0, &y0);
            draworbit(x0, y0, h, tlimit);
        } else if (cmd == 1) {
            while (1) {
                printf("マウスの左ボタンで初期値を指定して下さい (右ボタンで中止)\n");
                fmouse(&lbut, &mbut, &rbut, &x0, &y0);
                if (lbut == 1) {
                    printf("(x0,y0)=(%g,%g)\n", x0, y0);
                    draworbit(x0, y0, h, tlimit);
                } else if (mbut == 1) {
                    printf("(x0,y0)=(%g,%g)\n", x0, y0);
                    draworbit(x0, y0, -h, tlimit);
                } else {
                    printf("マウスによる初期値の入力を打ち切ります。\n");
                    break;
                }
            }
        } else if (cmd == 2) {
            /* 時間刻み幅、追跡時間の変更 */
            printf(" 時間刻み幅 h, 追跡時間 T= ");
            scanf("%lf %lf", &newh, &newT);
            if (newh != 0 && newT != 0) {
                h = newh;
                tlimit = newT;
                printf("新しい時間刻み幅 h = %g, 新しい追跡時間 T = %g\n",
                    h, tlimit);
            } else {
                printf(" h=%g, T=%g は不適当です。\n", newh, newT);
            }
        } else if (cmd == -1) {
            /* 終了 -- メイン・ループを抜ける */
            break;
        }
    }
    mkplot("dynamics.plot");
    printf("fplot ウィンドウを左ボタンでクリックして下さい\n");
    closepl();
    return 0;
}

/*************************************************************************/
```

```
/*************************************************************************/
/*      指示された初期値に対する解軌道を描く */
void draworbit(double x0, double y0, double h, double tlimit)
{
    double x, y, fx(), fy();
    double k1x, k1y, k2x, k2y, k3x, k3y, k4x, k4y, t;
    /* 時刻を 0 にセットする */
    t = 0.0;
    /* 初期値のセット */
    x = x0;
    y = y0;
    /* 初期点を描く */
    fpoint(x, y);
    /* ループの入口 */
    do {
        /* Runge-Kutta 法による計算 */
        /* k1 の計算 */
        k1x = h * fx(x, y);
        k1y = h * fy(x, y);
        /* k2 の計算 */
        k2x = h * fx(x + k1x / 2.0, y + k1y / 2.0);
        k2y = h * fy(x + k1x / 2.0, y + k1y / 2.0);
        /*  k3 の計算 */
        k3x = h * fx(x + k2x / 2.0, y + k2y / 2.0);
        k3y = h * fy(x + k2x / 2.0, y + k2y / 2.0);
        /*  k4 の計算 */
        k4x = h * fx(x + k3x, y + k3y);
        k4y = h * fy(x + k3x, y + k3y);
        /*  (Xn+1,Yn+1) の計算 */
        x += (k1x + 2.0 * k2x + 2.0 * k3x + k4x) / 6.0;
        y += (k1y + 2.0 * k2y + 2.0 * k3y + k4y) / 6.0;
        /* 解軌道を延ばす */
        fcont(x, y);
        /* 時刻を 1 ステップ分進める */
        t += h;
        /* まだ範囲内かどうかチェック */
    }    while (abs(t) <= fabs(tlimit));
    /* サブルーチンを抜ける */
    /* 確実に描画が終るのを待つ */
    xsync();
}

/*************************************************************************/
/*************************************************************************/
/* 微分方程式の右辺のベクトル値関数 f の x 成分 */
double fx(double x, double y)
{
    return a * x + b * y;
}

/* 微分方程式の右辺のベクトル値関数 f の y 成分 */
double fy(double x, double y)
{
    return c * x + d * y;
}
```

マウスを使って入力できるのがちょっと面白い。



3.5.2 熱方程式の初期値境界値問題の可視化

熱方程式の初期値境界値問題

$$\begin{aligned} u_t &= u_{xx} \quad ((x,t) \in (0,1) \times (0,\infty)) \\ u(0,t) &= u_(1,t) = 0 \quad (t \in (0,\infty)) \\ u(x,0) &= f(x) \quad (x \in [0,1]) \end{aligned}$$

を差分法で解く。

```
/* heat1d.c -- 陰的スキーム（いわゆるθ 法）で熱方程式を解く
*    空間 1 次元、同次 Dirichlet 境界条件の問題
*
*   「微分方程式と計算機演習」第 11 章 p237 のプログラムを修正・拡張したもの
C     ** IMPLICIT FINITE DIFFERENCE METHOD **
C     **         FOR HEAT EQUATION         **
C     nfunc : 関数の番号 ( 5 種類の関数を用意している )
C     theta : スキームのパラメーター
C     N : 分割の数
C     t : 時刻
C     tau : 時間の刻み幅
C     Tmax : 時刻の上限
C     h : 空間の刻み幅
C     AL,AD,AU : 係数行列
C     u : DIMENSION FOR UNKNOWN FUNCTION                    */

/*
*   このプログラムのコピーが欲しい場合は
*      cp /usr/local/meiji/D97/heat1d.c .
* とすればよい。
*
* 細かい拡張
*   1) 区間の左端、右端の座標を変数 a, b に設定するようにした。
*   2) グラフを描くごとに、それまでに描いたグラフを消去するかどうか、
```

```
*       変数 erase_always で制御するようにした。
*    3) 計算した数値データを表示するか変数 print_always で制御するよう
*       にした。
*    4) 何ステップおきにグラフを描き換えるか、数値データを表示するか、
*       変数 skips で制御するようにした。skips == 1 だと、毎回グラフを
*       書き換える。
*    5) 初期条件を与える関数 f(x, nfunc) で、登録されている関数の種類
*       を増やした。nfunc == 5 まである。特に nfunc == 4,5 は非対称な
*       関数である。
*
*/

#include <stdio.h>
#include <math.h>
#include <fplot.h>

#define ndim    1000

/* 円周率 (math.h にある M_PI の値を利用する
* あるいは double PI; として、どこかで PI = 4.0 * atan(1.0); と代入
*/

#define PI  M_PI

void print100(int, double, double *);
void axis(double, double, double, double);
void fsymbol(double, double, char *);
void print_data(double, double, double, double,
int, double, int, double, double, double, double, double);
void save_picture(void);
void trid(int, double *, double *, double *, double *f);
void trilu(int n, double *al, double *ad, double *au);
void trisol(int n, double *al, double *ad, double *au, double *f);
double f(double, int);

int main()
{
    int N, nfunc, i, j, Jmax;
    double a, b, theta, h, Tmax, tau, c1, c2, c3, c4, lambda, t;
    double AL[ndim], AD[ndim], AU[ndim], ff[ndim], u[ndim + 1];
    double xleft, ybottom, xright, ytop;
/* 何ステップおきにグラフを書き換えるか */
    int skips = 1;
    double dt;
/* 以前描いたグラフを消すか？(0=No, 1=Yes) */
    int erase_always = 0;
/* 数値を表示するか?(0=No, 1=Yes) */
    int print_always = 0;

/* 区間の左端、右端 */
    a = 0.0;
    b = 1.0;

/* 入力 */
    printf("入力して下さい : nfunc(1..5)=");
    scanf("%d", &nfunc);
    printf("入力して下さい : θ =");
    scanf("%lf", &theta);
```

```
    printf("入力して下さい : N(<=%d)=", ndim);
    scanf("%d", &N);
    if (N <= 1 || N > ndim)
return 0;
    printf("入力して下さい : λ =");
    scanf("%lf", &lambda);

    h = (b - a) / N;
    tau = lambda * h * h;
    printf(" 時間の刻み幅τ = %g になりました。\n", tau);

    printf("入力して下さい : Tmax=");
    scanf("%lf", &Tmax);

    printf("図を描く時間間隔 Δ t は : ");
    scanf("%lf", &dt);
    skips = (dt + 0.1 * tau) / tau;

/* 係数行列の計算 */
    c1 = -theta * lambda;
    c2 = 1.0 + 2.0 * theta * lambda;
    c3 = 1.0 - 2.0 * (1.0 - theta) * lambda;
    c4 = (1.0 - theta) * lambda;
    for (i = 1; i < N; i++) {
AL[i] = c1;
AD[i] = c2;
AU[i] = c1;
    }

/* LU 分解する */
    trilu(N - 1, AL + 1, AD + 1, AU + 1);

/* 初期値 */
    for (i = 0; i <= N; i++)
u[i] = f(a + i * h, nfunc);

/* 出力 (t=0) */
/* 出力 (ここでは画面に数値を表示すること) は Fortran と C で
   かなり異なるので、直接の翻訳は出来ない。ごたごたするので、
   print100 という関数にまとめた */
    t = 0.0;
    print100(N, t, u);

/* グラフィックス画面の準備 -- まず画面の範囲を決めてから */
    xleft = a - (b - a) / 10;
    xright = b + (b - a) / 10;
    ybottom = -0.1;
    ytop = 1.1;

/* fplot ライブラリィの呼び出し */
    openpl();
    fspace2(xleft, ybottom, xright, ytop);

/* パラメーター、入力データ等の表示 */
    print_data(xleft, ybottom, xright, ytop,
       nfunc, theta, N, lambda, tau, Tmax, t, dt);

/* 解 u の t=0 におけるグラフを描く */
```

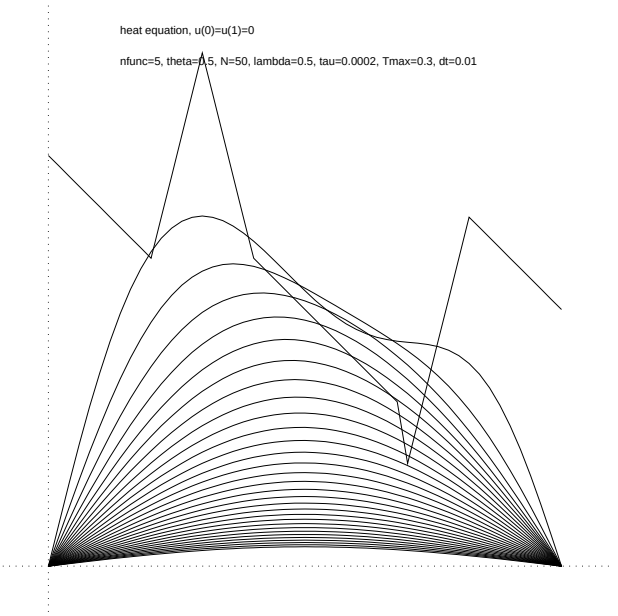
```
    fmove(a, u[0]);
    for (i = 1; i <= N; i++)
fcont(a + i * h, u[i]);
    xsync();

    Jmax = (Tmax + 0.1 * tau) / tau;

/* 繰り返し計算 */
    for (j = 1; j <= Jmax; j++) {

/* 連立一次方程式を作って解く */
for (i = 1; i < N; i++)
    ff[i] = c3 * u[i] + c4 * (u[i - 1] + u[i + 1]);
trisol(N - 1, AL + 1, AD + 1, AU + 1, ff + 1);

/* 求めた値を u[] に収める */
for (i = 1; i < N; i++)
    u[i] = ff[i];
u[0] = 0.0;
u[N] = 0.0;

/* 時刻の計算 */
t = j * tau;

if (j % skips == 0) {
    /* 数値データの表示 */
    if (print_always)
print100(N, t, u);

    /* 解のグラフを描く */
    if (erase_always) {
/* これまで描いたものを消し、パラメーターを再表示する */
erase();
print_data(xleft, ybottom, xright, ytop,
   nfunc, theta, N, lambda, tau, Tmax, t, dt);
    }
    fmove(a, u[0]);
    for (i = 1; i <= N; i++)
fcont(a + i * h, u[i]);
    xsync();
}
    }

    save_picture();

    printf("終了したければ、ウィンドウをマウスでクリックして下さい。\n");
    closepl();
    return 0;
}

/*************************************************************/

/* 初期条件を与える関数。複数登録して番号 nfunc で選択する。 */

double f(double x, int nfunc)
{
/* f(x)=max(x,1-x) */
    if (nfunc == 1) {
```

```
if (x <= 0.5)
    return x;
else
    return 1.0 - x;
    }
/* f(x)=1 */
    else if (nfunc == 2)
return 1.0;
    else if (nfunc == 3)
return sin(PI * x);
    /* f(x)= 変な形をしたもの (by M.Sakaue) */
    else if (nfunc == 4) {
if (x <= 0.1)
    return 5 * x;
else if (x <= 0.3)
    return -2 * x + 0.7;
else if (x <= 0.5)
    return 4.5 * x - 1.25;
else if (x <= 0.7)
    return -x + 1.5;
else if (x <= 0.9)
    return x + 0.1;
else
    return -10 * x + 10.0;
    }
/* やはり非対称な形をしたもの (by M.Sakaue) */
    else if (nfunc == 5) {
if (x <= 0.2)
    return -x + 0.8;
else if (x <= 0.3)
    return 4 * x - 0.2;
else if (x <= 0.4)
    return -4 * x + 2.2;
else if (x <= 0.7)
    return -x + 1.0;
else if (x <= 0.8)
    return 4 * x - 2.6;
else
    return -x + 1.5;
    }
}

/************************************************************/

/* 配列 u[] の内容 (u[0],u[1],...,u[n]) を一行 5 個ずつ表示する。 */

void print100(int N, double t, double *u)
{
    int i;
    printf("T= %12.4e\n", t);
    for (i = 0; i < 5; i++)
printf("  I     u(i)    ");
    printf("\n");
    for (i = 0; i <= N; i++) {
printf("%3d%12.4e ", i, u[i]);
if (i % 5 == 4)
    printf("\n");
    }
```

```
    printf("\n");
}

/************************************************************/

/* ウィンドウに座標軸を表示する */
void axis(double xleft, double ybottom, double xright, double ytop)
{
    linemod("dotted");
    fline(xleft, 0.0, xright, 0.0);
    fline(0.0, ybottom, 0.0, ytop);
    linemod("solid");
}

/* ウィンドウ内の位置 (x,y) から文字列 s を表示する */
void fsymbol(double x, double y, char *s)
{
    fmove(x, y);
    label(s);
}

#define X(x)    (xleft+(x)*(xright-xleft))
#define Y(y)    (ybottom+(y)*(ytop-ybottom))

/* パラメーターの値等をウィンドウに表示する */
void print_data(double xleft, double ybottom, double xright, double ytop,
int nfunc, double theta, int N, double lambda, double tau,
double Tmax, double t, double dt)
{
    char message[80];

    axis(xleft, ybottom, xright, ytop);
    sprintf(message, "heat equation, u(0)=u(1)=0");
    fsymbol(X(0.2), Y(0.95), message);
    sprintf(message,
    "nfunc=%d, theta=%g, N=%d, lambda=%g, tau=%g, Tmax=%g, dt=%g",
    nfunc, theta, N, lambda, tau, Tmax, dt);
    fsymbol(X(0.2), Y(0.9), message);
    sprintf(message, "t = %g", t);
    if (t != 0.0)
fsymbol(X(0.2), Y(0.85), message);
}

/* 現在ウィンドウに表示されている図をファイルに保存する */
void save_picture()
{
    char filename[200];
    printf("図を保存するファイル名: ");
    scanf("%s", filename);
    mkplot(filename);
}

/************************************************************
 ************************************************************
 ************************************************************/

/*
 *  三重対角行列に対する LU 分解に基づく連立 1 次方程式の求解
```

```
 *
 *  (Gauss の消去法を用いているが、ピボットの選択等はしていないので、
 *   係数行列が正定値対称行列でないと結果は保証されない。)
 *
 *  n は未知数の個数。
 *  f は最初は連立 1 次方程式の右辺のベクトル b。最後は解ベクトル x。
 *   (添字は 0 から。i.e. b[0],b[1],...,b[n-1] にデータが入っている。)
 *
 *  AL, AD, AU は係数行列の
 *     下三角部分 (lower part)
 *     対角部分 (diagonal part)
 *     上三角部分 (upper part)
 *  つまり
 *
 *  AD[0] AU[0]   0   ................... 0
 *  AL[1] AD[1] AU[1]   0  .............. 0
 *    0   AL[2] AD[2] AU[2]  0 ......... 0
 *          ...................
 *                     AL[n-2] AD[n-2] AU[n-2]
 *                       0     AL[n-1] AD[n-1]
 *
 *  ここで AL[0], AU[n-1] は意味がないことに注意。
 */

/* 三項方程式 (係数行列が三重対角である連立一次方程式のこと) を解く
 *   入力
 *     n: 未知数の個数
 *     al,ad,au: 連立一次方程式の係数行列
 *       (al: 対角線の下側、 ad: 対角線、 au: 対角線の上側)
 *        al[i] = A_{i,i-1}, ad[i] = A_{i,i}, au[i] = A_{i,i+1},
 *        al[0], au[n-1] は意味がない)
 *        f: 連立一次方程式の右辺の既知ベクトル b
 *   出力
 *     al,ad,au: 入力した係数行列を LU 分解したもの
 *     f: 連立一次方程式の解 x
 *   能書き
 *     一度 call すると係数行列を LU 分解したものが返されるので、
 *     以後は同じ係数行列に関する連立一次方程式を解くために、
 *     サブルーチン trisol が使える。
 *   注意
 *     ピボットの選択をしていないので、係数行列が正定値である
 *     などの適切な条件がない場合は結果が保証できない。
 */

/* 三項方程式を解く */
void trid(int n, double *al, double *ad, double *au, double *f)
{
    trilu(n, al, ad, au);
    trisol(n, al, ad, au, f);
}

/* 三重対角行列の LU 分解 (pivoting なし) */
void trilu(int n, double *al, double *ad, double *au)
{
    int i, nm1 = n - 1;
    /* 前進消去 (forward elimination) */
    for (i = 0; i < nm1; i++)
ad[i + 1] -= au[i] * al[i + 1] / ad[i];
```

```
}

/* LU 分解済みの三重対角行列を係数に持つ三項方程式を解く */
void trisol(int n, double *al, double *ad, double *au, double *f)
{
    int i, nm1 = n - 1;
    /* 前進消去 (forward elimination) */
    for (i = 0; i < nm1; i++)
f[i + 1] -= f[i] * al[i + 1] / ad[i];
    /* 後退代入 (backward substitution) */
    f[nm1] /= ad[nm1];
    for (i = n - 2; i >= 0; i--)
f[i] = (f[i] - au[i] * f[i + 1]) / ad[i];
}
```



3.6 ccx, f77x の正体

ともにシェル・スクリプトである。

/usr/local/bin/f77x

```
#!/bin/csh
f77 -O -o $1:r $* -L /usr/local/lib -R /usr/local/lib -lfplot -lX11 -lsocket -lnls
```

/usr/local/bin/ccx

```
#!/bin/csh
gcc -O -I/usr/local/include -o $1:r $* -L /usr/local/lib -L /usr/local/X11R6.3/lib -R /u
```

3.7 TEX への取り込み

1. 関数 mkplot() を使って、plot(5) フォーマットで記録する。

```
        mkplot("mygraph.plot");
```

2. plot2ps コマンドを使って、PostScript ファイルに変換する。

```
oyabun% plot2ps mygraph.plot > mygraph.ps
```

3. /usr/local/meiji/bin/toeps コマンドを使って、BoundingBox を埋め込む。

```
oyabun% toeps mygraph.ps
```

4. LaTeX では、eclepsf スタイルを読み込む。

   LaTeX 209 の場合 \documentstyle[eclepsf] **のようにする。**

   LaTeX 2$_\varepsilon$ の場合 \usepackage[eclepsf] **のようにする。**

5. 図を取り込みたいところで \epsfile{file=ファイル名}

```
 \includegraphics[width=8cm]{"program3/mygraph.ps"}
```

3.8 ccg の関数 fsymbol() の実現法

東海林先生の講義で使った ccg コマンドでリンクされるライブラリィには、fsymbol() という関数があるが、ccx にはない。これが使いたい場合は、次のようにすれば良い。

```
fsymbol(x, y, s)
double x, y;
char s[];
{
    fmove(x, y); label(s);
}
```

第4章　数値計算プログラミング

(準備中)

第5章　グラフを描こう (2) GLSC

5.1　はじめに — GLSC とは

GLSC (Graphic Library for Scientific Computing) とは、龍谷大学数理情報学科のグループ[1]が作成した、科学技術計算の結果を可視化するためのライブラリィです[2]。

ワークステーション環境 (具体的には UNIX + X) で、C や Fortran で図を描く手段には色々なものがありますが、GLSC は実際に数値解析の分野で仕事をしてきた人達が作ったものだけに、特定の目的には本当に便利なものだと筆者は思っています。具体的には、2 変数関数の可視化 (等高線、鳥瞰図(ちょうかんず)の描画) のためのサブルーチン、関数が揃っています。

マニュアル (PostScript フォーマット) も公開されていて、それを PDF に変換したものを

http://www.math.meiji.ac.jp/~mk/labo/howto/GLSC.pdf

においてあります (まだ機械的に PDF に変換しただけで、何も手を入れていませんが、ないよりはましでしょう)。

glscwin — Windows 版 GLSC

オリジナルの開発者の一人である高橋大輔氏の研究室によって、Win32 環境[3]に移植されたようです。以下の URL を参照してください。

http://www.mpdt.math.waseda.ac.jp/glscwin/index.html

なお、そのマニュアル

http://www.mpdt.math.waseda.ac.jp/glscwin/glscmm.html

はオリジナルの GLSC を使う場合にも役立つと思われます。

5.2　6701 号室では？

数学科計算機室にある Solaris 2.6 で動いているワークステーションに GLSC がインストールされています。具体的には

[1] 小林亮、高橋大輔、中野浩、松木平淳太。

[2] `ftp.st.ryukoku.ac.jp/pub/ryukoku/software/math/` から ftp で入手できます。

[3] 具体的な OS の名前で言うと、Windows 95, Windows 98, Windows Me, Windows NT, Windows 2000 のことを指します。

インクルード・ファイル　/usr/local/include/glsc.h
インクルード・ファイル　/usr/local/include/glsc_ftn.h
ライブラリィ・アーカイブ/usr/local/lib/libglscd.a
ライブラリィ・アーカイブ/usr/local/lib/libglscs.a

という 4 つのファイルです。
また、PostScript 形式のマニュアル

/usr/local/meiji/text/ryukoku/glsc/part1.ps
/usr/local/meiji/text/ryukoku/glsc/part2.ps

があります。印刷するには、例えば

両面印刷する

```
oyabun% cat part1.ps | double | lp -dlp2
```

のようにします。

プログラムの書き方 (1) 詳しくは GLSC のマニュアルを読むべきですが[4]、インクルード・ファイルの読み込みと、C や C++ の場合の浮動小数点数の宣言の問題を解決するおまじないについて説明しておきます。

(i) C の場合は、"#define G_DOUBLE"、それに続けて "#include <glsc.h>" とする。

(ii) ANSI C の場合も C と同様で良い。

(iii) C++ の場合も "#define G_DOUBLE"、それに続けて

```
extern "C" {
#include <glsc.h>
}
```

とする (まあ、C++ で C のファイルをインクルードするときの定跡ですが)。

(iv) UNIX 伝統の FORTRAN (f77) の場合は、

```
include '/usr/include/glsc_ftn.h'
```

とする。

プログラムの書き方 (2) 以下述べることはどの言語にも共通しています。

(i) 最初に "g_init("*metafilename*", ウィンドウの横幅, ウィンドウの高さ)" を呼び出します (サイズの単位は mm で、引数の型は浮動小数点型です)。メタファイルというのは、描画した図形を記録するためのファイルのことです。

[4]GLSC のマニュアルにはサンプル・プログラムのソース・ファイルと実行結果も載っています。また、これらサンプル・プログラムのファイルが http://www.math.meiji.ac.jp/~mk/labo/glsc-sample/ に置いてあります。

(ii) 出力デバイスを “g_device(出力先)” 呼び出しで指定します。G_BOTH とすると、画面とメタファイルの両方に出力するようになります。

(iii) “g_def_scale()” で座標系を定義します。座標系は複数個定義できて、以下は番号を使って g_sel_scale(番号); として指定できます。
→ GLSC の中で、マルチ・ウィンドウもどきが簡単に実現できます。

(iv) “g_def_line()” を呼び出して、使用する線 (色、太さ、線種) を定義します。複数の線が定義できて、後から番号で呼び出せます。

(v) “g_cls()” で画面のクリアをします。

(vi) 既に “g_def_ほげほげ” で定義したものを “g_sel_ほげほげ” で指定します。

(vii) 色々な描画命令を並べます。

(viii) “g_sleep()” で停止します。

(ix) “g_term()” で GLSC を終了します。

コンパイルの仕方 最もフツーのコンパイルの仕方。

倍精度の場合 “-I/usr/local/include” と “-lglscd -lX11 -lsocket -lm” を指定するのが基本です[5]。

(i) gcc の場合は
gcc -o *myprog* -I/usr/local/include *myprog*.c -lglscd -lX11 -lsocket -lm

(ii) g++ の場合は
g++ -o *myprog* -I/usr/local/include *myprog*.C -lglscd -lX11 -lsocket -lm

(iii) FORTRAN の場合は “f77 -o *myprog* *myprog*.f -lglscd -lX11”

いずれも少々面倒なので、エイリアスを定義したり、Makefile を作ったりして、手間を軽減すべきでしょう。筆者は次のような行を .cshrc に書き加えています。

```
alias ccmg 'gcc -O -o \!^:r -I/usr/local/include \!* -lmatrix -lglscd -lX11
-lsocket -lm'
```

単精度の場合 “-lglscd” の代わりに “-lglscs” を指定するのが基本です。

印刷の仕方 g_device() で、G_BOTH など、メタファイルを出力する設定をしておきます。出来上がったメタファイルの名前を Meta とすると、

```
oyabun% g_out -v Meta
```

として Meta.ps という PostScript 形式のデータが出来ます。後は

再表示 gs Meta.ps あるいは ghostview Meta.ps & あるいは gv Meta.ps &
印刷 lp Meta.ps

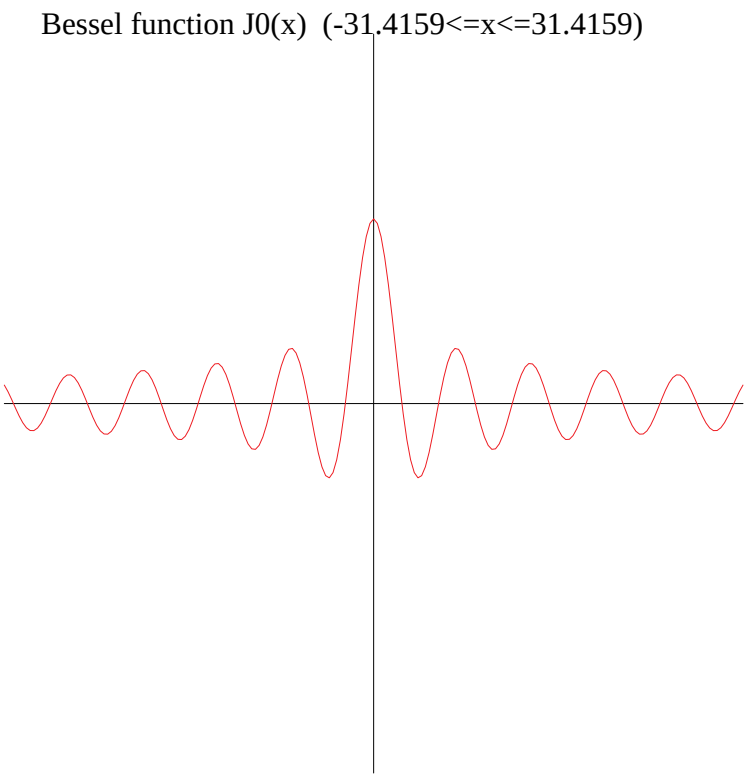
カラーのまま印刷したい場合は、-i というオプションをつけて Illustrator 形式に変換します。その場合、Meta.i00 のようなファイル名になります。

[5] コンパイル・オプション “-l ほげほげ” で lib ほげほげ.a をリンクすることになります。ここでは libglscd.a, libX11.a, libsocket.a, libm.a をリンクするわけです。

5.3 GLSC のサンプル・プログラム

5.3.1 何をするプログラムか

ありきたりですが、1 変数関数のグラフを描くプログラム例を示します[6]。0 次 Bessel 関数 `j0()` の $-10\pi \leq x \leq 10\pi$ の範囲のグラフを描きます。



5.3.2 ソースプログラム `draw-graph.c`

以下のプログラムは `http://www.math.meiji.ac.jp/~mk/labo/howto/glsc-progs/` に置いてあります。

```
/*
 * draw-graph.c -- 1 変数関数のグラフを描く
 *   コンパイル:  ccmg draw-graph.c
 */


#include <stdio.h>
#include <math.h>

#define G_DOUBLE
#include <glsc.h>
```

[6] グラフを描く目的でしたら、gnuplot などを使う方が便利です。あくまでも GLSC の解説用のプログラムです。

```
double pi;

int main()
{
  int i, n;
  double a, b, c, d;
  double h, x;
  double f(double);
  char title[100];
  double win_width, win_height, w_margin, h_margin;

  pi = 4 * atan(1.0);

  /* 表示する範囲 [a,b] × [c,d] を決定 */
  a = - 10 * pi; b = 10 * pi; c = - 2.0; d = 2.0;

  /* 区間の分割数 n */
  n = 200;

  /* GLSC の開始
     メタファイル名、ウィンドウ・サイズの決定 */
  win_width = 200.0; win_height = 200.0; w_margin = 10.0; h_margin = 10.0;
  g_init("GRAPH", win_width  + 2 * w_margin, win_height + 2 * h_margin);

  /* 出力デバイスの決定 */
  g_device(G_BOTH);

  /* 座標系の定義: [a,b] × [c,d] という閉領域を表示する */
  g_def_scale(0,
              a, b, c, d,
              w_margin, h_margin, win_width, win_height);

  /* 線を二種類用意する */
  g_def_line(0, G_BLACK, 2, G_LINE_SOLID);
  g_def_line(1, G_RED,   0, G_LINE_SOLID);

  /* 表示するための文字列の属性を定義する */
  g_def_text(0, G_BLACK, 3);

  /* 定義したものを選択する */
  g_sel_scale(0); g_sel_line(0); g_sel_text(0);

  /* 座標軸を描く */
  g_move(a, 0.0);  g_plot(b, 0.0);
  g_move(0.0, c); g_plot(0.0, d);

  /* タイトルを表示する */
  sprintf(title, "Bessel function J0(x)  (%g<=x<=%g)", a, b);
  g_text(20.0, 10.0, title);

  /* 刻み幅 */
  h = (b - a) / n;
  /* グラフを描くための線種の選択 */
  g_sel_line(1);
  /* 折れ線でグラフを描く */
  g_move(a, f(a));
  for (i = 1; i <= n; i++) {
    x = a + i * h;
```

```
    g_plot(x, f(x));
  }

  /* ユーザーのマウス入力を待つ */
  printf("終りました。X の場合はウィンドウをクリックして下さい。\n");
  g_sleep(-1.0);
  /* ウィンドウを閉じる */
  g_term();
  return 0;
}

double f(double x)
{
  /* 0 次 Bessel 関数 */
  return j0(x);
}
```

5.3.3 読んでみよう

どれが GLSC の命令か？

GLSC の関数はすべて、名前の先頭が g_ となっています。g_init(), g_device(), g_def_scale(), g_def_line(), g_def_text(), g_sel_scale(), g_sel_line(), g_sel_text(), g_text(), g_move(), g_plot(), g_sleep(), g_term() という関数を使っています。

ウィンドウのサイズ

ウィンドウのサイズは直接数値を書き込むと分かりづらくなるので、win_width + 2 * w_margin, win_height + 2 * h_margin のような式で表わしました。ここで win_width, win_height はそれぞれ表示領域の幅と高さで、w_margin, h_height はウィンドウの縁に用意する余白 (マージン) の大きさです。具体的な値は

```
win_width = 200.0; win_height = 200.0; w_margin = 10.0; h_margin = 10.0;
```

として与えています。こうして定めたウィンドウのサイズを

```
g_init("GRAPH", win_width  + 2 * w_margin, win_height + 2 * h_margin);
```

として GLSC に指示しています。ここで "GRAPH" は「メタファイル」の名前です。これは描画した図形の情報をファイルに記録する場合にはファイル名として採用されるものです。

出力先

g_device() によって、どこに出力するか指定します。選択肢は G_NONE (出力しない), G_DISP (ディスプレイ), G_META (メタファイル), G_BOTH (ディスプレイとメタファイルの両方) の 4 つです。サンプル・プログラムでは

```
g_device("G_BOTH");
```

として、画面にも表示するし、ファイルにも記録するように指定しています。

座標系の定義

「表示したい対象物の世界」における座標と出力デバイス (画面等) の座標との関係を定義する必要がありますが、これは、辺が座標軸に平行な長方形 (表示領域) を、二つの世界それぞれにおいて指定することで実現されます。

出力デバイスにおける長方形は、ウィンドウの左上隅の頂点からの余白 (水平方向, 垂直方向) と、長方形の辺の長さ (横、縦) の 4 つの数値を使って、指定されます。サンプル・プログラムでは、これらの値は w_margin, h_margin, win_width, win_height という変数に記憶されています。

「表示したい対象物の世界」における長方形は、プログラムの中の変数 a, b, c, d を用いて定義できる $[a, b] \times [c, d]$ です。

これらの情報を関数 g_def_scale() に渡すことになります。

```
/* 座標系の定義: [a,b] × [c,d] という閉領域を表示する */
g_def_scale(0,
            a, b, c, d,
            w_margin, h_margin, win_width, win_height);
```

第 1 の引数である 0 は、座標系の番号を表わします。実は GLSC では、一度に複数の座標系を定義しておいて、必要に応じて番号を使って呼び出すことができるようになっています。

線種、文字種の定義

描画に用いる線には、色、太さ、パターン (実線なのか点線なのか破線なのか) などの属性を持っています。GLSC では、これらの属性を備えた線種を定義することができます。

```
/* 線を二種類用意する */
g_def_line(0, G_BLACK, 2, G_LINE_SOLID);
g_def_line(1, G_RED,   0, G_LINE_SOLID);
```

同様に文字列を表示する場合の文字も、色と大きさの属性を持ち得ます。

```
/* 表示するための文字列の属性を定義する */
g_def_text(0, G_BLACK, 3);
```

定義しておいた座標系、線種、文字の呼び出し

これは簡単で g_sel_某 () という関数に番号を与えて呼び出すだけです。

```
/* 定義したものを選択する */
g_sel_scale(0); g_sel_line(0); g_sel_text(0);
```

線分を描く、グラフを描く

GLSC では、XY プロッター[7]風の、「現在点から指定した点までの線分を引く」という機能を持った関数 g_plot(double x, double y) が用意されています。線を描かずに現在点のみを移動する機能の関数 g_move(double x, double y) と一緒に使うことで、自由に線分が描けます。

例えば x 軸を表示する目的で、サンプル・プログラムでは二点 $(a,0), (b,0)$ を端点とする線分を

```
g_move(a, 0.0);  g_plot(b, 0.0);
```

として描いています。

g_move(), g_plot() のような関数があるときに、1 変数のグラフを描く手順は、次のようになります (定跡的であると言って良いでしょう)。

```
/* 折れ線でグラフを描く */
g_move(a, f(a));
for (i = 1; i <= n; i++) {
  x = a + i * h;
  g_plot(x, f(x));
}
```

マウスの入力待ち

何かを見せる目的のプログラムでは、ユーザーが好きな間だけ見ていられるように、他に何もしないで「待つ」ことが必要になります。GLSC ではそのために g_sleep(double t) という関数が用意されています。

```
g_sleep(-1.0);
```

のように t として負の数を与えると、「ユーザーが GLSC のウィンドウをマウスでクリックされるまで待つ」ことになります。

終了 (ウィンドウの削除)

描画用のウィンドウを削除するには単に

[7]今では XY プロッターと言っても目にする機会がなくなってしまったが、紙の上にペンをアームで動かすことにより、2 次元の線画を描く機械であった。現在ペンのある位置 (2 次元的な) が「現在点」のわけである。

```
g_term();
```

とするだけです。

5.4 GLSC+ について

5.4.1 その目的

当数学科での GLSC 利用の「歴史」も結構長くなって来ました。最初のうちは GLSC をオリジナルのままに使っていたのですが、しばしば等高線や鳥瞰図を描く関数が使いづらいと感じていました。その理由は、GLSC が C 言語で書かれているため、Fortran のような整合配列が利用できないためと言えるでしょう。格子点上の数値データや行列のような二重添字を持つ数列を二次元配列ではなく、長い 1 次元配列に (ユーザーが自分の責任で) 詰め込んで扱うのは、本質的でない複雑さが生じます。

そこで、長い 1 次元配列の代わりに

```
typedef G_REAL **matrix;
```

として定義した、G_REAL へのポインターのポインター (matrix 型) を使ったインターフェイス (C 言語用) を用意しました。

5.4.2 新しく導入した関数

とにかく 2 変数関数のグラフの鳥瞰図と等高線が描ければよい、と考えたので、次の 3 つだけです。

1. **g_hidden2()**
2. **g_fake_hidden2()**
3. **g_contln2()**

使い方は名前の末尾に 2 がついていない関数とほとんど同じです。ただ一点 G_REAL へのポインターでなく、G_REAL へのポインターのポインターを引数として受け取るところが異なります。

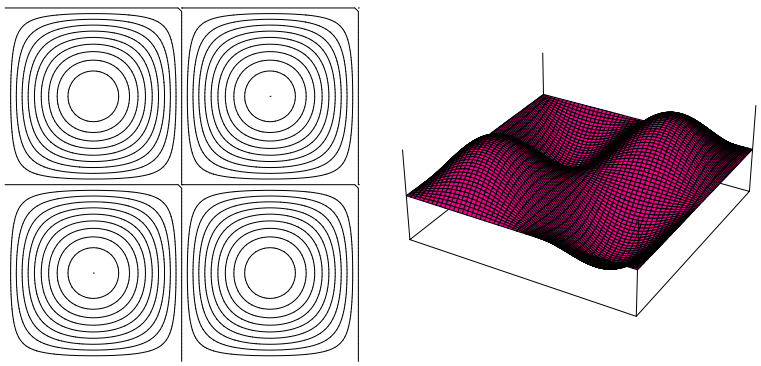
5.4.3 サンプル・プログラム・ソース

```
/*
 * test-contln2.c
 *   cc test-contln2.c -lmatrix -lglscd -lX11 -lm
 */

#define G_DOUBLE
#include <stdio.h>
#include <math.h>
```

```
#include "glsc.h"
#include <matrix.h>

#define W0 80.0
#define H0 80.0
#define W1 80.0
#define H1 80.0
#define W_MARGIN 10.0
#define H_MARGIN 10.0

double pi;

void compute(double (*)(), matrix,
        double, double, double, double,
        int, int);
double f(double, double);

double max(double x, double y) { return (x > y) ? x : y; }

int main()
{
    int m, n, k;
    double xmin, xmax, ymin, ymax, f();
    matrix u;

    pi = 4 * atan(1.0);

    /* 分割数 */
    m = 100; n = 100;
    /* 定義域は [-π , π ] × [-π , π ] */
    xmin = - pi; xmax = pi; ymin = - pi; ymax = pi;

    /* 格子点における数値を納める変数 */
    if ((u = new_matrix(m+1,n+1)) == NULL) {
        fprintf(stderr, " 行列のためのメモリーが確保できませんでした。\n");
        return 1;
    }
    /* GLSC */
    g_init("Meta", W0 + W1 + 3 * W_MARGIN, max(H0, H1) + 2 * H_MARGIN);
    g_device(G_BOTH);
    /* ウィンドウ 0 */
    g_def_scale(0,
                xmin, xmax, ymin, ymax,
                W_MARGIN, H_MARGIN, W0, H0);
    g_def_scale(1,
                xmin, xmax, ymin, ymax,
                W_MARGIN + W0 + W_MARGIN, H_MARGIN, W1, H1);
    /* */
    g_def_line(0, G_BLACK, 0, G_LINE_SOLID);
    g_def_text(0, G_BLACK, 2);
    /* 定義したものを呼び出す */
    g_sel_scale(0);
    g_sel_line(0);
    g_sel_text(0);
    /* title */
    g_text(W_MARGIN + W0 * 0.6, H_MARGIN / 2, "contour and bird view");
    /* 格子点上での関数値を計算する */
    compute(f, u, xmin, xmax, ymin, ymax, m, n);
```

```
    /* 等高線 */
    for (k = -10; k <= 10; k++)
        g_contln2(xmin, xmax, ymin, ymax, u, m+1, n+1, 0.1 * k);
    /* 鳥瞰図 */
    g_hidden2(1.0, 1.0, 0.4,
              -1.0, 1.0,
              /* 視点（距離，方角を表わす（θ，φ））*/
              5.0, 30.0, 30.0,
              W_MARGIN + W0 + W_MARGIN, H_MARGIN,
              W1, H1,
              u, m + 1, n + 1, 1,
              G_SIDE_NONE, 2, 1);

    printf("終了したらウィンドウをクリックして終了してください。\n");
    g_sleep(-1.0);
    g_term();

    return 0;
}

/*
 * [xmin,xmax]× [ymin,ymax] を x 軸方向に m 等分、y 軸方向に n 等分して
 * 各格子点上の f の値を u に格納する。
 */
void compute(double (*f)(), matrix u,
        double xmin, double xmax, double ymin, double ymax,
        int m, int n)
{
    int i, j;
    double dx, dy, x, y;

    dx = (xmax - xmin) / m;
    dy = (ymax - ymin) / n;
    for (i = 0; i <= m; i++) {
        x = xmin + i * dx;
        for (j = 0; j <= n; j++) {
            y = ymin + j * dy;
            u[i][j] = f(x, y);
        }
    }
}

double f(double x, double y)
{
    return sin(x) * sin(y);
}
```

contour and bird view



.1 よく使う関数の説明

g_init

書式 g_init(char *filename, G_REAL window_width, G_REAL window_height)

機能 GLSC の初期化を行なう。
filename はメタファイルの名前。*window_width*, *window_height* はウィンドウのサイズ (単位は mm)。

例
```
g_init("Meta", 340.0, 220.0);
```

g_device

書式 g_device(int device)

機能 出力先デバイスを指定する。
device は出力先を指定する数値。以下の定数がインクルード・ファイルに定義されている。

G_NONE	出力しない
G_META	メタファイル
G_DISP	画面
G_BOTH	画面とメタファイル

例
```
g_device(G_BOTH);
```

g_def_scale

書式 g_def_scale(int scale_id,
G_REAL x_left, G_REAL x_right, G_REAL y_bottom, G_REAL y_top,
G_REAL left_margin, G_REAL top_margin, G_REAL width, G_REAL height)

機能 座標変換を定義する。
scale_id は座標変換につける番号 (複数定義して、番号で区別することができる)。ユーザー座標系の $[x_{\mathrm{left}}, x_{\mathrm{right}}] \times [y_{\mathrm{bottom}}, y_{\mathrm{top}}]$ という長方形閉領域を $[\mathrm{left_margin}, \mathrm{left_margin} + \mathrm{width}] \times [\mathrm{top_margin}, \mathrm{top_margin} + \mathrm{height}]$ という長方形閉領域に写像する。

例
```
g_def_scale(0, -1.6, 1.6, -1.0, 1.0,
                  10.0, 10.0, 320.0, 200.0);
```

g_def_line

書式 `g_def_line(int line_id, int color_id, int line_width, int line_type)`

機能 線 (というよりはペン) を定義する。
line_id は線につける番号 (複数定義して、番号で区別することができる)。色が `color_id`、太さが `line_width`, 種類が `line_type` の線に `line_id` という番号をつける。

- 色としては

G_BLACK	黒	0
G_RED	赤	1
G_GREEN	緑	2
G_BLUE	青	3
G_MAGENTA	マゼンタ (赤紫)	4
G_YELLOW	黄	5
G_CYAN	シアン (澄んだ青緑色、赤の補色)	6
G_WHITE	白	7

が使える。

- 線の太さは 0 から 3 まで。0 と 1 は太さ同じだが 0 の方が速い。
- 線種としては

G_LINE_SOLID	実線
G_LINE_DOTS	
G_LINE_DASHED	
G_LINE_LONG_DASHED	
G_LINE_THIN_DOTS	
G_LINE_DOT_DASHED	
G_LINE_D_DOT_DASHED	

例
```
g_def_line(0, G_BLACK, 0, G_LINE_SOLID);
```

g_cls

書式 `g_cls()`

機能 画面の消去を行なう (これまでに描かれたものを削除する)。

例
```
g_cls();
```

g_sleep

書式 g_sleep(G_REAL time)

機能 time が正の場合、time 秒停止する。time が負の場合、マウスをクリックするまで停止する。

例
```
g_sleep(-1.0); /* マウスをクリックするまで待つ */
```

g_term

書式 g_term()

機能 GLSC を終了する。

例
```
g_term();
```

.2 PostScript への変換 (g_out に関するノウハウ)

関数 g_init() の第一引数で指定した「メタファイル」には、描画した図形データの内容が記録されていて、コマンド g_out により、PostScript 形式に変換できる。

-v を指定するとポートレイト形式になる。

出来上がったファイルの BounddingBox コメントはかなりいいかげんなので[8]、toeps コマンド[9]で直すか、ghostview などで表示させて測った値をテキスト・エディターで書き込むのがよい。

特に -i オプションを指定すると、Adobe のイラストレーター形式でセーブされるという。これも一種の PostScript であることに違いはないが、カラーがモノクロに変換されることがないので、色つきの画像データを作るときに使える。ただし、この形式でセーブしたとき、toeps コマンドで BoundingBox を書き込むことがなぜか失敗する。

線の太さや文字の大きさなどの情報は PostScript に変換すると消えてしまうが、例えば使用している文字をすべて大きくして構わないのならば、直接 PostScript ファイルを編集して直すことが比較的容易である。

熱方程式や波動方程式のような発展系の数値計算結果を可視化した場合、一つのメタファイルに複数の図が記録される。このとき、一枚の紙に複数の図を連ねて表示する「紙芝居」を作るには、-f 行の数, 列の数 と -m 倍率 というオプションを使うとよい。例えば

```
g_out -vfm 4,3 0.4 Meta
```

とすると、1 ページあたり 4 段 (行？) 3 列、全部で 12 の図が入った PostScript ファイルが出来ます。

[8] これは筆者が -v オプションを使っているせいかもしれない。
[9] これは明治大学数学科特有のコマンド。

.3 桂田研学生向け

桂田研の学生にはおなじみの heat1d-e.c の GLSC 版です。

```
/*
 * heat1d-e-glsc.c -- 1 次元熱伝導方程式の初期値境界値問題を陽解法で解く。
 *   コンパイル:  ccmg heat1d-e.c
 *
 * オリジナルは fplot ライブラリィを利用した
 *  http://www.meiji.ac.jp/servers/math/staffs/mk/program/fdm/heat1d-e.c
 */

#include <stdio.h>
#include <math.h>

#define G_DOUBLE
#include <glsc.h>

main()
{
  int i, n, nMax, N;
  double tau, h, lambda, Tmax;
  double *u, *newu;
  double f(double);
  double win_width, win_height, w_margin, h_margin;
  char message[100];

  /* N, λ  を入力する */
  printf("区間の分割数 N = "); scanf("%d", &N);
  printf("λ  (=τ /h^2) = "); scanf("%lf", &lambda);

  /* h, τ  を計算する */
  h = 1.0 / N;
  tau = lambda * h * h;
  printf("τ =%g\n", tau);

  /* 最終時刻を入力する */
  printf("最終時刻 Tmax = "); scanf("%lf", &Tmax);

  /* ベクトル u, newu を用意する */
  u = (double *)malloc(sizeof(double) * (N+1));
  newu = (double *)malloc(sizeof(double) * (N+1));

  /* 初期値の代入 */
  for (i = 0; i <= N; i++)
    u[i] = f(i * h);

  /* ***************** グラフィックスの準備 ***************** */
  /* メタファイル名は "HEAT",
   * ウィンドウのサイズは、
       横 win_width  + 2 * w_margin, 縦 win_height + 2 * h_margin */
  win_width = 200.0; win_height = 200.0; w_margin = 10.0; h_margin = 10.0;
  g_init("HEAT", win_width  + 2 * w_margin, win_height + 2 * h_margin);
  /* 画面とメタファイルの両方に記録する */
  g_device(G_BOTH);
  /* 座標系の定義: [-0.1,1.1]× [-0.1,1.1] という閉領域を表示する */
  g_def_scale(0,
```

```
              -0.1, 1.1, -0.1, 1.1,
              w_margin, h_margin, win_width, win_height);
  /* 線を二種類用意する */
  g_def_line(0, G_BLACK, 0, G_LINE_SOLID);
  g_def_line(1, G_BLACK, 0, G_LINE_DOTS);
  /* 表示するための文字列の属性を定義する */
  g_def_text(0, G_BLACK, 3);
  /* 定義したものを選択する */
  g_sel_scale(0); g_sel_line(0); g_sel_text(0);

  /* タイトルと入力パラメーターを表示する */
  g_text(30.0, 30.0,
         "heat equation, homogeneous Dirichlet boundary condition");
  sprintf(message, "N=%d, lambda=%g, Tmax=%g", N, lambda, Tmax);
  g_text(30.0, 60.0, message);

  /* 座標軸を表示する */
  g_sel_line(1);
  g_move(-0.1, 0.0); g_plot(1.1, 0.0);
  g_move(0.0, -0.1); g_plot(0.0, 1.1);
  g_sel_line(0);

  /* t=0 の状態を表示する */
  g_move(0.0, u[0]);
  for (i = 1; i <= N; i++)
    g_plot(i * h, u[i]);

  /* ループを何回まわるか計算する（四捨五入）*/
  nMax = rint(Tmax / tau);

  /* 時間に関するステップを進めるループ */
  for (n = 0; n < nMax; n++) {
    /* 差分方程式 (n -> n+1) */
    for (i = 1; i < N; i++)
      newu[i] = (1.0 - 2 * lambda) * u[i] + lambda * (u[i+1] + u[i-1]);
    /* 計算値を更新 */
    for (i = 1; i < N; i++)
      u[i] = newu[i];
    /* Dirichlet 境界条件 */
    u[0] = u[N] = 0.0;
    /* この時刻 (t=(n+1)τ ) の状態を表示する */
    g_move(0.0, u[0]);
    for (i = 1; i <= N; i++)
      g_plot(i * h, u[i]);
  }

  printf("終りました。X の場合はウィンドウをクリックして下さい。\n");
  g_sleep(-1.0);
  /* ウィンドウを閉じる */
  g_term();
  return 0;
}

double f(double x)
{
  if (x <= 0.5)
    return x;
  else
```

```
    return 1.0 - x;
}
```

.4 Cygwin+XFree86 環境での利用

個人的に GLSC がすごいと思うのは、X の基本的な機能しか使っていないので (その点非常に禁欲的です)、様々なシステム (GLSC の開発後に登場してきたものでも) に対して、無修整あるいは非常に微弱な修正で利用できるようになることです。

最近、普及してきた Cygwin+XFree86 という環境でもごく簡単に利用できます。自力で make するのも簡単ですが、バイナリーを用意しました。http://www.math.meiji.ac.jp/~mk/labo/cygwin/cygwin-glsc+.tar.gz から cygwin-glsc+.tar.gz を入手して、

```
tar xzf cygwin-glsc+.tar.gz -C /usr/local
```

のように展開すれば使えるようになります (ヘッダー・ファイルを /usr/local/include, ライブラリィ・ファイルを /usr/local/lib に格納します。コンパイルは /usr/local/bin/ccmg というスクリプトで可能です。スクリプトを見ればコンパイル&リンクの仕方が分かるでしょう)。

付 録A　常微分方程式

http://www.math.meiji.ac.jp/~mk/labo/text/num-ode においておきます。

付 録B　常微分方程式の初期値問題 — とにかく始めてみよう

(ある日の桂田ゼミから)

B.1　はじめに

数学科の学生向けに、常微分方程式の初期値問題の数値解法について解説する。数学の学習・研究の過程で現れる様々な問題を、とりあえず実際にコンピューターを使って解けるようになってもらうことが目標である。

B.2　目的とする問題

常微分方程式の初期値問題

$$\frac{dx}{dt} = f(t,x) \tag{B.1}$$
$$x(a) = x_0 \tag{B.2}$$

を考える。つまり f, a, x_0 が与えられたとき、(B.1), (B.2) を満たす未知関数関数 $x = x(t)$ を求める、ということである。

詳しく言うと、

- f は $\mathbf{R}^{n+1}$ のある開集合 Ω 上定義され、$\mathbf{R}^n$ に値を取る関数である: $f\colon \Omega \to \mathbf{R}^n$
- $x_0 \in \mathbf{R}^n$
- $(a, x_0) \in \Omega$

この問題に対して、解の存在や一意性などの基本的なことは十分に分かっていると言って良い (これについては、大抵の常微分方程式の数学科向けのテキストに解説がある)。

一方、この問題は、特別な f に対してしか、具体的に解けないことが良く知られている。例えば三体問題は歴史上重要なものとして精力的に研究されたが、結局求積法では解けないことが証明された。

そこで、数値解法の出番となる。

B.3 離散変数法

常微分方程式の初期値問題の数値解法には色々あるが、ここでは離散変数法と総称される「メジャーな」方法を紹介する。

離散変数法では、$[a,b]$ における解 x を求めたいとき、区間 $[a,b]$ を

$$a = t_0 < t_1 < t_2 < \cdots < t_{N-1} < t_N = b \tag{B.3}$$

と分割し、各分点 t_j における解 x の値 $x(t_j)$ の近似値 (以下でそれを x_j と書く) を求めることを目標とする[1]。

分点は、特に理由がなければ N 等分点にとる。すなわち

$$h = \frac{b-a}{N}$$

として

$$t_j = a + jh \quad (j = 0, 1, 2, \cdots, N)$$

とする。

B.4 Euler 法

微分係数の定義より、h が十分小さければ

$$\begin{aligned} \frac{dx}{dt}(t_j) &= \lim_{\varepsilon \to 0} \frac{x(t_j+e) - x(t_j)}{\varepsilon} \\ &\fallingdotseq \frac{x(t_j+h) - x(t_j)}{h} \end{aligned}$$

と考えることが出来る。

そこで

$$\frac{dx}{dt}(t_j) = f(t_j, x(t_j))$$

から $\{x_j\}$ に関する方程式

$$\boxed{\frac{x_{j+1} - x_j}{h} = f(t_j, x_j)} \tag{B.4}$$

を得る (正確にはこの方程式の解として $\{x_j\}$ を定義する、わけである)。

(B.4) を整理して、

$$x_{j+1} = x_j + hf(t_j, x_j) \tag{B.5}$$

なる「隣接二項」の漸化式を得る。x_0 は分かっているわけだから、これから x_1, x_2, $\cdots$, x_N を順番に計算できる。

以上が **Euler** 法である。Euler 法は素朴であるが、次の意味で「うまく働く」。

f に Lipschitz 連続程度の滑らかさがあれば、
(t_j, x_j) を結んで出来る折れ線をグラフとする関数は、
$N \to \infty$ とするとき、真の解に収束する。

しかし、実は Euler 法はあまり効率的ではないため、実際に使われることはまれである。

[1] つまり、変数 t の離散的な値に対する解の値のみを求める、という意味で「離散変数法」なわけ。このように目標を低く設定することによって、無限次元の問題が有限次元の問題に簡略化されていると言える。

B.5 Runge-Kutta 法

漸化式

$$x_{j+1} = x_j + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}, \tag{B.6}$$

ただし、

$$\begin{cases} k_1 &= h\, f(t_j, x_j) \\ k_2 &= h\, f(t_j + h/2, x_j + k_1/2) \\ k_3 &= h\, f(t_j + h/2, x_j + k_2/2) \\ k_4 &= h\, f(t_j + h, x_j + k_3) \end{cases} \tag{B.7}$$

で $\{x_j\}_{j=1}^N$ を計算する方法を **Runge-Kutta 法**という。

Runge-Kutta 法は、適度に簡単で、そこそこの効率を持つ方法のため、常微分方程式の初期値問題の「定番の数値解法」としての地位を得ている。

プロでないユーザーとしては、

まずは Runge-Kutta 法でやってみて、それでダメなら考える

という態度で取り組めばいい、と思う。どういう問題が Runge-Kutta 法で解くのにふさわしくないかは、後の章で後述する。

B.6 漸化式のプログラミング

ここでは直接に常微分方程式の数値解法のプログラミングに取り組む前に、漸化式

$$x_{j+1} = F(x_j) \quad (j = 0, 1, \cdots, N-1)$$

で定まる列 $\{x_j\}_{j=0}^N$ を計算するプログラムの書き方について述べよう。

B.6.1 F が実数値の場合

この場合は特に簡単である。例として $F(x) = x/4 + 1,\ N = 100$ の場合のプログラムを具体的にあげよう。

配列を用いるのは分かりやすい。

配列を用いたプログラム

```
#include <stdio.h>

#define N 100

int main()
{
    int j;
    double a[N+1], F(double);

    printf("a[0]: "); scanf("%lf", &a[0]);
    for (j = 0; j < N; j++) {
        a[j+1] = F(a[j]);
        printf("a[%d]=%g\n", j+1, a[j+1]);
    }
    return 0;
}

double F(double x)
{
    return 0.25 * x + 1.0;
}
```

注意　上のプログラムは、古い K&R スタイルで書いてある。関数のプロトタイプ宣言の使える ANSI C ならば

ANSI C の関数プロトタイプを用いたプログラム

```
#include <stdio.h>

#define N 100

int main()
{
    int j;
    double a[N+1], F(double);

    printf("a[0]: "); scanf("%lf", &a[0]);
    for (j = 0; j < N; j++) {
        a[j+1] = F(a[j]);
        printf("a[%d]=%g\n", j+1, a[j+1]);
    }
    return 0;
}

double F(double x)
{
    return 0.25 * x + 1.0;
}
```

とする。

ところが、残念なことに、 N が大きいときには、このプログラムの書き方はあまり良くない。配列 a[] を記憶するために大きなメモリーが必要になってしまう。そこで、次のようなプログラムを書くのが普通である。

配列を用いず、変数を書き換えて済ますプログラム

```
#include <stdio.h>

#define N 100

int main()
{
    int j;
    double a, F(double);

    printf("a[0]: "); scanf("%lf", &a);
    for (j = 0; j < N; j++) {
        a = F(a);
        printf("a[%d]=%g\n", j+1, a);
    }
    return 0;
}

double F(double x)
{
    return 0.25 * x + 1.0;
}
```

B.6.2 F がベクトル値の場合

F がベクトル値の場合も、実数値の場合と基本的には変わらないが、初心者がよく犯す間違いがあるので、特に説明する。(実数値とベクトル値でプログラムの書き方を変えねばならないのは、C 言語がベクトルを基本的なデータとして扱えないためであるとも言える。実際、C++ 言語でベクトルを扱うクラス・ライブラリィを適当に作れば、前小節のような感じでプログラミングできる。)

まず、間違いのあるプログラムから。

間違いのあるプログラム

```
#include <stdio.h>

#define N 100

int main()
{
    int j;
    double a, b, F(double, double), G(double, double);

    printf("a[0], b[0]: "); scanf("%lf %lf", &a, &b);
    for (j = 0; j < N; j++) {
        /* ここに間違いがある！真似をしてはダメ!! */
        a = F(a, b);
        b = G(a, b);
        printf("(a[%d],b[%d])=(%g,%g)\n", j+1, j+1, a, b);
    }
    return 0;
}

double F(double x, double y)
{
    return x / 2 + y / 3 + 1.0;
}

double G(double x, double y)
{
    return - x / 3 + y / 2 - 0.5;
}
```

正しくするには、例えば

正しいプログラム

```
#include <stdio.h>

#define N 100

int main()
{
    int j;
    double a, b, newa, newb, F(double, double), G(double, double);

    printf("a[0], b[0]: "); scanf("%lf %lf", &a, &b);
    for (j = 0; j < N; j++) {
        newa = F(a, b);
        newb = G(a, b);
        a = newa;
        b = newb;
        printf("(a[%d],b[%d])=(%g,%g)\n", j+1, j+1, a, b);
    }
    return 0;
}

double F(double x, double y)
{
    return x / 2 + y / 3 + 1.0;
}

double G(double x, double y)
{
    return - x / 3 + y / 2 - 0.5;
}
```

B.7 プログラミング課題

$$\begin{aligned} \frac{dx}{dt} &= x \\ x(0) &= 1 \end{aligned}$$

を $0 \leq t \leq 1$ の範囲で、Euler 法または Runge-Kutta 法で解くプログラムを以下の手順で作って実験する。

(1) $N = 10, 20$ などに対して計算してみて、x_N が $x(1) = e^1 = e = 2.7182818284\cdots$ に近いかどうかチェックせよ (大きくずれているような場合はプログラムが正しいかどうか見直すこと)。

(2) 計算結果をもとにして解曲線を描け。

(3) 誤差 $e_N \stackrel{\text{def.}}{=} |e - x_N|$ が N の増加とともにどう変化するか調べよ。

付 録C　常微分方程式の初期値問題の数値解法

ここから 3 章は、ある時期、数学科の『情報処理 II』で講義していた内容である。

C.1　はじめに

C.1.1　常微分方程式って何だったっけ — 復習

常微分方程式というのは大雑把に言うと、「一つの実独立変数 t の未知関数 $x = x(t)$ を求めるための問題で、x とその導関数 $\dfrac{dx}{dt}$, $\dfrac{d^2x}{dt^2}$, $\cdots$, $\dfrac{d^kx}{dt^k}$ についての方程式になっているもの」のことです。(以下では $\dfrac{dx}{dt} = x'$, $\dfrac{d^2x}{dt^2} = x''$, $\dfrac{d^3x}{dt^3} = x^{(3)}$, $\cdots$ のような書き方もします。)

例 **1** $x'(t) = f(t)$　(f は既知関数)

例 **2** $x''(t) = -g$　(g は既知の定数)

例 **3** $x'(t) = ax(t)$　(a は既知の定数)

例 **4** $x''(t) = -\omega^2 x(t)$　(ω は既知の定数)

例 **5** $x''(t) + 2\gamma x(t) + \omega^2 x(t) = 0$　(γ, ω は既知の定数)

ここに例としてあげた方程式はいずれも割とポピュラーなものなのですが、見覚えがあるでしょうか？どの場合もこれらの方程式だけでは解が一つに定まらず、何らかの条件を付け足すことによって初めて解が決定されます。その条件として、ある特定の t の値 t_0 に対する x の値 $x(t_0)$ や導関数の値を指定するというタイプのものがよくありますが、そういうものを初期条件と呼びます (これは t が時刻を表す変数で、t_0 を現象が始まる時刻のように解釈するからでしょう)。

例えば、上の例 $1, 3$ に対して

$$x(0) = x_0 \quad (x_0 \text{ は既知定数}),$$

例 $2, 4, 5$ に対して

$$x(0) = x_0, \quad x'(0) = v_0 \quad (x_0, v_0 \text{ は既知定数})$$

のように与えられた条件が初期条件です。また、初期条件を添えて解が決定されるようにした問題を、(常微分方程式の) 初期値問題と言います。

C.1.2　これからの目標

微分方程式は、他の諸科学への応用のみならず[1]、数学それ自体にとっても非常に重要です[2]。ところが困ったことに、微分方程式は大抵の場合に、良く知られている関数で解を表現することが出来ません。これは解が存在しないということではありません。解はほとんどいつでも存在するけれども、それを簡単な演算 (不定積分を取る、四則演算、逆関数を取る、初等関数に代入するなど) で求める — いわゆる求積法で解く — ことは、よほど特殊な問題でない限り出来ない、ということです。

この困った状況をある程度解決するのが、解を数値的に求める方法です。この情報処理 II では、いくつかの基本的な数値解法を学んで、実際に常微分方程式を解いてみます。これは計算機による数値シミュレーション[3]の典型例と呼べるものですし、マスターしておくと役に立ちます。

C.2　数値解法 (1)

C.2.1　問題の設定と数値解法の基本原理

常微分方程式としては正規形[4]のもののみを扱います。後で例で見るように、高階の方程式も一階の方程式に帰着されますから、当面一階の方程式のみを考えます。独立変数を t、未知関数を $x = x(t)$ とすれば、一階正規形の常微分方程式とは

$$\frac{dx}{dt} = f(t, x) \quad (t \in [a, b]) \tag{C.1}$$

の形に表わされる方程式のことです。ここで f は既知の関数です。初期条件としては

$$x(a) = x_0 \quad (x_0 \text{ は既知定数}) \tag{C.2}$$

の形のものを考えます。x_0 は既知の定数です。(1),(2) を同時に満たす関数 $x(t)$ を求めよ、というのが一階正規形常微分方程式の初期値問題です。この時関数 $x(t)$ を初期値問題 (1),(2) の解と呼びます。

常微分方程式の数値解法の基本的な考え方は次のようなものです。「問題となっている区間 $[a, b]$ を

$$a = t_0 < t_1 < t_2 < \cdots < t_N = b$$

と分割して、各 "時刻" t_j での x の値 $x_j = x(t_j)$ $(j = 1, 2, \ldots, N)$ を近似的に求めることを目標とする。そのために微分方程式 (1) から $\{x_j\}_{j=0,\ldots,N}$ を解とする適当な差分方程式[5]を作り、それを解く。」

---

[1] 微分方程式は物理学の問題を扱うために発明されましたが、現在では自然科学以外でも応用されています。

[2] 常微分方程式の簡単なものは高等学校でも学びましたし、1 年次にも微分方程式という授業がありました。数学科の 3 年次にもより詳しいことを学ぶための講義があります。常微分方程式に対する参考書は色々ありますが、例えば、 3 年生向けの講義の教科書になっている、笠原晧司著「微分方程式の基礎」朝倉書店、をあげておきます。

[3] simulation(模擬実験) を「シュミレーション」と読み間違えないでください。「シミュレーション」ですからね。

[4] 方程式が最高階の導関数について解かれている、ということですが、よく分からなくても差し支えありません。

[5] 漸化式のようなものだと思って構いません。差分とは、高等学校の数列で言う階差のことです。

区間 $[a,b]$ の分割の仕方ですが、以下では簡単のため N 等分することにします。つまり

$$h = (b-a)/N, \quad t_j = a + jh.$$

となります。

C.2.2 Euler(オイラー) 法の紹介

微分 $x'(t) = \dfrac{dx}{dt}$ は差分商 $\dfrac{x(t+h)-x(t)}{h}$ の $h \to 0$ の極限です。そこで、(1) 式の微分を差分商で置き換えて近似することによって、次の方程式を得ます。

$$\frac{x_{j+1}-x_j}{h} = f(t_j, x_j) \quad (j = 0, 1, \ldots, N-1)$$

変形すると

$$x_{j+1} = x_j + hf(t_j, x_j). \tag{C.3}$$

これを漸化式として使って、x_0 から順に x_1, x_2, …, x_N が計算出来ます。この方法を ruby オイラー **Euler** 法と呼びます。

こうして得られる $N+1$ 個の点 (t_j, x_j) を順に結んで得られる折れ線関数は (f に関する適当な仮定のもとで) $N \to +\infty$ の時 ($h = (b-a)/N$ について言えば $h \to 0$)、真の解 $x(t)$ に収束することが証明できます[6]。ここでは簡単な例で収束を確かめてみましょう。

C.2.3 プログラミングの仕方

初期値 x_0 が与えられたとき、漸化式 (C.3) によって、数列 $\{x_j\}_{j=1,\cdots,N}$ を計算するプログラムはどう作ったらよいでしょうか？ここでは二つの素朴なやり方を紹介しましょう。

配列を使う方法 数列を配列で表現するのは、Fortran では自然な発想です。例えば

```
integer MAXN
parameter (MAXN = 1000)
real x(0:MAXN)
```

のように配列 “x” を用意しておいて

```
x(0) = x0
t = a
do j=0,N-1
  x(j+1) = x(j) + h * f(t,x(j))
  t = t + h
end do
```

とするわけです。

[6]現在の数学科のカリキュラムでは 3 年次に開講されている常微分方程式の講義で学びます。

配列を使わないですませる方法 漸化式 (C.3) を解くために、配列は絶対必要というわけではありません[7]。例えば、変数 “x” に各段階の x_j の値を収めておくとして

```
x = x0
t = a
do j = 0,N-1
  x = x + h * f(t,x)
  t = t + h
end do
```

のようなプログラムで計算が出来ます。

重箱の隅をつつく注意: “t = t + h” とすると、誤差が蓄積されがちです。これを避けるには、次のように書き換えると良いでしょう。

```
t = a + (j + 1) * h
```

C.2.4 例題

例 1 初期値問題

$$x'(t) = x(t) \quad (t \in [0,1]), \quad x(0) = 1$$

の解は $x(t) = e^t$ であるが、Euler 法を用いて解くと $x_N = \left(1 + \frac{1}{N}\right)^N$ となる。したがって、確かに $N \to +\infty$ の時に $x_N \to e = x(1)$ となっている。

例題 5.1 Euler 法を用いて、例 1 の初期値問題を解くプログラムを作って収束を調べよ。

```
* reidai5-1.f -- 微分方程式の初期値問題を Euler 法で解く
      program rei51
*       開始時刻と終了時刻
      real a,b
      parameter (a=0.0,b=1.0)
*       変数と関数の宣言
      integer N,j
      real t,x,h,x0,f
      external f
*       初期値
      x0 = 1.0
*       区間の分割数 N を入力してもらう
      write(*,*) ' N='
      read(*,*) N
*       小区間の幅
      h=(b-a)/N
*       開始時刻と初期値のセット
      t=a
      x=x0
```

[7] 配列はメモリーを消費しますし、(特に Fortran の場合、配列の大きさは実行時に変更できないので) プログラムを書く際に、どれくらいの大きさの配列を用意したらいいのかという問題に悩まなくてはなりません。

```
      write(*,*) t,x
*      Euler 法による計算
      do j=0,N-1
         x=x+h*f(t,x)
         t=t+h
         write(*,*) t,x
      end do
      end
************************************************************************
*       微分方程式 x'=f(t,x) の右辺の関数 f の定義
      real function f(t,x)
      real t,x
      f=x
      end
```

このプログラムをコンパイルして[8]実行すると、分割数 N を尋ねてきますので、色々な値を入力して試してみて下さい。各時刻 t_j における x_j の値 $(j = 0, 1, \cdots, N)$ を画面に出力します。

確認用にいくつかの N の値に対する場合の、$x(1) = x_N$ の値を書いておきます。$N = 10$ の場合 $x_N = 2.59374261$, $N = 100$ の場合 $x_N = 2.70481372$, $N = 1000$ の場合 $x_N = 2.71692038$, $N = 10000$ の場合 $x_N = 2.71814346, \cdots$.

分割数 N が大きくなるほど、真の値 $e = 2.7182818284590452\cdots$ に近付いていくはずです。

問題 **5.1**　例題 5-1 とは異なる初期値問題を適当に設定して、それを Euler 法で解いてみよ。(結果についてもきちんと吟味すること。)

問題 **5.2**　“reidai5-1.f” の出力するデータを用いて、解曲線 (関数 $t \mapsto x(t)$ のグラフのこと) を描きなさい。

C.2.5　Euler 法の収束の速さ

先ほどの実験では Euler 法による解は N が大きくなればなるほど真の解に近くなるはずです。実際 $N \to +\infty$ とすると真の解に収束することが証明できるわけなのですが、それでは、どれくらいの速さで収束するのでしょうか？これを実験的に調べてみましょう。

例題 **5.2**　例題 5.1 と同じ初期値問題で、色々な分割数 N に対して問題を時、$t = 1$ での誤差の大きさ $|x(1) - x_N| = |e - x_N|$ について調べよ。

こういう場合は、色々な N に対して一斉に $|e - x_N|$ を計算するプログラムを作るのがいいでしょう。

```
* reidai5-2.f --- 常微分方程式の初期値問題を Euler 法で解く
      program rei52
*       開始時刻と終了時刻
```

[8] コンパイルすると、
“jwd2008i-i "reidai5-1.f", line 30: この仮引数は，副プログラム中で使用されていません.(名前:t)” という警告メッセージが出ますが、大丈夫です (言っていることは確かにもっともですが)。

```
      real a,b
      parameter (a=0.0,b=1.0)
*       初期値
      real x0
*       変数と関数の宣言
      real t,x,h,f,e
      integer N0,N1,N2,N,i
*       自然対数の底 e (=2.7182818284..)
      e=exp(1.0)
*       初期値の設定
      x0=1.0
*       どういう N について計算するか？
*       N0 から N1 まで、N2 刻みで増える N に
      write(*,*) ' FIRSTN,LASTN,STEPN='
      read(*,*) N0,N1,N2
*       計算の開始
      do N=N0,N1,N2
         h=(b-a)/N
*       開始時刻と初期値のセット
         t=a
         x=x0
*       Euler 法による計算
         do i=0,N-1
            x=x+h*f(t,x)
            t=t+h
         end do
         write(*,*) N,abs(e-x)
      end do
      end
**********************************************************************
*       微分方程式 x'=f(t,x) の右辺の関数 f の定義 -- 前と同じだから略
```

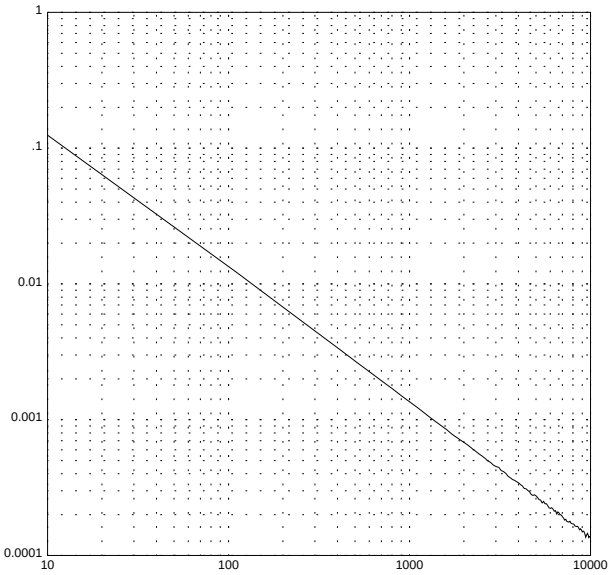
コンパイルして実行すると “ FIRSTN,LASTN,STEPN=” と尋ねてきます。例えば “100 1000 50” と答えると 100 から 1000 まで 50 刻みで増やして行った値 $100, 150, 200, \cdots, 900, 950, 1000$ を N として計算して、最終的に得られた誤差を出力します。実行結果を見てみましょう。

```
waltz11% reidai5-2
 FIRSTN,LASTN,STEPN=
100 1000 50
  100    1.34680271e-02
  150    9.00602341e-03
  200    6.76608086e-03
  .... (途中略) ...
  900    1.50966644e-03
  950    1.42812729e-03
  1000    1.36137009e-03
waltz11%
```

“mgraph” コマンドには、対数目盛によるグラフを描く機能 (起動時に -lx, -ly を指定します。ここで “l” は “logarithmic” (=「対数の」) の頭文字です) があります。この場合は、例えば次のようにしたりすると良いでしょう。

```
waltz11% reidai5-2 | mgraph -lx -ly | xplot
```

```
10,10000,100
```



このグラフから、誤差 $= O(N^{-1})\quad (N \to +\infty)$ であることが読みとれます。実はこれは Euler 法の持つ一般的な性質です。

実は Euler 法は収束があまり速くないので、実際には特殊な場合を除いて使われていません。そこで、、、、

C.3 Runge–Kutta (ルンゲ–クッタ) 法

前節で解説した Euler 法は簡単で、これですべてが片付けば喜ばしいのですが、残念ながらあまり効率が良くありません。高精度の解を計算するためには、分割数 N をかなり大きく取る (=大量の計算をする) 必要があります。特別な場合[9]を除けば、実際に使われることは滅多にないでしょう。率直にいって実用性は低いです。

より高精度の公式は、現在まで様々なものが開発されていますが、比較的簡単で、精度がまあまあ高いものに Runge–Kutta 法と呼ばれるものがあります。それは x_j から x_{j+1} を求める漸化式として次のものを用います。

$$\begin{aligned} k_1 &= hf(t_j, x_j) \\ k_2 &= hf(t_j + h/2, x_j + k_1/2) \\ k_3 &= hf(t_j + h/2, x_j + k_2/2) \\ k_4 &= hf(t_j + h, x_j + k_3) \\ x_{j+1} &= x_j + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

これがどうやって導かれたものかは解説しません。まずは使ってみましょう。

[9]例えば微分方程式の右辺に現れる関数 f が、解析的な式で定義された滑らかなものではなく、実験により計測されたデータにより定義されている場合等は、高精度の公式を用いるよりも、Euler 法の方が良いことがあります。

```
* reidai5-3.f -- 微分方程式の初期値問題を Runge-Kutta 法で解く
      program rei53
*       開始時刻と終了時刻
      real a,b
      parameter (a=0.0,b=1.0)
*       変数と関数の宣言
      integer N,j
      real t,x,h,x0,f,k1,k2,k3,k4
      external f
*       初期値
      x0 = 1.0
*      区間の分割数 N を入力してもらう
      write(*,*) ' N='
      read(*,*) N
*      小区間の幅
      h=(b-a)/N
*      開始時刻と初期値のセット
      t=a
      x=x0
      write(*,*) t,x
*      Runge-Kutta 法による計算
      do j=0,N-1
         k1 = h * f(t, x)
         k2 = h * f(t + h / 2, x + k1 / 2)
         k3 = h * f(t + h / 2, x + k2 / 2)
         k4 = h * f(t + h, x + k3)
         x = x + (k1 + 2 * k2 + 2 * k3 + k4) / 6
         t = t + h
         write(*,*) t,x
      end do
      end
**********************************************************************
*       微分方程式 x'=f(t,x) の右辺の関数 f の定義 --- これも前と同じだから略
```

コンパイル・実行の結果は次のようになるはずです。

```
waltz11% f77o reidai5-3.f
Fortran 90 diagnostic messages: program name(f)
  jwd2008i-i  ``reidai5-3.f'', line 30: この仮引数は, 副プログラム中で使用されていま
せん.(名前:t)
waltz11% reidai5-3
  N=
10
 0.000000000e+00 1.00000000
 0.100000001 1.10517085
 0.200000003 1.22140265
 0.300000012 1.34985852
 0.400000006 1.49182427
 0.500000000 1.64872062
 0.600000024 1.82211792
 0.700000048 2.01375151
```

```
 0.800000072 2.22553945
 0.900000095 2.45960140
 1.00000012 2.71827984
waltz11%
```

たった 10 等分なのに相対誤差が 10^{-6} 以下になっています ($\because x(1) = e^1 = e = 2.7182818284\cdots$ であるので、相対誤差$=|e-2.71827984|/e = 7.31\cdots\times10^{-7}$)。Runge–Kutta 法の公式は Euler 法よりは大部面倒ですが、それに見合うだけの価値があることが納得できるでしょう。

問題 **5-3**　“reidai5-3.f” を普通の “real”(単精度実数型) の代わりに “real*8” (倍精度実数型) を使うように書き換えて、大きな N に対してどうなるか、実験しなさい。倍精度化のための書き換えは、具体的には、(i) プログラムの中に何箇所かある “real” を “real*8” に書き換える、(ii) 実数の定数を “0.0” → “0.0d0” のように、末尾に “d0” を付けたりして[10]、倍精度実数定数に書き換える、 (iii) “real” や “complex” への型変換を指定しているとことがあれば、それぞれ “real*8”, “complex*16” への型変換に書き換える、というようなことをやればいいが、今の場合は (i) だけでも十分動作するプログラムになる。

問題 **5-4**　区間の分割数 N を変えながら

$$x'(t) = x \quad (t \in [0,1]), \quad x(0) = 1$$

を Runge–Kutta 法を用いて解くプログラムを作り、$t = 1$ での誤差 $|x_N - x(1)|$ を調べよ (前節 “reidai5-2.f” の真似をする)。Euler 法と比べてどうなるか？

[10] “d 整数” は “e 整数” のような指数形式の実数表現の指数部を表します。よって “1.0e-7” のような定数は “1.0d-7” のように書き換えます。

付 録D　定数係数線形常微分方程式

D.1　問題の説明 — 定数係数線形常微分方程式

前回は常微分方程式の初期値問題に対する数値解法 (Euler 法、Runge-Kutta 法) の紹介をしましたが、今回はそれを連立常微分方程式の初期値問題

$$\begin{cases} \dfrac{dx}{dt} &= ax+by \\ \dfrac{dy}{dt} &= cx+dy \end{cases} \quad (t \in \mathbf{R}), \qquad \begin{cases} x(0) &= x_0 \\ y(0) &= y_0 \end{cases}$$

を解くのに使ってみましょう。ここで $x = x(t)$, $y = y(t)$ は未知関数、a, b, c, d, x_0, y_0 は既知定数です。

この問題は後で注意するように、色々な応用があって重要ですが、それだけではなく、数学的にも基本的で面白く、是非一度は数値実験を体験しておきたいものです。

注意 1　この問題は、計算機を使わなくても線形代数を用いて解くことが出来ます。既にどこかで習っているかもしれませんし、そうでない場合も三年次の常微分方程式の講義で学ぶことになるでしょう。(このプリントの末尾に計算の仕方だけ説明しておきます。)

後で数学的な説明をするときのために、問題をベクトル、行列を用いて書き換えましょう。

$$\vec{x}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}, \quad A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \vec{x}_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

とおくと[1]、$\vec{x} = \vec{x}(t)$ は未知の 2 次元ベクトル値関数、 A は 2 次の実正方行列、$\vec{x}_0$ は $\mathbf{R}^2$ の要素となり、問題は

$$\frac{d\vec{x}}{dt} = A\vec{x} \tag{1}$$

$$\vec{x}(0) = \vec{x}_0 \tag{2}$$

と書き直されます。このような問題を定数係数線形常微分方程式の初期値問題とよびます。

初期値問題 (1), (2) の解は平面内での点の運動を表わしていると考えることが出来ます。初期値 $\vec{x}_0$ を色々と変えて、それに対応する解 $\vec{x}(t)$ の軌跡 (解軌道と呼びます) を描いてみましょう。この解軌道を考える時の空間 (ここでは平面 $\mathbf{R}^2$) を相空間 (phase space)[2] と呼びます。

[1] 今回は x がベクトルであることを強調するために矢印 $\vec{}$ をつけて $\vec{x}$ と書きます。

[2] “phase space” は数学以外の本では「位相空間」と訳されることが多いですが、数学では「相空間」という訳語を用います。これは「位相空間」という言葉は数学では “topological space” の訳語に使われるからです。

$f(t,\vec{x}) = Ax$ とおくと、方程式 (1) は

$$\frac{d\vec{x}}{dt} = f(t,\vec{x})$$

となって、前回の方程式と同じ形になります。前回紹介した Euler 法、Runge-Kutta 法などの数値解法は (実数だったところが、ベクトルになるだけで) まったく同様に適用することが出来ます。

注意 **2**　高校までの数学で、最も簡単で基本的な関数は正比例の関数 $x \mapsto ax$ (a は定数) でしょう。ここでの線形写像 $\vec{x} \mapsto A\vec{x}$ (A は正方行列) は、正比例の関数の一般化と考えられ、最も基本的な写像と言えるでしょう。

注意 **3**　(物理からの例) いわゆる単振動の方程式

$$\frac{d^2x}{dt^2} = -\omega^2 x \quad (\omega \text{ は正定数})$$

は $y = dx/dt$, $\vec{x} = {}^t(x,y)$ と置くことにより (1) の形に帰着されます。ここで

$$A = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix}.$$

同じ置き換えで、速度に比例する抵抗力がある場合の方程式

$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \omega^2 x = 0 \quad (\omega, \gamma \text{ は正定数})$$

も (1) の形に帰着されます。この場合は

$$A = \begin{pmatrix} 0 & 1 \\ -\omega^2 & -\gamma \end{pmatrix}.$$

D.2　例題プログラムによる実験

今回の例題は一つだけで、これで遊んでもらうだけでよしとします。

D.2.1　例題プログラムの使い方

例題 **7-1**　問題 (1),(2) で適当な係数行列を選び、初期値 $\vec{x}_0$ を色々と変えて、それに対応する解軌道を描け。

いつものように `getsample` コマンドでサンプルプログラムをコピーした後に、`f77x` でコンパイルして、実行して下さい。

```
waltz11% getsample
waltz11% f77x reidai7-1.f
waltz11% reidai7-1
```

最初に行列 A の成分 a, b, c, d を尋ねてきますので、自分が調べたいと思う行列を選んで入力します。

```
a,b,c,d=
0 1 -1 -1
```

するとウィンドウが開かれた後に、次のようなメニューが表示されます。

```
したいことを番号で選んで下さい。
 -1:メニュー終了，0:初期値のキーボード入力，1:初期値のマウス入力，
  2:刻み幅, 追跡時間変更 (現在 h= 0.0100,T=10.0000)
```

この意味は希望することを選ぶのに、-1 から 2 までの整数を入力しなさい、ということです。'0' を入力すると、キーボードから数値で初期条件 $x0, y0$ を入力することになります。

```
0                                          ← 0 番を選択する。
初期値 x0,y0=                              → x0,y0 の入力の催促。
0.5 0.5                                    ← 0.5 0.5 を入力。
```

また '1' を入力した場合は、マウスで初期値を指定することが出来ます。fplot のウィンドウの中の初期値としたい点のところまでマウス・カーソルを移動して、マウスの左ボタンをクリックします。

```
マウスの左ボタンで初期値を指定して下さい ( 右ボタンで中止 )。
(x0,y0)=-0.724609  -0.365234          → マウスで指定した点の座標
マウスを使って初期値を入力して下さい。    → 次の入力を催促
```

これに対してマウスの真中のボタンを押すと、t が負の方向に解きます。マウスを一箇所に固定したまま、左のボタンと真中のボタンを押して効果を確かめて下さい。

マウスを使っての初期値の入力を止めるには、マウスの右ボタンを押します。するとメニューまで戻るはずです。

メニューを抜けるには、メニューで '-1' を入力します。その後マウスを fplot ウィンドウに持っていき、ボタンをクリックすると reidai7-1 を終了することが出来ます。

ここでは初期値のサンプル・データ reidai7-1.data も用意してあります (内容は注意 3 の二つ目の方程式で $\omega = \gamma = 1$ の場合の実験です)。これを試すには以下のようにして下さい。

```
waltz11% cat reidai7-1.data | reidai7-1
```

注意 4　このサンプルの例では、解軌道は後で述べるように、内向きの対数螺旋 (らせん) になります。時刻 t が大きくなると点 $\vec{x}(t)$ は急速に原点に近付くのですが、到達はしません。画面では見分けがつきませんので、誤解しないように注意して下さい。

D.2.2　解説

さて、今日はこの reidai7-1 で色々 (行列を替えて) 実験してもらうのが目的なのですが、まったく闇雲にやっても、なかなかうまく行かない (重要な現象に遭遇できない) でしょうから、以下少し数学的背景を説明します。

行列 A を変えると、解軌道の作るパターンが変わるのですが、それらは以下のように比較的小数のケースに分類されます。どのケースに属するか調べるには、行列 A の固有値に注目します。A の固有値とは A の固有方程式

$$\det(\lambda I - A) = 0 \quad \text{すなわち} \quad \lambda^2 - (a+d)\lambda + (ad - bc) = 0$$

の根 λ_1, λ_2 のことでした。

Case I.　A の固有値が相異なる 2 実数である場合

固有値がいずれも 0 でない場合は、原点が唯一の平衡点になっていますが、詳しく分類すると

(i) $\lambda_1, \lambda_2 > 0$ (ともに正) ならば湧出点 (不安定結節点)

(ii) $\lambda_1, \lambda_2 < 0$ (ともに負) ならば沈点 (安定結節点)

(iii) $\lambda_1 \lambda_2 < 0$ (異符号) ならば鞍状点

となります (湧出点、沈点、鞍状点の定義はここには書きません。自分で試してみて納得してください)。

(iv) λ_1, λ_2 のいずれか一方が 0 ならば、ある原点を通る一つの直線上の点が平衡点の全体となります。

Case II.　A の固有値が 2 重根で、A が対角化可能である場合

これは結局 $A = \lambda I$ と書けるということで (λ は固有値)、単純なケースです。$\lambda \neq 0$ である限り、原点は唯一の平衡点となり、$\lambda > 0$ ならば湧出点、$\lambda < 0$ ならば沈点です。$\lambda = 0$ ならば平面上のすべての点が平衡点です (つまり $A = 0$ で、全然動かない)。

Case III.　A の固有値が 2 重根で、A が対角化不能である場合

例えば $A = \begin{pmatrix} \lambda & 1 \\ 0 & \lambda \end{pmatrix}$ のような場合です。$\lambda \neq 0$ であれば原点が唯一の平衡点で、$\lambda > 0$ であれば湧出点、$\lambda < 0$ であれば沈点です。$\lambda = 0$ であれば、原点を通るある直線上の点すべてが平衡点となります。

Case IV. A の固有値が二つの相異なる虚数である場合

この場合、固有値は $\mu \pm i\nu(\mu, \nu$ は実数, $\nu \neq 0)$ と書けます。平衡点は原点だけです。

1. $\mu > 0$ であれば、解軌道は外向きの対数螺旋になります。こういう場合「原点は不安定渦状点である」と言います。

2. $\mu < 0$ であれば、解軌道は内向きの対数螺旋になります。こういう場合「原点は安定渦状点である」と言います。

3. $\mu = 0$ であれば、解軌道は楕円になります (特別な場合として円を含みます)。こういう場合「原点は渦心点 (または中心点) である」と言います。

問題 **7-1**　様々な場合にについて、自分で適当な行列 A を探して解軌道を描いてみなさい。(自分で探すのが面倒という人は、以下の行列を試してみて下さい。どの Case に相当しますか？)

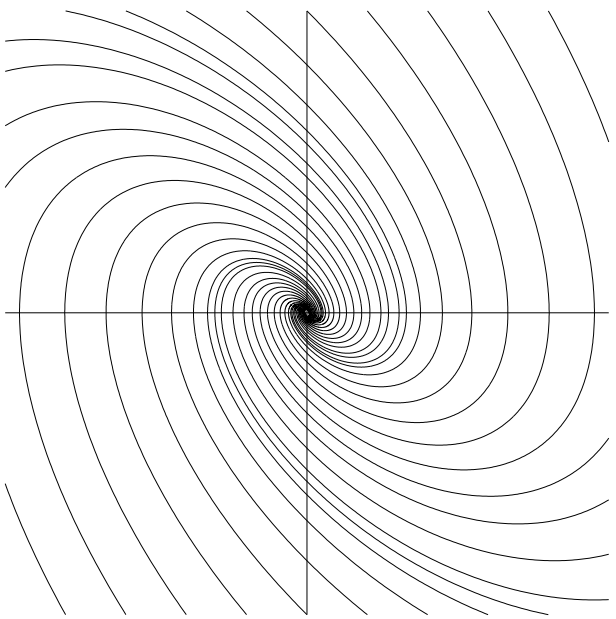
$$\begin{pmatrix} -\frac{4}{5} & -\frac{3}{5} \\ \frac{2}{5} & -\frac{11}{5} \end{pmatrix}, \quad \begin{pmatrix} \frac{8}{5} & -\frac{9}{5} \\ \frac{6}{5} & -\frac{13}{5} \end{pmatrix}, \quad \begin{pmatrix} \frac{2}{5} & \frac{9}{5} \\ -\frac{1}{5} & \frac{8}{5} \end{pmatrix}, \quad \begin{pmatrix} -1 & 2 \\ -1 & 1 \end{pmatrix}.$$

(Fortran プログラムの read 文では、分数を読み込めません。必ず小数に変換してから入力して下さい。たとえば $\frac{4}{5}$ は 0.8 として入力します。)

問題 **7-2**　`reidai7-1` で Runge-Kutta 法を用いているところを Euler 法に書き変えなさい。いくつかの行列 (特に Case IV-3 に属するもの) に対する問題 (1),(2) を 2 つのプログラムで解き比べて見なさい。

問題 **7-3**　注意 3 であげた 2 つの微分方程式は上の分類でどこに属するか？また解軌道を見て、その解のどんな性質が分かるか？

問題 **7-4**　`reidai7-1` で、初期値をキーボードから数値で入力する方法とマウスで入力する方法の長所、短所を論じなさい。



D.3　補足 — 紙と鉛筆で解く方法

ここに書いてあることは、線形代数や常微分方程式を学んでいる際に学ぶ機会があると思いますが、一応まとめておきます。

D.3.1　定数係数線形常微分方程式の解の公式, 行列の指数関数

定数係数線形常微分方程式の初期値問題 (1),(2) の解は一意で $\vec{x}(t) = e^{tA}\vec{x}_0$ で与えられる。ここで e^{tA} は行列の指数関数というもので、次式で定義される:

$$e^B = \exp B \equiv \sum_{n=0}^{\infty} \frac{B^n}{n!}.$$

いくつか具体例をあげると、$B = \begin{pmatrix} \alpha & 0 \\ 0 & \beta \end{pmatrix}$ の場合 $e^B = \begin{pmatrix} e^\alpha & 0 \\ 0 & e^\beta \end{pmatrix}$, $B = \begin{pmatrix} 0 & -\beta \\ \beta & 0 \end{pmatrix}$ の場合 $e^B = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix}$, $B = \begin{pmatrix} \alpha & -\beta \\ \beta & \alpha \end{pmatrix}$ の場合 $e^B = e^\alpha \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix}$ となる (定義にしたがって計算してみれば、5 分もあれば確かめられるであろう)。

後のために $\exp(P^{-1}BP) = P^{-1}e^B P$ となることを注意しておく。

D.3.2　$N = 2$ の場合の e^{tA}, $e^{tA}\vec{x}_0$

今回の問題を理解するため、行列の指数関数を $N = 2$ の場合に詳しく解析してみる。$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ として、A の固有方程式 $\lambda^2 - (a+d)\lambda + ad - bc = 0$ の根を判別して場合分けする。

(I) 相異なる 2 実根 λ_1, λ_2 を持つ場合

u_i を λ_i に属する A の固有ベクトルとする $(i = 1, 2)$ とすると、u_1, u_2 は線形独立になるので、任意の $x_0 \in \mathbf{R}^2$ は

$$x_0 = c_1 u_1 + c_2 u_2$$

と u_1, u_2 の線形結合で表すことが出来る。これから

$$A^n x_0 = A^n(c_1u_1 + c_2u_2) = c_1A^nu_1 + c_2A^nu_2 = c_1{\lambda_1}^n u_1 + c_2{\lambda_2}^n u_2 = {\lambda_1}^n(c_1u_1) + {\lambda_2}^n(c_2u_2),$$

$$e^{tA}x_0 = e^{\lambda_1 t}(c_1u_1) + e^{\lambda_2 t}(c_2u_2).$$

(つまり各 u_i 成分 c_iu_i に関しては $e^{\lambda_i t}$ をかけるという単純な作用になる。)

行列の言葉で書くと、 $P = (u_1\ u_2)$ と置くと、

$$P^{-1}AP = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \quad P^{-1}A^nP = \begin{pmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{pmatrix}, \quad P^{-1}e^{tA}P = \begin{pmatrix} e^{\lambda_1 t} & 0 \\ 0 & e^{\lambda_2 t} \end{pmatrix}.$$

これから

$$e^{tA} = P \begin{pmatrix} e^{\lambda_1 t} & 0 \\ 0 & e^{\lambda_2 t} \end{pmatrix} P^{-1}.$$

(II) 重根 λ_0 を持つ場合

この場合は、一次独立な固有ベクトルが 2 つ取れるか、1 つしか取れないかで、二つの場合に別れる。

(II-i) 重根 λ_0 に属する二つの一次独立な固有ベクトル u_1, u_2 が存在する場合

上と同様にして $P^{-1}AP = \begin{pmatrix} \lambda_0 & 0 \\ 0 & \lambda_0 \end{pmatrix}$, これは実は $A = \begin{pmatrix} \lambda_0 & 0 \\ 0 & \lambda_0 \end{pmatrix}$ ということだから、

$$A^n = \begin{pmatrix} \lambda_0^n & 0 \\ 0 & \lambda_0^n \end{pmatrix}, \quad e^{tA} = \begin{pmatrix} e^{\lambda_0 t} & 0 \\ 0 & e^{\lambda_0 t} \end{pmatrix}, \quad e^{tA}x_0 = e^{\lambda_0 t}x_0.$$

(II-ii) 重根 λ_0 に属する一次独立な固有ベクトルが一つしか取れない場合

仮定より $\mathbf{R}^2 \neq \ker(\lambda_0 I - A)$ であり、$u_2 \in \mathbf{R}^2 \setminus \ker(\lambda_0 I - A)$ が存在する。そこで $u_1 = (A - \lambda_0 I)u_2$ とおくと $u_1 \neq 0$.

一方で $(A-\lambda_0 I)^2 = O$ である (実際 λ_0 は固有方程式の重根だから、固有多項式 $= (\lambda-\lambda_0)^2$. ゆえに Hamilton-Cayley の定理から $(A-\lambda_0 I)^2 = O$.)。よって $(A-\lambda_0 I)u_1 = (A-\lambda_0 I)^2 u_2 = 0$ すなわち $Au_1 = \lambda_0 u_1$.

これと $Au_2 = u_1 + \lambda_0 u_2$ から $P = (u_1\ u_2)$ とおくと、 $AP = A(u_1\ u_2) = (Au_1\ Au_2) = (\lambda_0 u_1\ u_1 + \lambda_0 u_2) = (u_1\ u_2)\begin{pmatrix} \lambda_0 & 1 \\ 0 & \lambda_0 \end{pmatrix} = P\begin{pmatrix} \lambda_0 & 1 \\ 0 & \lambda_0 \end{pmatrix}$. u_1, u_2 は一次独立だから P^{-1} が存在して、$P^{-1}AP = \begin{pmatrix} \lambda_0 & 1 \\ 0 & \lambda_0 \end{pmatrix}$. これから

$$P^{-1}A^nP = \begin{pmatrix} \lambda_0^n & n{\lambda_0}^{n-1} \\ 0 & \lambda_0^n \end{pmatrix}, \quad P^{-1}e^{tA}P = \begin{pmatrix} e^{\lambda_0 t} & te^{\lambda_0 t} \\ 0 & e^{\lambda_0 t} \end{pmatrix}, \quad e^{tA} = P\begin{pmatrix} e^{\lambda_0 t} & te^{\lambda_0 t} \\ 0 & e^{\lambda_0 t} \end{pmatrix}P^{-1}.$$

(III) 相異なる 2 虚根 $\lambda = \alpha \pm i\beta$ $(\alpha, \beta \in \mathbf{R},\ i = \sqrt{-1})$ を持つ場合

$\alpha+i\beta$ に属する固有ベクトルの一つを $x+iy$ $(x, y \in \mathbf{R}^2)$ とする。$A(x+iy) = (\alpha+i\beta)(x+iy) = (\alpha x-\beta y)+i(\beta x+\alpha y)$ の実部、虚部を取ると、$Ax = \alpha x-\beta y$, $Ay = \beta x+\alpha y$, それで $P = (x\ y)$ とおくと、$P^{-1}AP = \begin{pmatrix} \alpha & -\beta \\ \beta & \alpha \end{pmatrix}$. これから

$$P^{-1}e^{tA}P = e^{\alpha t}\begin{pmatrix} \cos\beta t & -\sin\beta t \\ \sin\beta t & \cos\beta t \end{pmatrix}.$$

これから $\alpha = 0$ ならば、$e^{tA}\vec{x}_0$ は t に関する周期関数であることが分かる (解軌道は楕円になる)。$\alpha > 0$ ならば $e^{tA}\vec{x}_0$ は段々原点から遠ざかり、$\alpha < 0$ ならば $e^{tA}\vec{x}_0 \to \vec{0}$ $(t \to \infty)$ であることも分かる。

付 録E　力学系とリミット・サイクル

前章のプログラムを、ほんの少し修正するだけで色々な問題が解けます。いざと言う時はこの程度の数値計算を実行できるようにしておくと、扱える問題の幅が広がります。

E.1　力学系と Poincaré のリミット・サイクル

E.1.1　力学系

これまで常微分方程式一般を表すために $\dfrac{dx}{dt} = f(t,x)$ と書いて来ましたが、右辺に現われる f が t に依らない場合、つまり

$$\frac{dx}{dt} = f(x) \tag{E.1}$$

という形の方程式を力学系(**dynamical**[1]**system**) あるいは自励系 (autonomous system) と呼びます。実はこの「情報処理 II」で取り扱った常微分方程式はすべてこの形のものでした。

力学系は以下のようなイメージでとらえることが出来ます。

空間内に時間によらない「流れ」があり、
点 x での流れの速度[2]は $f(x)$ となっている。

力学系の初期値問題とは、ある時刻での質点の位置を指定して、後はこの流れにまかせて移動した場合の、質点の運動を決定するものである、と言うことが出来ます。

E.1.2　平衡点と線形化

f が行列とベクトルの積の形になっている $f(x) = Ax$ の場合は、少し理論的な話をしました (前回)。一般の力学系も、この講義で解説した方法によって計算機を用いて解くことは可能ですが、理論的なアプローチとしてはどういうものが可能でしょうか？

一つの重要な方法は、平衡点をすべて見つけて、その回りの流れを「線形化の手法」で解析する、というものです。

(ここで平衡点とは、方程式の右辺が 0 となるような点、すなわち $f(a) = 0$ を見たす点 a のことです。直感的には、そこでは流れが止まっている点のことです。a が平衡点である場合、$x(t) \equiv a$ (値が恒等的に a となる関数) は方程式 (1) の解になります。)

線形化という言葉を説明する前に、実例を見て下さい。

---

[1] "mechanics" の「力学」ではありません。"dynamical" は「動的」という意味で、"statical" の反対語です。

[2] $\dfrac{dx}{dt}$ は速度を意味することは分かりますね？

例題 8-1　次の力学系の流れの様子を $-4 \leq x, y \leq 4$ で描きなさい:

$$\frac{d}{dt}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ -6x - y - 3x^2 \end{pmatrix}. \tag{2}$$

まず最初に平衡点を求めましょう。方程式の右辺のベクトル値関数 f が 0 になるという条件、つまり連立方程式

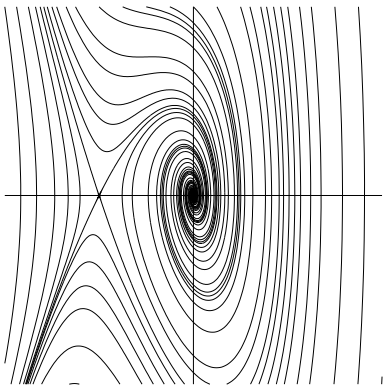
$$y = 0, \quad -6x - y - 3x^2 = 0$$

を解くと、$(x, y) = (0, 0), (-2, 0)$ となりますから、$\binom{0}{0}, \binom{-2}{0}$ という 2 点が平衡点です (それ以外に平衡点はありません)。「$-4 \leq x, y \leq 4$ で描きなさい」としたのは、その二つの平衡点のまわりの様子が分かるような範囲で描きなさい、という意味です。

さて、これを実行するには前回のプログラムをちょっと修正すれば OK です。そうして作ったプログラム `reidai8a.f` を用意してあります。いつものように `getsample` コマンドで手元にコピーした後に、コンパイルして実行してみましょう。ここではサンプルの入力データを収めたファイル `rei8a.data` もありますので、それを使って試すことにすれば、

```
getsample                      ← サンプルをコピー
f77x reidai8a.f                ← コンパイル
cat rei8a.data | reidai8a      ← サンプル・データで実行
```

で OK です。



さて、これを見て何に気がつくでしょうか？全体としては、これまで見たことがない図ですが、平衡点の近くでは、「どこかで見た」形をしていますね？

$\binom{0}{0}$ の回りでは安定渦状点、$\binom{-2}{0}$ の回りでは不安定結節点のような流れになっています。

大事なことは二つあって、一つは

平衡点は力学系の「ツボ」であって、
それを調べると多くの情報が分かる。

というものです。上の図で平衡点から離れたところはかなり単純な流れになっていることに注意して下さい。試しに描く範囲を大きくとって、自分でマウスを使って初期値を与えた図を描いてみるのもいいですね。次の例では $-100 \leq x, y \leq 100$ の範囲で描くように指定しています (基本的な使い方はこれまでと同じなので、もう説明は不要ですね？)。

```
waltz11% reidai8a
 範囲(xleft,ybottom,xright,ytop)?
-100 -100 100 100
 したいことを番号で選んで下さい。
  -1:メニュー終了，0:初期値のキーボード入力，1:初期値のマウス入力，
   2:change h,T(h= 0.0100,T=10.0000)
1
 マウスの左ボタンで初期値を指定して下さい(右ボタンで中止)。
```

もう一つの大事なことは、平衡点の周囲の流れがどうなるかは、微分法を使ってある程度まで解析できるということです。上の例題の右辺の f を微分してヤコビ行列を作ると、

$$\begin{pmatrix} 0 & 1 \\ -6-6x & -1 \end{pmatrix}$$

となりますが、平衡点 $\binom{0}{0}, \binom{-2}{0}$ での値はそれぞれ

$$A_1 = \begin{pmatrix} 0 & 1 \\ -6 & -1 \end{pmatrix}, \quad A_2 = \begin{pmatrix} 0 & 1 \\ 6 & -1 \end{pmatrix}$$

となります。$\binom{0}{0}$ の回りでの流れは $\dfrac{dx}{dt} = A_1 x$ の原点での流れに、$\binom{-2}{0}$ の回りでの流れは $\dfrac{dx}{dt} = A_2 x$ の原点での流れに似ている、ということです。

ちょっと詳しく解説　なんでヤコビ行列なんかが出て来るのか、不思議に感じる人がいるかも知れません。高校数学を思い出すと、「微分する = 接線の傾きを求める(接線を引く)」、という幾何学的理解が有効でした。接線の傾きを知るだけで結構色々なこと(関数がそこの近くで x の増加にともない増加しているのか、減少しているのか、極値となっているか等)が分かるということでした。曲線 $y = f(x)$ の点 $x = a$ における接線 $y = f'(a)(x-a) + f(a)$ とは、a の近くで、f を 1 次式で近似したもの、ということです(というか、そういうふうに解釈するのが、微分法の現代的な見方です)。大学の数学では、話が多次元になってしまって、微分することの意味が少し見え難くなりましたが、「微分するとは 1 次式で近似することだ」という認識は有効です。多次元の場合の 1 次式とは $Ax + b$ (A は行列、x,b はベクトルで、Ax は行列とベクトルのかけ算を表す)の形の式のことです。つまり $f(x)$ を点 a で微分して、微分係数(ヤコビ行列)が A になったということは、$f(x) \simeq A(x-a) + f(a)$ と考えられる、ということだったわけですね。a の近くでは、$A(x-a) + f(a)$ という 1 次式を調べるだけで、色々分かる、ということです。

問題 **8-1**　上の例題 8-1 の説明中に現われた力学系 $dx/dt = A_1 x$, $dx/dt = A_2 x$ について調べなさい。特に流れの図を描いてみて、力学系 (2) の流れと見比べて見なさい。

問題 **8-2**　次の力学系について、平衡点を調べ(前回のプリントの分類に従うと、どのタイプか？)、流れの様子を示す図を描きなさい。

$$\frac{d}{dt}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ -\sin x - y \end{pmatrix}.$$

(ヒント: 平衡点は $\binom{n\pi}{0}$ (n は整数) で、無限個ありますが、右辺は x につき周期 2π なので、$\binom{0}{0}$ と $\binom{\pi}{0}$ を調べれば十分です。この二つをすっぽり含むような範囲の図を書いて下さい。)

E.1.3　リミット・サイクル

前節では平衡点の解析の重要性を説明したのですが、2 次元の力学系には、もう一つ、周期運動を意味する閉軌道という大事なものがありました (単振動とかで既に見ましたね？)。

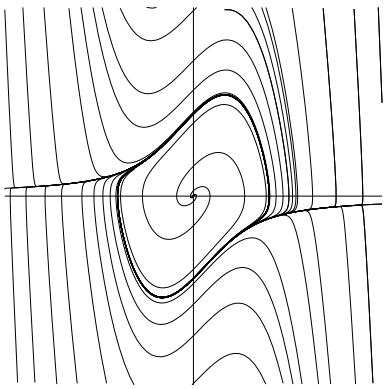
前回では、渦心点というものがあった時に、閉軌道が現われたのですが、一般の力学系では渦心点がなくても閉軌道が現われます。ここではもう面倒な理屈抜きで、それを見てもらいましょう。

例題 8-2　van der Pol[3] の方程式 $x'' + \mu(x^2 - 1)x' + x = 0$ (μ は正定数) を一階に直して出来る力学系

$$\frac{d}{dt}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ -x + \mu(1 - x^2)y \end{pmatrix} \tag{3}$$

の流れの様子を $-5 \leq x, y \leq 5$ の範囲で描きなさい。

上の例題と同様に `reidai8b.f` というプログラムと `rei8b.data` というサンプル・データを用意してあります。それを使って描いた図が



です。原点を回っている一つの閉軌道が目につきますが、特徴的なのは、その付近の軌道が、閉軌道にまつわりついて行っていることです。描画中の図を眺めていると分かりますが、どこからスタートしても速やかに閉軌道に近付いていきます。この種の閉軌道 (サイクル) のことをリミット・サイクル(極限閉軌道、limit cycle) と呼びます。

時間が経つと、はるか彼方に飛んでいってしまうような現象は別にして、時間によって変化する現象のうちの多くのものは長い時間が経つと、ある停止状態に落ち着くか (沈点)、周期運動 (極限閉軌道) に落ち着きます。2 次元の常微分方程式という簡単なモデルで、そういう現象を見ることが出来たわけです。

問題 8-3　次の方程式で定まる力学系の流れの様子を $-5 \leq x, y \leq 5$ の範囲で描きなさい。

$$\frac{d}{dt}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + y - x(x^2 + y^2) \\ -x + y - y(x^2 + y^2) \end{pmatrix}. \tag{4}$$

[3] ファンデルポール、と読みます。

微分方程式について、最後に一言　実は、世の中のすべての力学系は上に述べたようなものだけで十分説明し切れると (暗黙のうちに) 信じられていた時期があります。日常生活のレベルでは、今でもそう信じている人が多いでしょう。例えば、気象現象などで長年の平均値からずれたこと— 異常気象 — が生じるのは、何かこれまでにはなかった「異常な」原因があるに違いない、という推論をしたりします。これは、原因がシンプルならば結果もシンプルである (定常状態に落ち着いたり、きちんと周期的に変動するようになる) と信じているためです。でも、今ではこの「思い込み」が本当に正しいかどうかは、個々の場合を詳しく調べてみないと分からない、と考えられようになりました。それは計算機シミュレーションを通じてカオス (chaos、混沌) と呼ばれる現象が見つかったからです。それはシンプルなメカニズムで支配される系が、極めて複雑で、ほとんど予測不可能と言える結果を生じさせることを呼びます。カオスの発見の物語は、計算機と科学の研究の関係について色々考えさせてくれる、面白いものです。

E.2　追加の問題

問題 **8-4**　No.5 の方程式の解を求める問題では、二分法、Newton 法などの繰り返し計算によって、段々精度の高い近似解を得ることが出来たが、二つの方法で収束の速さがどのくらいか、対数グラフ (No.6 で紹介した) を利用して調べなさい。

問題 **8-5**　以下の、振り子の振動を記述する微分方程式の初期値問題を解きなさい。

$$\frac{d^2\theta}{dt^2} = -\frac{g}{\ell}\sin\theta. \tag{1}$$

ここで ℓ は振り子の長さ、g は重力加速度で (MKS 単位系では $g \approx 9.8m/sec^2$ です)、いずれも正の定数です。$\theta = \theta(t)$ は振り子の鉛直線からの傾きをあらわす量で、未知関数です。初期条件としては、オモリを時刻 $t = 0$ で、鉛直線から角度 α のところから、そっと手放すということを意味する

$$\theta(0) = \alpha, \quad \theta'(0) = 0 \tag{2}$$

を課します。周期 T を求めてみなさい。特に初期角度 α を色々変えたとき、どうなるか調べなさい (振り子の等時性は成り立っていますか？)。

問題 **8-6**　3 次元の力学系のうち、Chaos で有名な、いわゆる Lorenz Model

$$\begin{aligned} x'(t) &= -\sigma x + \sigma y \\ y'(t) &= rx - y - xz \\ z'(t) &= -bz + xy \end{aligned}$$

に初期条件

$$\begin{aligned} x(0) &= x_0 \\ y(0) &= y_0 \\ z(0) &= y_0 \end{aligned}$$

を課した初期値問題を解いて、相図を描け。ここで σ, r, b は正定数。このパラメータの選定は重要な意味を持つが、 まずは Lorenz が例として選んだという値 ($\sigma = 10$, $r = 28$, $b = 8/3$) を試してみよ。

E.3 プログラム

E.3.1 dynamics.f

```
* dynamics-f.f
************************************************************************
************************************************************************
*
*   2 次元の定数係数線形常微分方程式
*       x'(t) = a x + b y
*       y'(t) = c x + d y
*  に初期条件
*       x(0)=x0
*       y(0)=y0
*  を課した常微分方程式の初期値問題を解いて、相図を描く。
*
*  このプログラムは次の 4 つの部分から出来ている。
*     program dyna
*       主プログラム
*       行列の係数の入力、ウィンドウのオープン等の初期化をした後に、
*       ユーザーにメニュー形式でコマンドを入力してもらう。
*       実際の計算のほとんどは他の副プログラムに任せている。
*     subroutine draworbit(x0,y0,h,tlimit)
*       (x0,y0) を初期値とする解の軌道を描く。
*       刻み幅を h、追跡時間を tlimit とする。
*       近似解の計算には Runge-Kutta 法を用いる。
*     function fx(x,y)
*       微分方程式の右辺の x 成分
*     function fy(x,y)
*       微分方程式の右辺の y 成分
************************************************************************
************************************************************************
      program dyna
*      初期値
      real x0,y0
*      時間の刻み幅、追跡時間
      real h,tlimit,newh,newT
*      係数行列 A の成分 ( common 文により function fx,fy と共有する )
      real a, b, c, d
      common /coef/a,b,c,d
*      ウィンドウに表示する範囲 ( common 文により draworbit と共有する )
      real xleft, xright, ybottom, ytop
      common /region/xleft, xright, ybottom, ytop
*      メニューに対して入力されるコマンドの番号
      integer cmd
*      マウスのボタンの状態
      integer lbut, mbut, rbut
*      ウィンドウに表示する範囲の設定
      xleft = -1.0
      xright = 1.0
      ybottom = -1.0
```

```
      ytop = 1.0
*      時間刻み幅、追跡時間 ( とりあえず設定 )
      h=0.01
      tlimit=10.0
*      行列の成分を入力
      write(*,*) ' a,b,c,d='
      read(*,*) a,b,c,d
*      ウィンドウを開く
      call openpl
      call fspace(xleft, ybottom, xright, ytop)
*      x 軸、y 軸を描く
      call fline(xleft, 0.0, xright, 0.0)
      call fline(0.0, ybottom, 0.0, ytop)
      call xsync
*      メイン・ループの入口
 100  continue
*      メニューを表示して、何をするか、番号で選択してもらう
      write(*,*) ' したいことを番号で選んで下さい。'
*      write(*,*) '  -1:finish, 0:keyboard input x0,y0, 1:mouse input,'
      write(*,*) '  -1:メニュー終了, 0:初期値のキーボード入力,',
     -       ' 1:初期値のマウス入力,'
      write(*,1000) h,tlimit
 1000 format('   2:change h,T(h=',f7.4,',T=',f7.4,')')
* 1000 format('   2:刻み幅, 追跡時間変更 (現在 h=',f7.4,',T=',f7.4,')')
      read(*,*) cmd
*      番号 cmd に応じて、指示された仕事をする
      if (cmd .eq. 0) then
*       初期値の入力
        write(*,*) ' 初期値 x0,y0='
        read(*,*) x0,y0
        call draworbit(x0,y0,h,tlimit)
      else if (cmd .eq. 1) then
*
 200    continue
        write(*,*) 'マウスの左ボタンで初期値を指定して下さい',
     -             '( 右ボタンで中止 )。'
        call fmouse(lbut, mbut, rbut, x0, y0)
if (lbut .eq. 1) then
           write(*,*) '(x0,y0)=', x0,y0
    call draworbit(x0,y0,h,tlimit)
    goto 200
else if (mbut .eq. 1) then
    write(*,*) '(x0,y0)=', x0,y0
    call draworbit(x0,y0,-h,tlimit)
    goto 200
endif
        write(*,*) 'マウスによる初期値の入力を打ち切ります。'
      else if (cmd .eq. 2) then
*      時間刻み幅、追跡時間の変更
        write(*,*) ' 時間刻み幅 h, 追跡時間 T='
        read(*,*) newh, newT
        if (newh .ne. 0 .and. newT .ne. 0) then
          h = newh
          tlimit = newT
          write(*,*) ' 新しい時間刻み幅 h = ', h,
     -               ' 新しい追跡時間 T = ', tlimit
        else
          write(*,*) ' h=', h, ' T=', newT, ' は不適当です。'
```

```
        endif
      else if (cmd .eq. -1) then
*      終了 -- メイン・ループを抜ける
        goto 999
      endif
*      メイン・ループの先頭に戻る
      goto 100
*
 999  continue
      call mkplot("dynamics-f.plot")
      write(*,*) 'fplot ウィンドウを左ボタンでクリックして下さい'
      call closepl
      end
************************************************************************
************************************************************************
*      指示された初期値に対する解軌道を描く
      subroutine draworbit(x0,y0,h,tlimit)
      real x0,y0,h,tlimit
      real x,y,fx,fy
      real k1x,k1y,k2x,k2y,k3x,k3y,k4x,k4y,t
      real xleft, xright, ybottom, ytop
      common /region/xleft, xright, ybottom, ytop
*      時刻を 0 にセットする
      t = 0.0
*      初期値のセット
      x = x0
      y = y0
*      初期点を描く
      call fpoint(x,y)
*      ループの入口
 100  continue
*      Runge-Kutta 法による計算
*      k1 の計算
      k1x = h * fx(x, y)
      k1y = h * fy(x, y)
*      k2 の計算
      k2x = h * fx(x + k1x / 2.0, y + k1y / 2.0)
      k2y = h * fy(x + k1x / 2.0, y + k1y / 2.0)
*      k3 の計算
      k3x = h * fx(x + k2x / 2.0, y + k2y / 2.0)
      k3y = h * fy(x + k2x / 2.0, y + k2y / 2.0)
*      k4 の計算
      k4x = h * fx(x + k3x, y + k3y)
      k4y = h * fy(x + k3x, y + k3y)
*      (Xn+1,Yn+1) の計算
      x = x + (k1x + 2.0 * k2x + 2.0 * k3x + k4x) / 6.0
      y = y + (k1y + 2.0 * k2y + 2.0 * k3y + k4y) / 6.0
*        解軌道を延ばす
      call fcont(x,y)
*      時刻を 1 ステップ分進める
      t = t + h
*        まだ範囲内かどうかチェック
      if (abs(t) .le. abs(tlimit)) goto 100
*        サブルーチンを抜ける
999   continue
*      確実に描画が終るのを待つ
      call xsync
      end
```

```
************************************************************************
************************************************************************
*      微分方程式の右辺のベクトル値関数 f の x 成分
      real function fx(x,y)
      real x,y
      real a, b, c, d
      common /coef/a,b,c,d
      fx = a * x + b * y
      end
*      微分方程式の右辺のベクトル値関数 f の y 成分
      real function fy(x,y)
      real x,y
      real a, b, c, d
      common /coef/a,b,c,d
      fy = c * x + d * y
      end
```

E.3.2 dynamics.c

```
/*
 * dynamics.c
 */

/************************************************************************
 ************************************************************************
 *
 *   2 次元の定数係数線形常微分方程式
 *       x'(t) = a x + b y
 *       y'(t) = c x + d y
 *  に初期条件
 *       x(0)=x0
 *       y(0)=y0
 *  を課した常微分方程式の初期値問題を解いて、相図を描く。
 *
 *  このプログラムは次の 4 つの部分から出来ている。
 *     main()
 *       主プログラム
 *       行列の係数の入力、ウィンドウのオープン等の初期化をした後に、
 *       ユーザーにメニュー形式でコマンドを入力してもらう。
 *       実際の計算のほとんどは他の副プログラムに任せている。
 *     draworbit(x0,y0,h,tlimit)
 *       (x0,y0) を初期値とする解の軌道を描く。
 *       刻み幅を h、追跡時間を tlimit とする。
 *       近似解の計算には Runge-Kutta 法を用いる。
 *     fx(x,y)
 *       微分方程式の右辺の x 成分
 *     fy(x,y)
 *       微分方程式の右辺の y 成分
 ************************************************************************/

#include <stdio.h>
#include <math.h>
#include "fplot.h"

/* 係数行列 A の成分 ( common 文により function fx,fy と共有する ) */
double a, b, c, d;
```

```
/* ウィンドウに表示する範囲 ( common 文により draworbit と共有する ) */
double xleft, xright, ybottom, ytop;

void draworbit(double, double, double, double);

int main()
{
    /* 初期値 */
    double x0, y0;
    /* 時間の刻み幅、追跡時間 */
    double h, tlimit, newh, newT;
    /* メニューに対して入力されるコマンドの番号 */
    int cmd;
    /* マウスのボタンの状態 */
    int lbut, mbut, rbut;
    /* ウィンドウに表示する範囲の設定 */
    xleft = -1.0;
    xright = 1.0;
    ybottom = -1.0;
    ytop = 1.0;
    /* 時間刻み幅、追跡時間 ( とりあえず設定 ) */
    h = 0.01;
    tlimit = 10.0;
    /* 行列の成分を入力 */
    printf(" a,b,c,d=");
    scanf("%lf %lf %lf %lf", &a, &b, &c, &d);
    /* ウィンドウを開く*/
    openpl();
    fspace2(xleft, ybottom, xright, ytop);
    /* x 軸、y 軸を描く */
    fline(xleft, 0.0, xright, 0.0);
    fline(0.0, ybottom, 0.0, ytop);
    xsync();
    /* メイン・ループの入口 */
    while (1) {
        /* メニューを表示して、何をするか、番号で選択してもらう */
        printf("したいことを番号で選んで下さい。\n");
        printf(" -1:メニュー終了, 0:初期値のキー入力, 1:初期値のマウス入力,");
        printf("  2:刻み幅 h, 追跡時間 T 変更 (現在 h=%7.4f, T=%7.4f)\n",
               h, tlimit);
        scanf("%d", &cmd);
        /* 番号 cmd に応じて、指示された仕事をする */
        if (cmd == 0) {
            /* 初期値の入力*/
            printf(" 初期値 x0,y0=");
            scanf("%lf %lf", &x0, &y0);
            draworbit(x0, y0, h, tlimit);
        } else if (cmd == 1) {
            while (1) {
                printf("マウスの左ボタンで初期値を指定して下さい (右ボタンで中止)\n");
                fmouse(&lbut, &mbut, &rbut, &x0, &y0);
                if (lbut == 1) {
                    printf("(x0,y0)=(%g,%g)\n", x0, y0);
                    draworbit(x0, y0, h, tlimit);
                } else if (mbut == 1) {
                    printf("(x0,y0)=(%g,%g)\n", x0, y0);
                    draworbit(x0, y0, -h, tlimit);
                } else {
```

```
                    printf("マウスによる初期値の入力を打ち切ります。\n");
                    break;
                }
            }
        } else if (cmd == 2) {
            /* 時間刻み幅、追跡時間の変更 */
            printf(" 時間刻み幅 h, 追跡時間 T= ");
            scanf("%lf %lf", &newh, &newT);
            if (newh != 0 && newT != 0) {
                h = newh;
                tlimit = newT;
                printf("新しい時間刻み幅 h = %g, 新しい追跡時間 T = %g\n",
                       h, tlimit);
            } else {
                printf(" h=%g, T=%g は不適当です。\n", newh, newT);
            }
        } else if (cmd == -1) {
            /* 終了 -- メイン・ループを抜ける */
            break;
        }
    }
    mkplot("dynamics.plot");
    printf("fplot ウィンドウを左ボタンでクリックして下さい\n");
    closepl();
    return 0;
}

/*************************************************************************/
/*************************************************************************/
/*      指示された初期値に対する解軌道を描く*/
void draworbit(double x0, double y0, double h, double tlimit)
{
    double x, y, fx(), fy();
    double k1x, k1y, k2x, k2y, k3x, k3y, k4x, k4y, t;
    /* 時刻を 0 にセットする */
    t = 0.0;
    /* 初期値のセット */
    x = x0;
    y = y0;
    /* 初期点を描く */
    fpoint(x, y);
    /* ループの入口 */
    do {
        /* Runge-Kutta 法による計算 */
        /* k1 の計算 */
        k1x = h * fx(x, y);
        k1y = h * fy(x, y);
        /* k2 の計算 */
        k2x = h * fx(x + k1x / 2.0, y + k1y / 2.0);
        k2y = h * fy(x + k1x / 2.0, y + k1y / 2.0);
        /*  k3 の計算 */
        k3x = h * fx(x + k2x / 2.0, y + k2y / 2.0);
        k3y = h * fy(x + k2x / 2.0, y + k2y / 2.0);
        /*  k4 の計算 */
        k4x = h * fx(x + k3x, y + k3y);
        k4y = h * fy(x + k3x, y + k3y);
        /*  (Xn+1,Yn+1) の計算 */
        x += (k1x + 2.0 * k2x + 2.0 * k3x + k4x) / 6.0;
```

```
        y += (k1y + 2.0 * k2y + 2.0 * k3y + k4y) / 6.0;
        /* 解軌道を延ばす */
        fcont(x, y);
        /* 時刻を 1 ステップ分進める */
        t += h;
        /* まだ範囲内かどうかチェック */
    }
    while (abs(t) <= fabs(tlimit));
    /* サブルーチンを抜ける */
    /* 確実に描画が終るのを待つ */
    xsync();
}

/*************************************************************************/
/*************************************************************************/
/* 微分方程式の右辺のベクトル値関数 f の x 成分 */
double fx(double x, double y)
{
    return a * x + b * y;
}

/* 微分方程式の右辺のベクトル値関数 f の y 成分 */
double fy(double x, double y)
{
    return c * x + d * y;
}
```

E.4 参考文献

専門家による解説としては、三井 [3], [4] がある。[3] の方が詳しいが、[4] は後から書かれているだけに、要領よくまとまっている ([4] を通読し、必要に応じて [3] を参照すると良いだろう)。

解説の分かりやすさに定評ある二氏による一松 [2], 戸川 [1] も奨めたい。特に戸川 [1] は C のプログラムつき (ネットで公開) である。

少し雰囲気が違うが、杉原による解説 [5] も読むに価する。

関連図書

[1] 戸川隼人, UNIX ワークステーションによる科学技術計算ハンドブック, サイエンス社 (1992).
http://www.mscom.or.jp/~tog/anna/ でプログラムが公開されている。

[2] 一松 信 (ひとつまつ しん), 数値解析, 朝倉書店 (1982).

[3] 三井 斌友 (たけとも), 数値解析入門, 朝倉書店 (1985).

[4] 三井 斌友, 微分方程式の数値解法 I, 岩波書店 (199?).

[5] 渡部 力, 名取 亮, 小国 力監修, Fortran 77 による数値計算ソフトウェア, 丸善 (1982).
題名からするとプログラムのみの本のようであるが、そうではなく、常微分方程式の項は杉原正顕氏によるすぐれた解説がある。

付 録A　ヒント集

A.1　インデンテーション

A.1.1　GNU indent

プログラムのインデンテーション (字下げ) を自動的に行なうプログ ラムとして、cb (C beautifulier) や indent がある。

私が個人的に好きなのは、-kr オプションです。これはかつて C 言語のバイブルと言われた、カーニハン＆リッチー著『プログラミング言語 C』で使われているインデンテーションの流儀に整形するものです。

vi 使いにお勧め？ — K&R スタイル

```
oyabun% indent -kr myprog.c
```

日頃、Mule などの GNU Emacs のデフォールトの c-mode を使ってプログラムを書いている場合は、オプションなしか、あるいは -br オプションを使うのが良いと思われます。

Mule 使いにお勧め？

```
oyabun% indent -br myprog.c
```

A.2　参考になる文書

『C 言語プログラミング FAQ』http://www.catnet.or.jp/kouno/c_faq/c_faq.html
『C プログラミング診断室』http://kjmcci.kct.ne.jp/~fuji/mybooks/cdiag/index.html
ネットニュース fj.comp.lang.c

付 録B　がらくた箱

B.1　参考書

C 言語の勉強は

1. カーニハン＆リッチー、プログラミング言語 C 第 2 版、共立出版
2. 柴田望洋, ,

などがある。 1. は C 言語のバイブルであるが、翻訳の質が低いと良く言われている。出来れば英語で読むことを勧める。

C 言語による数値計算についての参考書は良いものがない！本当に少ない。以下は (問題がないこともないが) 比較的まともな方である。

1. Numerical Recipes in C、邦訳 技術評論社
2. 戸川隼人、科学技術計算ハンドブック、サイエンス社

B.2　C 言語の種類

C 言語を勉強するために、最初に C 言語に種類があることを注意しておく。

C 言語も歴史が古いので、いくつか版があるが、現在では

K&R の C C のバイブルとも呼ばれた、Kernighan & Ritchie のプログラミング言語 C (第 1 版)、で解説されている古い C 言語のこと。

ANSI C K& R の C の抱えている問題を解決するために作られた、新しい c 言語の規格。ANSI の規格になったことから、ANSI C と呼ばれる。

の二通りのみ考えればいい。

理想を言えば、どちらも使えた方が良いが、今から勉強するならば、まず ANSI C を勉強することを勧める。

K & R C と ANSI C の最も大きな相違点は、後者が関数のプロトタイプ宣言をサポートしていることにある。後者を学んでおけば、C++ 言語への (従って Java への) 移行もスムーズに進められる。

古い K & R の C で (きちんと) 書かれたプログラムは、ANSI C のプログラムとしても大抵は正しいプログラムであるので、 ANSI C をサポートしたコンパイラーがあれば嬉しい。

B.3 UNIX 上の C コンパイラーの使い方

UNIX 互換の OS[1] で稼働しているシステムでは、 “cc” という名前の C コンパイラーが用意されていることが多い “cc” そのものがなくても、それと互換性を持つコンパイラーが用意されていることがほとんど。例えば GNU C compiler “gcc” などがそう。以下この小節の説明は cc でするが、gcc でも全く同様である。

- C プログラムのファイル名は末尾が “.c” でなければいけない[2]。
- 数学関数を使っていない C プログラム *foo*.c をコンパイルするには

  ```
  cc foo.c
  ```

  あるいは

  ```
  cc -o bar foo.c
  ```

  のようにする。前者の方法では “a.out” という名前の実行プログラム・ファイルが、後者の方法では “*bar*” という名前の実行プログラム・ファイルが出来る。(“*foo*”, “*bar*” はプログラムを書く人が自由に決めていいもので、 “*foo*”=“*bar*” でもよい。) 数学関数を用いている C プログラムをコンパイルするには “-lm” というオプションを指定する (“libm.a” を link する、という意味)。例えば

  ```
  cc -o foo.c -lm
  cc -o bar foo.c -lm
  ```

  とする。なお、このオプション “-lm” の指定位置には注意が必要で[3]、とりあえずは最後に置く、と覚えておいて欲しい。
- 明治大学数学科の講義＆演習では “cco” というコマンドのエイリアス (別名) 定義をしている。これは “~/.cshrc” の中に

  ```
  alias   cco     'cc -o \!^:r \!* -lm'
  ```

  と書くことで定義されている。この場合 “cco *myprog*.c” とすると、 “cc -o myprog myprog.c -lm” をしたのと同じことが起こり、“*myprog*” という名前の実行プログラムが出来る。
- 複数の C プログラム・ファイルをまとめてコンパイルするには、

[1] OS=operating system.

[2] ファイル名末尾の文字でファイルの種類を表すという慣習がある。“.f” は FORTRAN プログラム、“.p” は Pascal プログラム、“.s” はアセンブリー言語によるプログラム、 “.o” はコンパイル or アセンブルして出来たオブジェクト・プログラム、 “.a” はライブラリィ・アーカイブ、 etc.

[3] “-lm” はリンカーに与えるオプションのため、リンカー以外のオプションよりも後に指定しないといけない。意味が分からない場合は、なるべく最後、とすれば間違いないだろう。

1. 一斉にコンパイルするには “cc” コマンドに並べて与えればよい。例えば

   ```
   cc prog1.c prog2.c ··· prog$n$.c
   ```

2. 一つ一つコンパイルして最後にまとめるには、例えば

   ```
   cc -c prog1.c                          prog1.o を作る。
   cc -c prog2.c                          prog2.o を作る。
   ...
   cc -c prog$n$.c                        prog$n$.o を作る。
   cc prog1.o prog2.o ··· prog$n$.o       prog$x$.o をリンクして
                                          一つのプログラムにする。
   ```

   のようにする。

- プログラムの最適化をするには “-O” オプションを用いる。コンパイルに時間がかかるようになるが、出来上がった実行形式は効率的に動作する。

  ```
  cc -o foo.c
  cc -o bar -o foo.c
  cc -o -c foo.c
  ```

  などなど。長い計算時間がかかるプログラムをコンパイルする場合は必ず指定する。

- その他のオプション

  -S アセンブリ言語で出力する。*myprog*.c から *myprog*.s が出来る。

  -E プリプロセッサーのみ通す (マクロを展開する)。結果は標準出力に出す。

  -g (デバッガーでデバッギングするための) デバッグ情報をつける。

  -D 名前=値

- より詳しいオプションの情報は “man cc” で見られる。

ある程度、複雑な仕事をするようになったら、 make コマンドに慣れることを勧める。

B.4 数値データの種類、変数宣言

C 言語には数値を表すために色々なデータ型が使えるが、大きく分けると

整数型 int, char. (signed, unsigned, short, long などの修飾詞もあります)

浮動小数点型 float, double, long double

となる。初歩の数値処理プログラミングでは **int** と **double** さえ知っていればなんとかなるであろう。

int と double, どういうふうに使い分けるのか？

個数、順番、回数を表す量 (整数値しか取らない) を収めるには **int**,
その他 (小数部分が必要なもの) は **double**

としよう。注意すべきは、大は常に小を兼ねるとは限らず、すべてを double にする、などということは出来ない。(プログラムの字面上で判断する法) int を使って表すべきものとして、

- 配列の添字
- ほとんどすべての for 文のカウンター

などがあげられる。

入出力の方法　画面への出力[4]には printf() が、キーボードからの入力[5]には scanf() や fgets() が使える[6]。その際の書式指定は

int の場合 “%d” (=decimal) を使う。

```
#include <stdio.h>
main()
{
    int n;                          整数型の変数 n を宣言
    scanf("%d", &n);                変数 n に読み込む
    printf("%d\n", n*n);            n*n の値を出力
}
```

double の場合 これは

- 入力の書式指定には “%lf”, “%le”, “%lg” のいずれでも使える。どれも違いはない。
- 出力には
  (i) “%f” 小数形式で出力する。
  (ii) “%e” 指数形式で出力する。
  (iii) “%g” 小数形式、指数形式のうち表現が短い方を使って出力する。
  が使る。桁数などもっと細かいことも指定できるが、それは後述する。

[4]正確には標準出力への出力。
[5]正確には標準入力からの入力。
[6]scanf() については、使うべきでない、という意見が強いが、これについては後述する。

```
#include <stdio.h>

void test(double);

int main()
{
    test(1.0);
    test(1.23456);
    return 0;
}

void test(double A)
{
    int i;
    double a;
    a = A;
    for (i = 0; i < 10; i++) {
        printf("%f %e %g\n", a, a, a);
        a *= 10.0;
    }
    a = A;
    for (i = 0; i < 10; i++) {
        printf("%f %e %g\n", a, a, a);
        a /= 10.0;
    }
}
```

```
1.000000 1.000000e+00 1
10.000000 1.000000e+01 10
100.000000 1.000000e+02 100
1000.000000 1.000000e+03 1000
10000.000000 1.000000e+04 10000
100000.000000 1.000000e+05 100000
1000000.000000 1.000000e+06 1e+06
10000000.000000 1.000000e+07 1e+07
100000000.000000 1.000000e+08 1e+08
1000000000.000000 1.000000e+09 1e+09
1.000000 1.000000e+00 1
0.100000 1.000000e-01 0.1
0.010000 1.000000e-02 0.01
0.001000 1.000000e-03 0.001
0.000100 1.000000e-04 0.0001
0.000010 1.000000e-05 1e-05
0.000001 1.000000e-06 1e-06
0.000000 1.000000e-07 1e-07
0.000000 1.000000e-08 1e-08
0.000000 1.000000e-09 1e-09
1.234560 1.234560e+00 1.23456
12.345600 1.234560e+01 12.3456
123.456000 1.234560e+02 123.456
1234.560000 1.234560e+03 1234.56
12345.600000 1.234560e+04 12345.6
123456.000000 1.234560e+05 123456
1234560.000000 1.234560e+06 1.23456e+06
12345600.000000 1.234560e+07 1.23456e+07
123456000.000000 1.234560e+08 1.23456e+08
1234560000.000000 1.234560e+09 1.23456e+09
1.234560 1.234560e+00 1.23456
0.123456 1.234560e-01 0.123456
0.012346 1.234560e-02 0.0123456
0.001235 1.234560e-03 0.00123456
0.000123 1.234560e-04 0.000123456
0.000012 1.234560e-05 1.23456e-05
0.000001 1.234560e-06 1.23456e-06
0.000000 1.234560e-07 1.23456e-07
0.000000 1.234560e-08 1.23456e-08
0.000000 1.234560e-09 1.23456e-09
```

お勧めの設定

- とにかく表示されればいい、という場合は %g がお勧め。
- 1. 複数の数をきれいに並べたい[7]
  2. 計算した精度一杯まで表示したい[8]

  などの場合は、

[7]これは大事なことである。膨大な数を前にしても、人間は案外そこから何かを発見することが出来るもので、その作業を支援することを馬鹿にしてはいけない。

[8]C 言語で書かれたプログラムは、普通は倍精度計算をしているくせに、「お任せ」にすると単精度程度しか表示しない。高精度に計算した結果を保存する場合は注意しよう。

%全体の桁数.小数部分の桁数{f|e|g}

のような指定をする。例えば %12.5f とすると、全部で 12 桁の幅、小数点以下は 5 桁の幅を使うことになる。このとき、全体の桁数は小数の桁数より十分大きくするよう注意する必要がある。例えば %e で桁数を指定する場合、8 桁以上大きくする必要がある。内訳は

- 符号のための 1 桁
- 小数点より上の数字のための 1 桁
- 小数点のための 1 桁
- 指数部 ("e-128" のようなやつに最大) のための 5 桁

倍精度は 15 桁強あるので、 %23.15e とすると、計算精度分フルに表示できることになる。

B.5 数の型とのおつきあい

B.5.1 定数にも型がある

当然のことだが、定数にも型がある。

1	int
1.2	double
1e+2	double
3L	long int
1.2f または 1.2F	float
1.2l または 1.2L	long double

注意すべきことは[9]、

1.2 は float でなく double である !!!!!

くらいかな。初歩のうちは int と double だけで OK と書いたので、接尾子 "L", "F" は覚えなくて良いということである[10]。

クイズ　あるプログラムの仕様に、「入力として 1.23 以下の数値のみ受け付ける」とあったので、次のようなプログラムを書いた。

```
float x;
....
if (x > 1.23) {
   受付を拒否
}
```

ところが

[9]実に多くの人が誤解しているよなあ。
[10]まあ、キャストするという手はあるけど。

```
chronos% ./test
1.23
greater than 1.23
chronos%
```

のように 1.23 ははじかれてしまった (こうなるかどうかはシステムによるが)。なんでだろう?

B.5.2 式の中の型変換について

プログラムの中では、int と double など複数の型のデータが混在するわけだが、式の中に現れた場合、大雑把に言って「上の型に変換してから計算する」というルールがある。例えば int データと double データの四則は、 int データを double に変換してから演算を行なう。

例えば、x, a, h が double で i が int の場合、

```
x = a + i * h;
```

では、i を double に変換してから h とかけ算されて、その結果を a と足した結果を x に代入する。

B.5.3 int の割算に注意

int 同士のデータの割算はいわゆる整商が計算される。つまり「余りのある割算をしたときの商」、言い替えると「商の整数部分」になることに注意しよう。例えば

```
7 / 2          は 3
1 / 10         は 0
```

となる。

初心者のうちにありがちなミスに

```
int n;
double h;
....
h = 1 / n;
```

のようなものがある。これは h に $\frac{1}{n}$ を代入しているつもりなのだろうが、とんでもない結果 (大抵は 0) が代入される。

```
h = 1.0 / n;
```

とすべきである。この場合、分子が double 型の定数だから、n は double 型に変換されてから割算が実行される。分子、分母ともに int 型の定数であるような場合は、

```
int num_win, num_games;
double rate;
...
rate = (double) num_win / num_games;
```

のように少なくとも一方のデータを double にキャストすれば良い。

B.6　数学関数の使い方

初等関数など、よく使われる数学関数が用意されている。それを使うには

- “`#include <math.h>`” として関数の宣言をインクルードする。一度 “`/usr/include/math.h`” を読んでみよう。
- リンクする時に “`-lm`” を指定する。
- プロトタイプ宣言の使えない古い C では、関数の引数は必ず double になるように注意しよう。例えば (以下の例では n は整数データだとします)。

`sqrt(1)`	誤り	`sqrt(1.0)` あるいは `sqrt((double)1)`	正しい
`log(n)`	誤り	`log((double) n)`	正しい

B.7　配列の使い方

B.7.1　添字は 0 から

“`double a[4];`” と宣言すると、double 型のデータ 4 個分のメモリーが確保され、“`a[0]`”, “`a[1]`”, “`a[2]`”, “`a[3]`” という名前で読み書きできる。添字が 0 から始まるため、“`a[4]`” はないことに注意。(FORTRAN では、“`real a(4)`” とすると、`a(1)`, `a(2)`, `a(3)`, `a(4)` の 4 個が使える。

FORTRAN や Pascal では、

FORTRAN `a(-10:20)`

Pascal `a:  array [-10..20] of real;`

のように、任意の番号から任意の番号までを添字に持つような配列が宣言できるが、C ではこの種のことは出来ない。これを何とか実現させるためのダーティートリックもあるが[11]、可搬性に疑問があるし、ここでは紹介しないことにする。

B.7.2　配列の大きさは整定数

配列の添字はコンパイル時に決めないといけない (実行時にはもう変えられない)。

```
/* 間違っているプログラム */
#include <stdio.h>
main()
{
    double a[n];
    int n;
```

[11] Numerical Recipes in C を参照。

```
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%lf", &a[i]);
    ....
}
```

のようなプログラムは駄目だということになる。このプログラムの修正の方法は二通りあって、

1. 十分な大きさの配列を宣言する。例えば、n が大きくても 1000 は越えないと判断したら、最初から大きさ 1000 の配列を宣言して使う、という手があります。

```
#include <stdio.h>
#define MAXN 1000
main()
{
    double a[MAXN];
    int n;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%lf", &a[i]);
    ...
}
```

2. 配列でなくて、ポインターを利用する。

```
#include <stdio.h>
main()
{
    double *a;
    int n;
    scanf("%d", &n);
    if ((a = (double *)malloc(sizeof(double) * n)) == NULL) {
         fprintf(stderr, "必要なメモリーが確保できませんでした\n");
         exit(1);
    }
    for (i = 0; i < n; i++)
        scanf("%lf", &a[i]);
    ...
}
```

のようにします。

B.8 ANSI C のプロトタイプ宣言とは？

プロトタイプ宣言なしのプログラム例

```
#include <stdio.h>
main()
{
  double solution();
  printf("2 x = 3 の解=%g\n", solution(2.0, 3.0));
}

double solution(a, b)
double a, b;
{
  if (a == 0) {
    fprintf(stderr, " この方程式は不定または不能です。\n");
    exit(1);
  }
  else
    return b / a;
}
```

プロトタイプ宣言を使ったプログラム例

```
#include <stdio.h>
main()
{
  double solution(double, double);
  printf("2 x = 3 の解=%g\n", solution(2, 3));
}

double solution(double a, double b)
{
  if (a == 0) {
    fprintf(stderr, " この方程式は不定または不能です。\n");
    exit(1);
  }
  else
    return b / a;
}
```

B.9 コンパイラーなどのエラーメッセージを読むための単語帳

warning まず読み方であるが「ウォーニング」である。断じて「ワーニング」ではない。警告という意味。エラーではないから、実は問題ない、ことも多いが、注意すること。

parameter パラメーター。関数の引数 (argument) のこと。

declaration 宣言。変数宣言と関数の宣言。

previous 前の。

suffix ファイル名の末尾の “.” の後にある部分のこと。“File with unknown suffix” とは、 “.c”, “.f”, “.p”, “.s”, “.o” のような (システムが) 知っているサフィックスでないという意味。例えば、DOS と勘違いして、

```
cc myprog
```

と “.c” を省略したり、

```
cc myprog.C
```

のように大文字にすると出る。

type 型。(type mismatch = 型があっていない。double とすべきを int にしてしまったとか。)

unclassifiable 分類不可能な、という意味。 “unclassifiable statement” この文、何が何だか、さっぱり分からないよ、という意味。まあ、ひどい文法エラーをしたということ。

B.10 “Segmentation fault”, “Bus error” って何ですか？

本来、そのプロセスが読み書きしてはいけないメモリー内の領域を読み書きしようとすると起きるエラーで、同じ意味のものに “Segmentation Violation”, よく似たものに “Bus Error[12]” がある。 がある。

これらの原因としては

1. ポインターの使い間違い。例えば

   - scanf で変数に値を設定する際に “&” が抜けている。

     ```
     int a;
     ...
     scanf("%d", a);
     ```

   - 配列の添字が負になったり、宣言した大きさを越えている。

     ```
     double a[10];
     ...
     for (i=0;i<10;i++)
       s += a[i-1];      /* a[-1] を読む */
     ...
     a[10] = x;          /* a[9] までなのに、a[10] に書く */
     ```

2. 関数への引数受渡しのエラー

[12]例えば SPARCstation の主記憶はバイト単位でアドレスがついている (バイト・マシン) が、メモリーの読み書きの単位は 4 バイトが基本で、命令によってはアドレスが 4 の倍数でないとエラーが出る、など。

などがある。

このエラーはいきなりプログラム終了となることが多い。まず “**printf デバッグ**” や、デバッガーによるデバッギングを行なって場所を特定すること。探し方は、二分探索。これにつきる。

printf デバッグ　プログラムの状況を表示するための printf 文をあちこちに書き加える手法。もっとも基本的。

B.11　数学定数どうしよう？

例えば π などはどうしよう？

1. 自分で定義する。

   ```
   #define pi 3.14159265358979323846
   ```

   あるいは

   ```
   double pi = 3.14159265358979323846;
   ```

   この方法は

   - こんなの覚えてらんないぜ
   - 打ち間違いが危ない
   - 途中でゴミが入っても気がつきにくい

   などの短所がある。

2. “math.h” あるいは “values.h” の中の定義を利用する。

   ```
   #include <math.h>
   #define PI M_PI
   ```

   ポータビリティー[13]があるかどうか少し心配。

3. 数学関数を利用して計算する。

   ```
   #include <math.h>
   ...
   double PI = 4.0 * atan(1.0);
   ```

   この方法の欠点は、(i), (ii) のような単純な定義、代入文とは異なり、初等関数の呼び出しはコンパイル時にしてくれないので、初期化をするところをきちんと用意しないといけないことである。例えば

[13] 可搬性。

```
#include <stdio.h>
#include <math.h>
double PI = 4.0 * atan(1.0);
main()
{
   ...
}
```

は間違いで、

```
#include <stdio.h>
#include <math.h>
double PI;
main()
{
   ...
   (宣言文の終り)
   PI = 4.0 * atan(1.0);
}
```

のようにしないといけない。

B.12 C で負の添字を使う方法

C 言語の配列の添字は 0 から始まりますが、何とかこれをこちらの好きなものに変更する方法はないでしょうか？この問題はささいなことのようですが、不可能となると、不自然なプログラミングを強要されることになりかねないので困ってしまいます。

他の言語では、任意の整数から添字を始めることが出来るものがあります。例えば -2 から 5 までを添字として使いたい場合は

FORTRAN `real a(-2:5)`

Pascal `a:  array [-2..5] of real;`

のように出来ます。

それで C 言語の場合はどうかというと、残念ながら正の整数から添字を始める標準的な方法はありませんが、負の整数から添字を始める方法は場合によっては、一応存在します。

```
/*
 * negative-index.c
 */

#include <stdio.h>

#define LOWER (-2)
```

```
#define HIGHER 5

int main()
{
  int i;
  double adummy[HIGHER-LOWER+1], *a;

  a = adummy - LOWER;
  for (i = LOWER; i <= HIGHER; i++) {
    a[i] = i;
  }
  for (i = LOWER; i <= HIGHER; i++)
    printf("a[%d]=%f\n", i, a[i]);
  for (i = 0; i <= HIGHER-LOWER; i++)
    printf("adummy[%d]=%f\n", i, adummy[i]);

  return 0;
}
```

```
a[-2]=-2.000000
a[-1]=-1.000000
a[0]=0.000000
a[1]=1.000000
a[2]=2.000000
a[3]=3.000000
a[4]=4.000000
a[5]=5.000000
adummy[0]=-2.000000
adummy[1]=-1.000000
adummy[2]=0.000000
adummy[3]=1.000000
adummy[4]=2.000000
adummy[5]=3.000000
adummy[6]=4.000000
adummy[7]=5.000000
```

つまり、a[-2] が adummy[0] を指すように、ポインター a を細工します。実際には a[0] が ad[2] を指すように a = ad+2 とすれば OK です。

なぜ正の添字からは始められないか？: 正の添字、例えば 2 から始めるには、上と同様に考えて a = ad-2 とすればいいと思うかも知れません。また実際それで動くこともあるかも知れませんが、これは一般には駄目です。それはこの場合 a[0] が ad[-2] に相当するアドレス (これは不正です!) を指すことになってしまうからです。a[0] は使うつもりがないわけですが、これを認めない処理系も存在することと思われます。

B.13 scanf() を使うのは邪道だと言われちゃった

そもそも scanf というのは、お節介と言ってもいいくらい、複雑な処理をすることが出来る関数なんです。そういう奴が間違った入力を受けとると、ずいぶんと間抜けなことをしてくれ

ます。 scanf() は正常に読みとれた値の個数を返しますから、それをチェックすればいいと考える人もいるでしょうが、実は駄目です。

```
/* とりあえず正しいつもりのプログラム */
#include <stdio.h>

int main()
{
    double a;
    while (1) {
        if (scanf("%lf", &a) == 0) {
            fprintf(stderr, " scanf() で 0 個しか正常に読み込めませんでした。\n");
        }
        else {
            printf("%f\n", a);
            exit(0);
        }
    }
    return 0;
}
```

これを実行すると以下のようになります。

```
oyabun% test-scanf
1.2
1.200000
oyabun% test-scanf
0.000000
oyabun% scanf
1,2
1.000000
oyabun% scanf
,2
 scanf() で 0 個しか正常に読み込めませんでした。
 scanf() で 0 個しか正常に読み込めませんでした。
 scanf() で 0 個しか正常に読み込めませんでした。
略
 scanf() で 0 個しか正常に読み込めませんでした。
 scanf() で 0 個しか正常に読み込めませんでした。
^C scanf() で 0 個しか正常に読み込めませんでした。

oyabun%
```

つまり “,” という読み込めない文字があると、ポインターを先に進めずに終了するため、「引っかかってしまって」無限ループに落ちるわけです。このような欠点がありますから、システム・プログラム等で scanf() を使うのは確かに馬鹿者です。

では正しい入力はというと、

1. とりあえず fgets() で文字列として読み込む。
2. 読み込んだ文字列を次のいずれかで解析する。

(a) sscanf() 個数チェックが出来る。

(b) atof() 不正入力と 0 入力が区別出来ない。

(c) strtod() どこまで解析出来たかの位置まで分かる (ほぼ完璧なチェック)。

として行なうことになります。

```
#include <stdio.h>
#include <stdlib.h>             /* atof() のため */
#include <floatingpoint.h>        /* strtod() */

int main()
{
    double a;
    char input[100];
    if (fgets(input, sizeof(input), stdin) == NULL) {
        fprintf(stderr, "input error?\n");
    exit(1);
    }
    /* atof() の利用 */
    a = atof(input);
    printf("%f\n", a);
    /* sscanf() の利用 */
    if (sscanf(input, "%lf", &a) == 0) {
        fprintf(stderr, " sscanf() で 0 個しか正常に読み込めませんでした。\n");
    }
    else
        printf("%f\n", a);
    /* strtod() の利用 */
    {
    char *p;
    a = strtod(input, &p);
    if (input == p) {
        fprintf(stderr, " strtod() が読み込みに失敗しました\n");
        exit(1);
    }
    printf("%f\n", a);
    }
    return 0;
}
```

どうですか？めんどい、やってらんない、と思う人が多いと想像します。筆者は、普通の数値計算実行者が作る、いわゆるシステム・プログラムでないプログラムでは、それほど神経質になる必要はなく、「あ、入力を間違えた！しかたないから、最初からやりなおそう」でいいと思います。ただし、長い計算の後にユーザーのキーボードからの指示を読んで態度を決めるようなプログラムでは、それまでの計算結果を無駄にしないように、それなりの注意がいるでしょう。その場合も sscanf() でチェックする程度でよいと思います。

B.14　ポインターについてどれだけ知ればいいですか？

俗に

ポインターを使いこなすことが出来るかどうかが、
C 言語をマスター出来るかどうかの分かれ道

と言われますが、かなりの部分まで真実です。

数値処理プログラミングでは、システム・プログラミングにおいては必須となるような事項 (リストや木など) でも、使わずにすますことが出来る場面が多いですが、それですべて通そうとするのは窮屈です。

次の節で説明するように、C 言語では行列を表現するために 2 次元配列を使うのはあまり適当でなく、ポインター配列を使う方が望ましいと思われる場合が多いので、ポインターと無縁で過ごすのは、ほぼ不可能と思えます。

B.15　行列はどうする？

B.15.1　はじめに

微分方程式の数値シミュレーションのプログラムには、大きな行列 (や二重添字を持つ数列[14]) が現れます。FORTRAN では、行列を表すのに、2 次元配列を用いるのが普通です。しかし C 言語には整合配列の機能がないので、2 次元配列を用いると、ポータビリティーのある関数を作るのが難しくなります。そこで、次のような工夫をします。

1 次元配列の方法 1 次元配列、あるいはポインターを用いて領域を確保し、二重添字から自前で 1 次元的な番地を計算して、アクセスする

ポインター配列の方法 ポインター配列、あるいはポインターのポインターを用いて確保した領域を 2 次元配列的な記法でアクセスする

B.15.2　1 次元配列の方法

例えば $m \times n$ 型の行列 A を表現して、計算に用いるために

領域の確保

```
double a[m * n];
```

のように 1 次元配列 a[] を宣言するか、あるいは (こちらの方がずっと C らしいが)

```
double *a;
```

とポインターを宣言しておいて、

[14] 例えば、長方形領域を格子に切って、格子点上での関数値を扱う場合など。

```
if ((a = (double *)malloc(sizeof(double) * m * n)) == NULL) {
    エラーの場合の処理
}
```

のように動的に記憶領域を確保する。

成分へのアクセス　(i,j) 成分 $(0 \leq i \leq m-1, 0 \leq j \leq n-1)$ をアクセスする時は面倒だが

```
a[n * i + j]
```

のように書くか、どこかで

```
#define A(i,j) a[n*(i)+(j)]
```

のようなマクロを定義しておいて

```
A(i,j)
```

と書く。

B.15.3　ポインター配列の方法

例えば $m \times n$ 型の行列 A を表現して、計算に用いるために

領域の確保 (準備その 1)　(ここは 1 次元配列の方法と同様。)

```
double *abody;
```

とポインターを宣言しておいて、

```
if ((abody = (double *)malloc(sizeof(double) * m * n)) == NULL) {
    エラーの場合の処理
}
```

のように動的に記憶領域を確保する。

ポインター配列の設定 (準備その 2)

```
double **a;
....
if ((a == (double *)(malloc(sizeof(double *) * m))) == NULL) {
    エラーの場合の処理
}
```

のようにポインターへのポインターを宣言して、必要な領域を確保した後で、行列の各行の先頭のアドレスを

```
for (i = 0; i < m; i++)
    a[i] = abody + i * n;
```

とセットする。ここまでは大変だが、

成分のアクセス　(i, j) 成分 $(0 \leq i \leq m-1, 0 \leq j \leq n-1)$ をアクセスする時は単に

```
a[i][j]
```

と書けばよい。

B.15.4　二つの方法の優劣

いずれが優れているか、決定打はないという感じである。

- 一次元配列の方法は、二重添字から番地を計算するのがとにかく面倒である (マクロで一応は処理できるが、行列が増えると個数分のマクロを定義せねばならず、ちょっとイヤミ)。
- 一次元配列の方法は、二重添字から番地を計算するのに、乗算が必要で、CPU によっては時間がかかる。
- ポインター配列の方法は、余分なメモリーを必要とする。
- ポインター配列の方法は、成分へのアクセスに、余分なメモリー・アクセスを必要とする。システムによっては時間がかかる。(少し工夫したコーディングをすれば無駄は減少するが、面倒である。)
- ポインター配列の方法では、行列の行の交換が、次のように実際の成分を移動せずに実現できるので、非常に高速。

```
double *ap;
...
ap = a[i]; a[i] = a[j]; a[j] = ap;
```

B.15.5　サンプル・プログラム

ポインター配列の方法で、行列を確保する関数 new_matrix() を以下に示す。

```
/*
 * test4.c --- GLSC で使うために連続した領域を確保することにした。
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef double *vector;
typedef vector *matrix;

/* m 行 n 列の行列を作る */
matrix new_matrix(int m, int n)
```

```
{
    int i;
    matrix a;
    vector abody;

    if ((a = (matrix)malloc(m * sizeof(vector))) == NULL)
        return NULL;
    if ((abody = (vector)malloc(m * n * sizeof(double))) == NULL)
        return NULL;
    for (i = 0; i < m; i++)
        a[i] = abody + n * i;
    return a;
}

void del_matrix(matrix a)
{
    free(a[0]);
    free(a);
}

int main()
{
    int m,n,i,j;
    matrix a;
    void kuku(matrix, int, int);

    printf("m,n: "); scanf("%d %d", &m, &n);
    a = new_matrix(m,n);
    if (a == NULL) {
        fprintf(stderr, "new_matrix() に失敗\n");
        exit(1);
    }
    kuku(a,m,n);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf("a[%d][%d]=%g ", i, j, a[i][j]);
        printf("\n");
    }
    del_matrix(a);
    return 0;
}

void kuku(matrix a, int m, int n)
{
    int i,j;
    for (i = 0; i < m; i++)
        for (j=0; j < n; j++)
            a[i][j] = (i + 1) * (j + 1);
}
```

B.16 FORTRAN プログラムを C に書き換える

数値計算のプログラミングにおいては、まだまだ FORTRAN の優位性は動かず、参考書や出来合いのプログラムの量は圧倒的に FORTRAN の方が豊富です。それで FORTRAN で書かれたプログラムを C に書き直すことはしばしば行なわれます。

B.16.1 プログラムの構成

FORTRAN プログラムは唯一の主プログラム[15] と、複数 (0 個のこともある) の副プログラム (subroutine, function) からなる。

- FORTRAN の主プログラム[16]は `main()` に

```
program fem02
...
end
↓
main()
{
...
}
```

- FORTRAN の関数は (当然) 関数に

```
real function max(a,b)
real a,b
if (a .gt b) then
  max = a
else
  max = b
endif
end
↓
double max(double a, double b)
{
   /* 普通は return (a > b) ? a : b; かな */
   if (a > b)
      return a;
   else
      return b;
```

[15]行儀の良いプログラマーならば、主プログラムは program 文で開始してあるはずだが、そうでない場合は、いきなり主プログラムが始まる。

[16]お行儀の良いプログラムでは、program プログラム名 と end 文で囲まれた範囲が主プログラムだが、program 文を書かない人も多い。

```
}
```

- FORTRAN のサブルーチンは void 型の関数に

```
call rkf(m,a,sol)
...
subroutine rkf(n,x0,y)
integer n
real x0,y(*)
...
↓
void rkf(int,double,double *);          プロトタイプ宣言
...
rkf(m,a,sol);                           呼び出し
...
void rkf(int n, double x0, double *y)   関数の定義
{
...
```

B.16.2 定数・変数の宣言

- integer は int に

```
integer a,b,c(10)
      ↓
int a,b,c[11];
```

- real, real8, double precision は double に[17]

```
real a,b,c(10)
real*8 x
      ↓
double a,b,c[11],x;
```

- parameter 文による定数の宣言は #define に直す。

```
integer maxn
real x0
parameter (maxn=100, x0=0.0)
↓
#define MAXN 100
#define X0 0.0
```

[17] C 言語では単精度実数型 float が中途半端なので、double を使うことを強く勧めする。

ちなみに

- C では慣習として、名前の中で使う英字は
  - 変数名の場合はすべて小文字
  - 定数名の場合はすべて大文字

  を使う (逆をやったらモグリである)。したがって、定数は maxn とか MaxN とするよりは MAXN とするのが普通。
- #define は C の文というよりも、マクロなので、
  - 最後にセミコロン ";" はつけない。
  - # は必ず行の先頭 (1 桁目) にないとダメ。

- 大きな配列を用いる場合は、動的に確保することも検討する。少なくとも、自動変数でなく、静的変数として宣言しよう。

```
real a(10000)
  ↓
int n;
double *a;
...
n = 今回の実行で本当に必要な量
a = malloc(sizeof(double) * n);
if (a == NULL) エラー処理
...
```

- dimension 文を使ってあった場合、変数宣言にまとめる。

```
int a
dimension a(1000)
     ↓
int a[1001];
```

- read(5,) や read(*,) は scanf() や fgets(,,stdin) & sscanf() に治す。

```
read(*,*) n,a
    ↓
scanf("%d %lf", &n, &a);
あるいは
char buf[1024];
....
fgets(buf, sizeof(buf), stdin);
sscanf(buf, "%d %lf", &n, &a);
```

- write(6,) や write(*,) は printf() に

```
write(*,*) n,a
    ↓
printf("%d %f\n", n, a);
```

- format() は printf() や fprintf() のフォーマット指定文字列の中に

```
      write(*,1000) n,a,b
 1000 format(' ', I5, ' ', 2F14.5)
           ↓
      printf(" %5d %14.5f%14.5f\n", n, a, b);
```

- do 文は for 文に

```
      do 10 i=1,10
        do 20 j=N0,N1,N2
          むにゃむにゃ
 20     continue
 10   continue
           ↓
      for (i = 1; i <= 10; i++)
        for (j = N0; j <= N1; j += N2) {
          むにゃむにゃ
        }
```

- (定数の定義) parameter 文は #define に

```
        integer n
        real a
        parameter (n=100, a=1.0)
          ↓
#define N 100
#define A 1.0
```

- 1 次元配列 x(0:n) は x[n+1] に

```
        integer mm(0:n)
          ↓
        int mm[N+1];
```

- 宣言されていない変数は IJKLMN ルールで型を判定して、きちんと宣言

```
         ↓
        int n,n1,nn,j,k;
        double x0, a;
```

B.16.3 式

- 論理式の値
  - .FALSE. は 0
  - .TRUE. は 0 以外の任意の整数値
- 論理演算子
  - .not. は !
  - .and. は &&
  - .or. は ||
- 比較演算子
  - .eq. は ==
  - .ne. は !=
  - .gt. は >
  - .ge. は >=
  - .lt. は <
  - .le. は <=
- 配列について。
  - 括弧は (,) から [,] に

```
real a(0:NN)
..
x = a(3)
...
↓
double a[NN+1];
..
x = a[3];
```

  - FORTRAN では配列の添字は、任意の整数から任意の整数までと出来る。デフォールトでは 1 から。C では 0 から。

```
real a(N)
...
do i=1,N
  a(i)=0.0
end do
...
x=a(3)
```

```
↓
double a[N];
...
for (i=0; i<N; i++)
  a[i] = 0.0;
...
x = a[2];
```
または
```
double a[N+1];
...
for (i=1; i<=5; i++)
  a[i] = 0.0;
...
x = a[3];
```

– 多次元配列は FORTRAN では , で区切る。C では、[] を繰り返す。

```
real mat(3,3)
x = mat(i,j)
↓
double mat[3][3];
x = mat[i][j];
```

– FORTRAN の多次元配列には整合配列の機能があるが、C にはない。C で同じことをするには、次のいずれかの方法を用いる。

  * 十分大きな 2 次元配列を宣言して、その一部を使う。(「十分」これは関数の可搬性が損なわれる。例えば、次の inv() はライブラリィに登録することは出来ない。)

```
real a(NDIM,NDIM)
...
a(i,j)=むにゃ
...
call inv(NDIM,3,a)
..
subroutine inv(NDIM, N, a)
integer NDIM,N
real a(NDIM,*)
...
↓
double a[NBIG][NBIG];
...
a[i][j] = むにゃ
...
inv(3,a);
```

```
...
void inv(int n, double a[][NBIG])
```

* 一次元配列を宣言して (あるいはポインターで同様のことをする)、添字計算を自分でやる。(これはかなり面倒くさい。マクロを使って処理すべき？)

```
double *a;
...
a = malloc(sizeof(double) * n * n); if (a == NULL) ...
...
a[i*3+j] = むにゃ
...
inv(3,a);
...
void inv(int n, double *a)
```

* ポインター配列 (あるいはポインターのポインター) を用いる。

```
double *a[NDIM];
for (i = 0; i < 3; i++) {
   if ((a[i] = malloc(sizeof(double) * 3)) == NULL) {
   }
}
...
a[i][j] = むにゃ
...
inv(3,a);
...
void inv(int n, double **a)
```

– FORTRAN では、そのプログラム内で使う大きな配列は、主プログラムで宣言するのが普通だが、C では動的にポインターで確保するのが普通。

```
real a(0:NN,M)
...
x = a(0,j)
↓
double a[NN+1][M];
...
x = a[0][j];
```

- FORTRAN の stop 文は C では exit() 関数の呼び出しに書き換える。(exit() 関数の引数には小さな正整数を与えるが、正常終了の場合は 0, 異常終了の場合は 0 以外の値を返すのが慣習である。)

```
 stop
↓
```

```
exit(0);            正常終了の場合
```

あるいは

```
exit(1);            異常終了の場合
```

- FORTRAN の冪乗演算子は適当に

```
L=j**2
n=j**3
m=j**4
x=y**0.5
z=y**(1.0/3.0)
```

↓

```
L=j*j;
n=j*j*j;
tmp = j*j; m=tmp*tmp;
x=sqrt(x);
z=pow(x,1.0/3.0)
```

- FORTRAN の format 文の E, F, G 変換はそれぞれ printf() の書式文字列内では %e, %f, %g に。 I 変換は printf() の書式文字列内では %d に。

- FLOAT() のような型変換関数は何もしないでよい時もあるし、必要な場合はキャスト演算子で。

```
integer n;
real x,r
...
x=float(n)
r=float(1)/float(n)
call mysub(a, float(n))
↓
int n;
double x,r
...
x=n;                        自動的に型変換される。
r=1.0/n;                    1.0 に合わせて n は double に型変換される
mysub(a, (double)n);        もしも mysub() にプロトタイプ宣言があれば
                            キャストは不要である。
```

B.17 NaN って何ですか

B.18 コンパイルとは何ですか

B.19 NEmacs 虎の巻 @(#) Oct 4 1992

B.19.1 NEmacs 入門編

以下 C-x と書いてあるところは、キーボードのコントロール・キー (“Control” とか “CTRL” と書かれているキー) を押しながら、x をタイプすることを表す。

起動の仕方 nemacs *filename*

終了の仕方 C-x C-c[18]
セーブ (文書登録) するかどうか聞いてきたら、y or n で答えます。
n と答えると、本当に終了してよいかどうか確認されるので、yes
または no を入力して改行キーを押します。

カーソル移動 上下左右の矢印キーが使えます。

一字抹消 カーソル位置の文字: C-d
カーソル直前の文字: Delete (BackSpace の場合もある)

トラブル対策 おかしな状態からの脱出: C-g アンドゥ (元に戻す) : C-/

漢字変換 変換機能のオンオフ : C- 変換 : Space 変換中止 (取消): C-g 次候補 : Space 文節を縮める : C-i 前候補 : C-p 文節を伸ばす : C-o

B.19.2 NEmacs 初級編

カーソル移動 矢印キー以外の、 Control-何々、ESC -何々、を覚えよう。

左 : [←] または C-b	上 : [↑] または C-p
右 : [→] または C-f	下 : [↓] または C-n
次ページ : C-v	*文書頭 : Esc <
前ページ : Esc v	*文書末 : Esc >
行頭 : C-a	*現在位置をマーク : C-@ または C-Space
行末 : C-e	*マークまで移動 : C-x C-x

(*印は、実行前の位置がマークされます。)

削除、消去 行末までの消去と、マークまで (リージョン) の削除

一字消去 : C-d
行末まで削除 : C-k
１行削除 : C-k C-k

ｎ行削除： Esc 行数 C-k
マークまで削除： C-w
(削除したものは記憶されます。消去したものは記憶されません。)

移動、*複写ク： C-y　　(最後に記憶した内容を現在位置に挿入。
挿入された文章の先頭にマークがつきます。
記憶し直すまで何度でもヤンクできます。)
マークまで記憶：　　ESC w

検索 検索開始： C-s　　検索完了： Esc
次検索： C-s　　中止 (元の位置へ): C-g

【さらに詳しいことを知りたい人のための参考書】

1. 永峯猛志、Nemacs の手引き、明治大学情報科学センター
   ftp://mjuserv.isc.meiji.ac.jp/pub/meiji/doc/Nemacs-tebiki.tar.Z
2. 、アスキー出版局
3. 矢吹道朗・宮城史朗、はじめて使う GNU Emacs、啓学出版
4. R. ストールマン、GNU Emacs マニュアル、共立出版

付 録C　Fortran と一緒に使う

C.1　UNIX における C と Fortran の併用

従来から、UNIX 上の cc と f77 には互換性があり、ある程度混合したプログラミングが可能であった。この事情は gcc と g77 にも受け継がれている。

C.2　プログラムの書き方

C.2.1　副プログラムの名前について

C で myfunc() という名前の関数は、アセンブリ言語では _myfunc という名前になる (先頭に下線がつく)。

Fortran で mysub という名前のサブルーチンは、アセンブリ言語では _mysub_ という名前になる (先頭と末尾に下線がつく)。

C のプログラムから Fortran の mysub というサブルーチンを呼び出すには、mysub_() と書く必要がある。

C.2.2　引数について

C では値による呼び出し (call by value) が普通だが、Fortran ではアドレスによる呼び出し (ほとんど call by reference) が基本である。だから、C 側から Fortran のサブルーチンを呼び出すには、ポインターを渡せば良い。

C.3　配列

1 次元配列については問題ない。2 次元配列については、、、

C.3.1 プログラム例

test1main.c

```
#include <stdio.h>
main()
{
  int i;
  double x;
  char c;
  i = 1;
  x = 1.23;
  c = 'A';
  mysub_(&i, &x, &c);
  printf("i=%d, x=%f, c=%c\n", i, x, c);
}
```

test1sub.f

```
      subroutine mysub(i,x,c)
      integer i
      real*8 x
      character c
      write(*,*) i
      write(*,*) x
      write(*,*) c
      i = 234
      x = 234.567
      c = 'B'
      end
```

C.4 コンパイルの仕方

通常 cc で作成されるプログラムと、f77 で作成されるプログラムでは、使用するライブラリィに相違がある。

大抵は f77 で使用するライブラリィの方が多い。そこで f77 でコンパイルすると、うまく行くことが多い。

cc あるいは gcc を使ってコンパイル＆リンクする場合には、リンクするライブラリィをこちらで指定してやる必要がある (何もしないと、libc.a をリンクするだけのコンパイラーが多い)。どういうライブラリィが必要か、cc の場合はなかなか分かりにくい。

FreeBSD 標準の f77 は f2c という Fortran → C トランスレーターが元になっているもので、この場合 Fortran 用のライブラリィは `libf2c.a` という名前になっている。だから、cc (gcc) に `-lf2c` というオプションを渡せば良いはず (なお、数値計算をするので -lm はほとんどの場合に必須である)。

なお、g77 (GNU Fortran 77 compiler) は、現時点ではライブラリィ関係は f2c を元にしていて、やはり libf2c.a をリンクしているらしい。

cc と f77

```
oyabun% cc -c test1main.c
oyabun% f77 -c test1sub.f
oyabun% f77 -o prog test1main.o test1sub.o
```

gcc と g77 (1)

```
oyabun% gcc -c test1main.c
oyabun% g77 -c test1sub.f
oyabun% g77 -o prog test1main.o test1sub.o
```

gcc と g77 (2)

```
oyabun% gcc -c test1main.c
oyabun% g77 -c test1sub.f
oyabun% gcc -o prog test1main.o test1sub.o -lf2c
```

C.5 LAPACK

LAPACK のライブラリィ名は liblapack.a で、そこから呼び出す BLAS のライブラリィ名は libblas.a なので、LAPACK のサブルーチンを使用する場合は、`-llapack -lblas` というコンパイラー・オプションが必要になる。

LAPACK

```
oyabun%  g77 -o prog test1main.o test1sub.o -llapack -lblas
```

付 録D　C++ を使ってみよう

D.1　C++ について私が考えること

D.1.1　数学屋にとって面白く有望である

(省略)

D.1.2　変化が速すぎる

C との可能な限りの互換性を確保したまま拡張を続けている。その変化の激しさを見ると、「トランスレーター」作製者が規格に追い付くのがなかなか大変だと思う (Stroustrup はかの有名なバイブルの第 3 版で、GCC と思われる某コンパイラーを新しい規格のサポートが不十分だとしてけなしているようだが、ちょっと酷だと思う ... 実際、Visual C++, Borland C++ など、現在でも完全には規格準拠になっていないのではないだろうか)。また C++ について解説した書籍もある意味では「古く」なってしまったものが多い。さらに既に書かれたプログラムも今では規格外になってしまったものが少なくないようである。

D.1.3　printf() ファミリーの代替物がないぞ！

C++ の信奉者達は、現在のストリーム入出力が大層お気に入りのようで、もう昔ながらの C の標準入出力ライブラリィ、例えば `printf()` ファミリーなどは使うなとおっしゃるのですが... 書式指定一つ取ってもあまり使いやすくないですな。

- これまで %g, %f, %e とだけ覚えれば良かったのに、デフォールト, `fixed`, `scientific` なんて覚えるのは面倒ですな。しかも、一度何かを指定すると、それを解除するまでフラグの変化が残ってしまって、デフォールトの状態に戻すのが面倒である。

GCC のように、`printf()` 互換の `form()` メンバー関数をこっそりと準備するのももっともなことだと思う。

D.2　簡単な入門

D.2.1　printf() の代りに cout と insertion 演算子 <<

少し前まで C++ 版 Hello World は次のように書くと言われていた。

hello_world.C

```
// prog01.C

#include <iostream.h>

int main()
{
  cout << "Hello, world" << endl;
  return 0;
}
```

しかし今では名前空間 (**namespace**) が導入されて、

hello_world-new.C

```
// prog01.C

#include <iostream>
using namespace std; // std::cout, std::endl などが単に cout, endl と書ける

int main()
{
  cout << "Hello, world" << endl;
  return 0;
}
```

と書くものらしい。もっともまだサポート出来ていないコンパイラーがある。

これまた C で有名なコピー・プログラム

getchar_puthcar.c

```
#include <stdio.h>
int main()
{
  int c;
  while ((c = getchar()) != EOF)
    putchar(c);
}
```

の C++ 版は

prog02.C

```
#include <iostream.h>

int main()
{
  char c;
  while (cin.get(c))
    cout << c; // あるいは cout.put(c); か？
  return 0;
}
```

となるのかな？ (あまり自信がないので、何冊か本を探してみよう[1]。)

[1] このあたりに、Stroustrup の「プログラミング言語 C++」が K&R の「プログラミング言語 C」のレベルに達していないと感じさせる原因があると思う。1000 ページもあって、色々書いているくせに、この種の基本的なことが分かりにくい...

D.2.2 cout.width() とインサーター setw()

cout.width(n) とすると、少なくとも n 桁の幅を取る。一つ実際に出力したら元に戻ってしまうので、毎回指定する。

prog03.C

```
// prog03.C

#include <iostream.h>

int main()
{
  int i, j;

  for (i = 1; i <= 9; i++) {
    for (j = 1; j <= 9; j++) {
      // printf("%3d", i * j); と同等
      cout.width(3);
      cout << i * j;
    }
    cout << endl;
  }
  return 0;
}
```

D.2.3 左詰め, 右詰め

left-right.C

```
#include <iostream.h>
#include <iomanip.h>

int main()
{
  char s[] = "Katsurada";

  cout.fill('#');
  cout << "1. default" << endl;
  cout << setw(20) << s << endl;

  cout << "2. left" << endl;
  cout.width(20); cout.setf(ios::left, ios::adjustfield);
  cout << s << endl;

  cout << "3. internal" << endl;
  cout.width(20); cout.setf(ios::internal, ios::adjustfield);
  cout << s << endl;

  cout << "4. right" << endl;
  cout.width(20); cout.setf(ios::right, ios::adjustfield);
  cout << s << endl;

  cout << "5. left" << endl;
  cout << setw(20) << setiosflags(ios::left) << s << endl;
  cout << resetiosflags(ios::left);

  cout << "6. internal" << endl;
  cout << setw(20) << setiosflags(ios::internal) << s << endl;
  cout << resetiosflags(ios::internal);

  cout << "7. right" << endl;
  cout << setw(20) << setiosflags(ios::right) << s << endl;

  cout << endl << "internal は文字列に対しては意味がないみたい。" << endl;

  return 0;
}
```

left-right.out

```
mathpc00% ./left-right
1. default
###########Katsurada
2. left
Katsurada###########
3. internal
###########Katsurada
4. right
###########Katsurada
5. left
Katsurada###########
6. internal
###########Katsurada
7. right
###########Katsurada

internal は文字列に対しては意味がないみたい。
mathpc00%
```

D.2.4 フラッシュ

`fflush(stdout);` に相当するのは `cout << flush;` か。

D.2.5 8 進数, 10 進数, 16 進数

`printf()` では、整数データに対して、%o, %d, %x という書式指定で、それぞれ 8 進数, 10 進数, 16 進数を表示できた。C++ ではどうするか？

kakezan.C

```
// kakezan.C -- 掛け算 FF

#include <iostream.h>

int main()
{
  int i, j;

  // 掛け算 FF
  cout.setf(ios::hex, ios::basefield);
  for (i = 1; i <= 15; i++) {
    for (j = 1; j <= 15; j++) {
      cout.width(3);
      cout << i * j;
    }
    cout << endl;
  }
  // 掛け算 77
  cout.setf(ios::oct, ios::basefield);
  for (i = 1; i <= 7; i++) {
    for (j = 1; j <= 7; j++) {
      cout.width(3);
      cout << i * j;
    }
    cout << endl;
  }
  // 掛け算 99
  cout.setf(ios::dec, ios::basefield);
  for (i = 1; i <= 9; i++) {
    for (j = 1; j <= 9; j++) {
      cout.width(3);
      cout << i * j;
    }
    cout << endl;
  }
  return 0;
}
```

kakezan-new.C

```
// kakezan.C -- 掛け算 FF

#include <iostream.h>
#include <iomanip.h>

int main()
{
  int i, j;

  // 掛け算 FF
  cout << hex;
  for (i = 1; i <= 15; i++) {
    for (j = 1; j <= 15; j++)
      cout << setw(3) << i * j;
    cout << endl;
  }
  // 掛け算 77
  cout << oct;
  for (i = 1; i <= 7; i++) {
    for (j = 1; j <= 7; j++)
      cout << setw(3) << i * j;
    cout << endl;
  }
  // 掛け算 99
  cout << dec;
  for (i = 1; i <= 9; i++) {
    for (j = 1; j <= 9; j++)
      cout << setw(3) << i * j;
    cout << endl;
  }
  return 0;
}
```

kakezan-new2.C

```
// kakezan-new2.C -- 掛け算 FF

#include <iostream.h>
#include <iomanip.h>

int main()
{
  int i, j;

  // 掛け算 FF
  cout << setbase(16);
  for (i = 1; i <= 15; i++) {
    for (j = 1; j <= 15; j++)
      cout << setw(3) << i * j;
    cout << endl;
  }
  // 掛け算 77
  cout << setbase(8);
  for (i = 1; i <= 7; i++) {
    for (j = 1; j <= 7; j++)
      cout << setw(3) << i * j;
    cout << endl;
  }
  // 掛け算 99
  cout << setbase(10);
  for (i = 1; i <= 9; i++) {
    for (j = 1; j <= 9; j++)
      cout << setw(3) << i * j;
    cout << endl;
  }
  return 0;
}
```

D.2.6 浮動小数点数の書式指定 (1)

- フォーマット・フラグ showpos の設定によって、非負値の表示の先頭に+ をつけるかどうかを決める。

test-showpos.C

```
#include <iostream.h>
#include <iomanip.h>

int main()
{
  for (int i = 0; i <= 10; i++) {
    cout << setw(5) << - 1.0 + 0.2 * i;
  }
  cout << endl;
  cout.setf(ios::showpos);
  for (int i = 0; i <= 10; i++) {
    cout << setw(5) << - 1.0 + 0.2 * i;
  }
  cout << endl;
  return 0;
}
```

test-showpos.out

```
mathpc00% ./test-showpos
   -1 -0.8 -0.6 -0.4 -0.2    0  0.2  0.4  0.6  0.8    1
   -1 -0.8 -0.6 -0.4 -0.2   +0 +0.2 +0.4 +0.6 +0.8   +1
mathpc00%
```

- フォーマット・フラグ showpoint の設定によって、どうかを決める。

test-showpoint.C

```
#include <iostream.h>
#include <iomanip.h>

int main()
{
  int i;
  for (i = 0; i <= 10; i++) {
    cout << setw(5) << - 1.0 + 0.2 * i << endl;
  }
  cout.setf(ios::showpoint);
  for (i = 0; i <= 10; i++) {
    cout << setw(5) << - 1.0 + 0.2 * i << endl;
  }
  return 0;
}
```

test-showpoint.out

```
mathpc00% ./test-showpoint
   -1
 -0.8
 -0.6
 -0.4
 -0.2
    0
  0.2
  0.4
  0.6
  0.8
    1
-1.00000
-0.800000
-0.600000
-0.400000
-0.200000
0.00000
0.200000
0.400000
0.600000
0.800000
1.00000
mathpc00%
```

- フォーマット・フラグ fixed の設定によって、固定小数点形式で表示するかどうかを決める。floatfield は fixed と scientific の論理和になっている。

test-fixed.C

```
#include <iostream.h>
#include <iomanip.h>

void print()
{
  int i;
  double x1, x2;
  x1 = x2 = 1;
  for (i = 1; i <= 8; i++) {
    cout << x1 << " " << x2 << endl;
    x1 *= 8; x2 /= 8;
  }
  cout << endl;
}

int main()
{
  cout << "default" << endl;
  print();

  cout << "cout.setf(ios::fixed, ios::floatfield);" << endl;
  cout.setf(ios::fixed, ios::floatfield);
  print();

  cout << "resetiosflags(ios::fixed);" << endl;
  cout << resetiosflags(ios::fixed);
  print();

  // cout << fixed; とはできないのかな？
  cout << "setiosflags(ios::fixed);" << endl;
  cout << setiosflags(ios::fixed);
  print();

  // これでもリセットされるか？
  cout << "resetiosflags(ios::floatfield);" << endl;
  cout << resetiosflags(ios::floatfield);
  print();

  return 0;
}
```

test-fixed.out

```
mathpc00% ./test-fixed
default
1 1
8 0.125
64 0.015625
512 0.00195312
4096 0.000244141
32768 3.05176e-05
262144 3.8147e-06
2.09715e+06 4.76837e-07

cout.setf(ios::fixed, ios::floatfield);
1.000000 1.000000
8.000000 0.125000
64.000000 0.015625
512.000000 0.001953
4096.000000 0.000244
32768.000000 0.000031
262144.000000 0.000004
2097152.000000 0.000000

resetiosflags(ios::fixed);
1 1
8 0.125
64 0.015625
512 0.00195312
4096 0.000244141
32768 3.05176e-05
262144 3.8147e-06
2.09715e+06 4.76837e-07

setiosflags(ios::fixed);
1.000000 1.000000
8.000000 0.125000
64.000000 0.015625
512.000000 0.001953
4096.000000 0.000244
32768.000000 0.000031
262144.000000 0.000004
2097152.000000 0.000000

resetiosflags(ios::floatfield);
1 1
8 0.125
64 0.015625
512 0.00195312
4096 0.000244141
32768 3.05176e-05
262144 3.8147e-06
2.09715e+06 4.76837e-07

mathpc00%
```

- フォーマット・フラグ scientific の設定によって、固定小数点形式で表示するかどうかを決める。

test-scientific.C

```
#include <iostream.h>
#include <iomanip.h>

void print()
{
  int i;
  double x1, x2;
  x1 = x2 = 1;
  for (i = 1; i <= 8; i++) {
    cout << x1 << " " << x2 << endl;
    x1 *= 8; x2 /= 8;
  }
  cout << endl;
}

int main()
{
  cout << "default" << endl;
  print();

  cout << "cout.setf(ios::scientific, ios::floatfield);" << endl;
  cout.setf(ios::scientific, ios::floatfield);
  print();

  cout << "resetiosflags(ios::scientific);" << endl;
  cout << resetiosflags(ios::scientific);
  print();

  // #include <ios> とすると cout << scientific; とするだけで良いのか？
  cout << "setiosflags(ios::scientific);" << endl;
  cout << setiosflags(ios::scientific);
  print();

  // これでもリセットされるか？
  cout << "resetiosflags(ios::floatfield);" << endl;
  cout << resetiosflags(ios::floatfield);
  print();

  return 0;
}
```

test-scientific.out

```
mathpc00% ./test-scientific
default
1 1
8 0.125
64 0.015625
512 0.00195312
4096 0.000244141
32768 3.05176e-05
262144 3.8147e-06
2.09715e+06 4.76837e-07

cout.setf(ios::scientific, ios::floatfield);
1.000000e+00 1.000000e+00
8.000000e+00 1.250000e-01
6.400000e+01 1.562500e-02
5.120000e+02 1.953125e-03
4.096000e+03 2.441406e-04
3.276800e+04 3.051758e-05
2.621440e+05 3.814697e-06
2.097152e+06 4.768372e-07

resetiosflags(ios::scientific);
1 1
8 0.125
64 0.015625
512 0.00195312
4096 0.000244141
32768 3.05176e-05
262144 3.8147e-06
2.09715e+06 4.76837e-07

setiosflags(ios::scientific);
1.000000e+00 1.000000e+00
8.000000e+00 1.250000e-01
6.400000e+01 1.562500e-02
5.120000e+02 1.953125e-03
4.096000e+03 2.441406e-04
3.276800e+04 3.051758e-05
2.621440e+05 3.814697e-06
2.097152e+06 4.768372e-07

resetiosflags(ios::floatfield);
1 1
8 0.125
64 0.015625
512 0.00195312
4096 0.000244141
32768 3.05176e-05
262144 3.8147e-06
2.09715e+06 4.76837e-07

mathpc00%
```

- 精度 (小数点以下の桁数) を指定するには、precision() メンバー関数または setprecision マニュピュレーターを用いる。

test-precision.C

```
#include <iostream.h>
#include <iomanip.h>
#include <cmath>

int main()
{
  int i;
  double pi = atan(1.0) * 4;

  cout << "デフォールトでは精度" << cout.precision() << "桁である" << endl;
  for (i = 1; i <= 16; i++) {
    cout.precision(i);
    cout << setw(2) << i << ":" << pi << endl;
  }
  for (i = 1; i <= 16; i++)
    cout << setw(2) << i << ":" << setprecision(i) << pi << endl;

  return 0;
}
```

test-precision.out

```
mathpc00% ./test-precision
デフォールトでは精度 6 桁である
 1:3
 2:3.1
 3:3.14
 4:3.142
 5:3.1416
 6:3.14159
 7:3.141593
 8:3.1415927
 9:3.14159265
10:3.141592654
11:3.1415926536
12:3.14159265359
13:3.14159265359
14:3.1415926535898
15:3.14159265358979
16:3.141592653589793
 1:3
 2:3.1
 3:3.14
 4:3.142
 5:3.1416
 6:3.14159
 7:3.141593
 8:3.1415927
 9:3.14159265
10:3.141592654
11:3.1415926536
12:3.14159265359
13:3.14159265359
14:3.1415926535898
15:3.14159265358979
16:3.141592653589793
mathpc00%
```

testformat1.C

```
#include <iostream.h>

void print()
{
  int i;
  double x1, x2;
  x1 = x2 = 1;
  for (i = 1; i <= 30; i++) {
    cout << x1 << " " << x2 << endl;
    x1 *= 2; x2 /= 2;
  }
}
int main()
{
  cout << "何もしないと C の %g 状態" << endl;
  print();
  cout << "cout.setf(ios::showpoint);" << endl;
  cout.setf(ios::showpoint);
  print();
  cout << "cout.setf(ios::fixes, ios::floatfield); とすると %f 相当" << endl;
  cout << "cout.setf(ios::fixed, ios::floatfield);" << endl;
  cout.setf(ios::fixed, ios::floatfield);
  print();
  cout << "cout.unsetf(ios::showpoint); としても変化なし？" << endl;
  cout << "cout.unsetf(ios::showpoint);" << endl;
  cout.unsetf(ios::showpoint);
  print();
  cout << "cout.setf(ios::scientific, ios::floatfield); とすると %e 相当" << endl;
  cout << "cout.setf(ios::scientific, ios::floatfield);" << endl;
  cout.setf(ios::scientific, ios::floatfield);
  print();
  cout << "cout.precision(16); ... この場合は小数点以下の桁数" << endl;
  cout.precision(16);
  cout << "  12345678901234567890" << endl;
  print();
  cout << "cout.setf(ios::fixed, ios::floatfield);" << endl;
  cout.setf(ios::fixed, ios::floatfield);
  cout << "  12345678901234567890" << endl;
  print();
}
```

testformat1.out

```
mathpc00% g++ -o testformat1.C
mathpc00% ./testformat1
何もしないと C の %g 状態
1 1
2 0.5
4 0.25
8 0.125
16 0.0625
32 0.03125
64 0.015625
128 0.0078125
256 0.00390625
512 0.00195312
1024 0.000976562
2048 0.000488281
4096 0.000244141
8192 0.00012207
16384 6.10352e-05
32768 3.05176e-05
65536 1.52588e-05
131072 7.62939e-06
262144 3.8147e-06
524288 1.90735e-06
1.04858e+06 9.53674e-07
2.09715e+06 4.76837e-07
4.1943e+06 2.38419e-07
8.38861e+06 1.19209e-07
1.67772e+07 5.96046e-08
3.35544e+07 2.98023e-08
6.71089e+07 1.49012e-08
1.34218e+08 7.45058e-09
2.68435e+08 3.72529e-09
5.36871e+08 1.86265e-09
cout.setf(ios::showpoint);
1.00000 1.00000
2.00000 0.500000
4.00000 0.250000
8.00000 0.125000
16.0000 0.0625000
32.0000 0.0312500
64.0000 0.0156250
128.000 0.00781250
256.000 0.00390625
512.000 0.00195312
1024.00 0.000976562
2048.00 0.000488281
4096.00 0.000244141
8192.00 0.000122070
16384.0 6.10352e-05
32768.0 3.05176e-05
65536.0 1.52588e-05
131072. 7.62939e-06
262144. 3.81470e-06
524288. 1.90735e-06
1.04858e+06 9.53674e-07
2.09715e+06 4.76837e-07
4.19430e+06 2.38419e-07
8.38861e+06 1.19209e-07
1.67772e+07 5.96046e-08
3.35544e+07 2.98023e-08
6.71089e+07 1.49012e-08
1.34218e+08 7.45058e-09
2.68435e+08 3.72529e-09
```

D.2.7　浮動小数点数の書式指定 (2) 安直な form()

これは GCC 拡張であって、規格外なのかも知れないけれど、printf() 互換の form() メンバー関数がある。

testformat2.C

```
#include <iostream.h>

void print()
{
  int i;
  double x1, x2;
  x1 = x2 = 1;
  for (i = 1; i <= 30; i++) {
    cout.form("%f %e %g", x1, x1, x1);
    cout.form("%f %e %g", x2, x2, x2);
    cout << endl;
    x1 *= 2; x2 /= 2;
  }
}

int main()
{
  print();
}
```

D.2.8　文字列

string クラスについて一通り。古い C++ のテキストでは文字列クラスを作ってみせるのが定番という雰囲気もあったが (それどころか C 風の文字列の説明を一所懸命している本もまだ珍しくはない)、標準 C++ ライブラリィに含まれるようになったので、もうその利用方法を学ぶ時期でしょう。

string1.C

```
#ifdef __FreeBSD__
#include <iostream.h>
#else
#include <iostream>
#endif

#include <string>
using namespace std;

void add_newline(string &s)
{
  s += '\n';
}

int main()
{
  // string に文字列リテラルを代入できる
  string s1 = "Hello";
  string s2 = "world";

  // string は + 演算子で、string，文字列リテラル，文字を結合できる
  string s3 = s1 + ", " + s2 + '!';

  add_newline(s3);
  cout << s3;

  // string 同士、string と文字列リテラルは == で比較できる
  if (s3 == "Hello, world!\n")
    cout << "Equal" << endl;

  // C 形式の文字列は c_str() メンバ関数で得られる。
  cout.form("%s\n", s1.c_str());

  // 一行丸々入力するには getline()
  string line_input;
  cout << "input a line" << endl;
  getline(cin, line_input);
  cout << "入力された行は" << line_input << endl;
}
```

string1.out

```
mathpc00% ./test-string1
Hello, world!
Equal
Hello
input a line
To be to be, ten made tobe!
入力された行は To be to be, ten made tobe!
mathpc00%
```

test-string1.C

```
#include <iostream.h>
#include <string>

int main()
{
  string s1 = "aa";
  string s2("bb");
  string s3;

  s3 = s1 + s2;
  cout << s1 << endl;
  cout << s2 << endl;
  cout << s3 << endl;
  for (int i = 0; i < 5; i++)
    s3 += s3;
  cout << s3 << endl;
  for (size_t i = 0; i < s3.length(); i++)
    cout << s3[i];
  cout << endl;

  // 部分文字列について
  s3 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << '"' << s3 << '"' << "という文字列について" << endl;
  string first10(s3, 0, 9);
  string cut_first10(s3, 10);
  cout << "最初の 10 文字" << first10 << endl;
  cout << "最初の 10 文字をカット" << cut_first10 << endl;
  return 0;
}
```

D.2.9 sprintf() の代りに

数値を文字列に変換するには、C では sprintf() を用いるのが便利だが、C++ では **ostrstream** クラスを利用する。

test-strstream1.C --- これは反則？

```
#include <iostream.h>
#include <strstream.h>

int main()
{
  int N;
  double x, err;
  ostrstream ost;
  N = 123; x = 1.234; err = 1.2e-34;
  ost.form("N=%2d,x=%f,err=%e", N, x, err);
  cout << ost.str() << endl;
}
```

test-strstream2.C

```
// test-strstream1.C

#include <iostream.h>
#include <iomanip.h>
#include <strstream.h>

int main()
{
  int N;
  double x, err;
  ostrstream ost;
  N = 123; x = 1.234; err = 1.2e-34;
  ost << "N=" << setw(2) << N << ",x=" << setiosflags(ios::fixed) << x
      << ",err=" << setiosflags(ios::scientific) << err;
  cout << ost.str() << endl;
}
```

test-strstream.out

```
mathpc00% ./test-strstream1
N=123,x=1.234000,err=1.200000e-34
mathpc00% ./test-strstream2
N=123,x=1.234000,err=1.2e-34
mathpc00%
```

D.2.10 sscanf() の代りに

test-sscanf.C

```
// 古い方法を使うと

#include <iostream.h>
#include <string>
#include <cstdio>

int main()
{
  double x;
  int n;
  string s;
  s = "123.4 56";
  sscanf(s.c_str(), "%lf%d", &x, &n);
  cout << "x=" << x << ", n=" << n << endl;
  return 0;
}
```

test-strstream3.C

```
#include <iostream.h>
#include <string>
#include <strstream.h>

int main()
{
  double x;
  int n;
  string s;
  s = "123.4 56";
  istrstream is(s.c_str());
  is >> x >> n;
  cout << "x=" << x << ", n=" << n << endl;
  return 0;
}
```

D.3 vector

vector1.C

```
#include <iostream.h>
#include <vector.h>

int main()
{
  int i, n;
  vector<double> a;

  cout << "input integer: ";
  cin >> n;

  vector<double> b(n);
  for (i = 0; i < n; i++)
    b[i] = i;

  a = b;
  for (i = 0; i < a.size(); i++)
    cout << a[i] << ' ';
  cout << endl;

#if 0
  n = b.size();
  b.resize(n * 2);
  for (i = n; i < b.size(); i++)
    b[i] = b[i - n];
  for (i = 0; i < b.size(); i++)
    cout << b[i] << ' ';
  cout << endl;
#endif
}
```

D.4 ファイル入出力

C の **FILE**, **fopen()**, **fclose()**, **fprintf()** などを使った入力はどうするのか？

fileio.C

```
#include <iostream.h>
#include <fstream.h>

int main()
{
  char c;
  char fname[] = "fileio.C";

  ifstream input(fname);
  if (input.bad()) {
    cerr << fname << "をオープンできませんでした" << endl;
    abort();
  }
  while (input.get(c))
    cout << c;
  return 0;
}
```

fileio1.C

```
#include <iostream.h>
#include <fstream.h>

int main()
{
  char c;
  char fname[] = "fileio1.C";
  ifstream input(fname);
  if (input.bad()) {
    cerr << fname << "をオープンできませんでした" << endl;
    abort();
  }
  while (input.get(c)) {
    cout << c;
  }
  return 0;
}
```

fileio2.C

```
#include <iostream.h>
#include <fstream.h>

int main()
{
  char c;
  char fname1[] = "fileio2.C";
  char fname2[] = "fileio2.C.backup";
  ifstream input(fname1);
  if (input.bad()) {
    cerr << fname1 << "をオープンできませんでした" << endl;
    abort();
  }
  ofstream output(fname2);
  if (output.bad()) {
    cerr << fname2 << "をオープンできませんでした" << endl;
    abort();
  }
  while (input.get(c)) {
    cout << c;
    output << c;
  }
  return 0;
}
```

fileio3.C

```
#include <iostream.h>
#include <fstream.h>

int main()
{
  char c;
  char fname1[] = "fileio3.C";
  char fname2[] = "fileio3.C.backup";
  ifstream input(fname1);
  if (input.bad()) {
    cerr << fname1 << "をオープンできませんでした" << endl;
    abort();
  }
  ofstream output(fname2, ios::noreplace);
  if (output.bad()) {
    cerr << fname2 << "をオープンできませんでした。" << endl;
    cerr << "(既にファイルが存在するのかもしれません。" << endl;
    abort();
  }
  while (input.get(c)) {
    cout << c;
    output << c;
  }
  return 0;
}
```

fileio4.C

```
#include <iostream.h>
#include <fstream.h>

int main()
{
  char c;
  char fname1[] = "fileio4.C";
  char fname2[] = "fileio4.C.backup";
  ifstream input(fname1);
  if (input.bad()) {
    cerr << fname1 << "をオープンできませんでした" << endl;
    abort();
  }
  //  ofstream output(fname2, ios::ate);
  ofstream output(fname2, ios::ate | ios::out);
  if (output.bad()) {
    cerr << fname2 << "をオープンできませんでした。" << endl;
    abort();
  }
  while (input.get(c)) {
    cout << c;
    output << c;
  }
  return 0;
}
```

D.5 数学関数

mathtest.C

```
#ifdef __FreeBSD__
#include <iostream.h>
#else
#include <iostream>
using namespace std;
#endif

#include <cerrno>
#include <cmath>

int main()
{
  double x;
  cin >> x;
  x = abs(x);
  x = ceil(x);
  x = floor(x);
  x = sqrt(x);
  x = pow(x, x);
  x = pow(x, 3);
  x = cos(x);
  x = sin(x);
  x = tan(x);
  x = acos(x);
  x = asin(x);
  x = atan(x);
  x = atan2(x, x);
  x = sinh(x);
  x = cosh(x);
  x = tanh(x);
  x = exp(x);
  x = log(x);
  x = log10(x);
  cout << x << endl;
  // 次の文で FreeBSD 3.4 では Floating exception 発生!
  x = sqrt(-1);
  if (errno == EDOM)
    cerr << "sqrt()" << endl;
  x = pow(1e+300, 2);
  if (errno == ERANGE)
    cerr << "pow()" << endl;
}
```

D.6 複素数の扱い

(準備中)

D.7 new と delete ... 動的メモリー確保

C++ では、動的にメモリーを確保するには、new 演算子を用いる。メモリーの解放には delete 演算子を用いる。

```
double *u;
...
u = new double [N+1];
if (u == NULL) {
  /* エラー処理 */
  cerr << "u[] のメモリー確保に失敗しました。" << endl;
  exit(1);
}

(ここから後は double u[N+1]; のように配列として定義したのと殆んど同様に扱える。)

...
/* メモリーの解放 */
delete [] u;
```

なお、C++ の場合は実行文の後に変数定義をすることも出来るので、

```
  ...
  cout << "区間の分割数 N: "; cin >> N;
  double u[N+1], newu[N+1];
  ...
```

のようにプログラムを書くことも可能である (と思っていたのだが、もしかするとこれ[2]は GCC 拡張なのか？使わない方が無難かも知れない。)。

D.8 C で書かれたライブラリィとのリンク

C で書かれた関数とリンクするには、関数プロトタイプ宣言を extern "C" { }; の中に書く必要があることに注意する。GLSC の場合、プロトタイプ宣言は glsc.h に入っているから、

```
extern "C" {
#include <glsc.h>
}
```

とすれば良い。

D.9 書式付き出力

数表のような形式でデータを出力するのに、C では printf() で様々な書式指定をする。もちろん C++ は C のスーパーセットなので、printf() を使うことも可能だが、cout ストリームと併用するのは気持ちが悪い。以下にどうすれば良いか説明するが、あまりすっきりしない。多くの人が C++ でも printf() を使い続けているようだが、この辺に理由があるのかもしれない。

[2]配列のサイズに変数を使えること。

- 実は cout はメンバー関数として form() というのを持っていて、これは printf() 相当のことが出来る。

```
  cout.form("N=%d, t=%g, error=%e\n", N, t, error);
```

- 直後の表示について、表示幅を指定するには cout.width(整数); とする。

```
  printf("N=%2d, m=%3d\n", N, m);
```

相当のことを実現するには、

```
  cout << "N=";
  cout.width(2);
  cout << N << ", m=";
  cout.width(3);
  cout << m << endl;
```

とするわけである (うぇっ)。あるいは

```
#include <iomanip.h>
...
  cout << setw(2) << N << ", m=" << setw(3) << m << endl;
```

とする (これならまあまあ[3])。

- 浮動小数点数の表示については、
  - cout はデフォールトでは C の %g 相当の動作をする。
  - 固定小数点数表示 (? — %f 相当の動作) をさせたかったら、

```
   cout.setf(ios::fixed, ios::floatfield);
```

あるいは

```
#include <iomanip.h>
...
   cout << setiosflags(fixed)
```

とする。元に戻すには

```
   cout.unsetf(ios::fixed, ios::floatfield);
```

あるいは

```
#include <iomanip.h>
...
  cout << resetiosflags(fixed)
```

とする。

[3]とは言っても、あまりきれいでない、と思う。そもそも、この方法が用意されたのは比較的最近である。

– 指数形式表示 (%e 相当の動作) をさせたかったら、

```
   cout.setf(ios::scientific, ios::floatfield);
```

あるいは

```
#include <iomanip.h>
...
  cout << setiosflags(scientific)
```

とする。もちろん戻すのは

```
   cout.unsetf(ios::scientific, ios::floatfield);
```

あるいは

```
#include <iomanip.h>
...
  cout << resetiosflags(scientific)
```

とする。

– 表示桁数を指定するには

```
   cout.precision(整数式);
```

あるいは

```
#include <iomanip.h>
....
   cout << setprecision(整数式)
```

とする (小数点以下の桁数の指定になる)。

format-sample.C

```
#include <iostream.h>
#include <iomanip.h>

int main()
{
  int N;
  double x, y, z;
  N = 8;
  x = 1.234567890123;
  y = 123.4567890123;
  z = 123.4567890123;
  cout.form("N=%2d, x=%g, y=%22.15f, z=%e\n", N, x, y, z);
  cout << "N=" << setw(2) << N << ", x=" << x << ", y="
       << setiosflags(ios::fixed) << setw(22) << setprecision(15) << y
       << resetiosflags(ios::fixed) << setprecision(6)
       << ", z=" << setiosflags(ios::scientific) << z
       << endl;
  return 0;
}
```

実行結果

```
yurichan% program-cpp/format-sample
N= 8, x=1.23457, y=   123.456789012300007, z=1.234568e+02
N= 8, x=1.23457, y=   123.456789012300007, z=1.234568e+02
yurichan%
```

D.10 クラス定義についての注意

D.10.1 デフォールト・コンストラクター

default-constructor1.C

```
class Complex {
private:
  double re, im;
public:
  Complex(double r, double i): re(r), im(i) { }
};

int main()
{
  Complex a(1.0, 2.0);
  Complex I(0, 1);
}
```

default-constructor2-error.C — コンパイルできない！

```
class Complex {
private:
  double re, im;
public:
  Complex(double r, double i): re(r), im(i) { }
};

int main()
{
  Complex a;
}
```

Complex a; とすると引数のないコンストラクターを呼び出すことになる。それが定義できていないのでエラーになるわけである。例えば次のように修正しなければならない。

default-constructor2.C

```
class Complex {
private:
  double re, im;
public:
  Complex(double r, double i): re(r), im(i) { }
  Complex(): re(0.0), im(0.0) { }
};

int main()
{
  Complex a;
}
```

D.10.2 コピー・コンストラクター

クラス T のオブジェクト x を宣言すると同時に、既に生成されているオブジェクト y を用いて初期化するには、T x(y); のように宣言する。この場合、クラス T のコピー・コンストラクターが呼ばれる。それは T(const T&) という宣言をして作られる。ここで const は絶対必要である。

copy-constructor.C

```
 1  #include <iostream.h>
 2
 3  class Vector {
 4  private:
 5    int dim;
 6    double *body;
 7  public:
 8    Vector(int n) {
 9      cout << "constructor" << endl; dim = n; body = new double [dim];
10    }
11    Vector() { cout << "default constructor" << endl; dim = 0; body = 0; }
12    ~Vector() { cout << "destructor" << endl; delete [] body; }
13    int dimension() { return dim; }
14    double& element(int i) const { return body[i]; }
15    // コピーコンストラクター
16    Vector(const Vector&);
17  };
18
19  Vector::Vector(const Vector& x)
20  {
21    cout << "copy constructor" << endl;
22    dim = x.dim;
23    body = new double [dim];
24    for (int i = 0; i < dim; i++)
25      body[i] = x.element(i);
26  }
27
28  int main()
29  {
30    const int n = 3;
31    int i;
32    Vector c;
33    Vector a(n);
34    for (i = 0; i < n; i++)
35      a.element(i) = i;
36    for (i = 0; i < n; i++)
37      cout << a.element(i) << endl;
38    Vector b(a);
39    for (i = 0; i < n; i++)
40      cout << b.element(i) << endl;
41  }
```

D.10.3 代入演算子

コピー・コンストラクターが必要な場合、代入演算子も自前で用意する必要がある可能性が高い。

T& operator=(const T&) という宣言になる。ここで const は絶対必要である。

substitution.C

```
 1  #include <iostream.h>
 2
 3  class Vector {
 4  private:
 5    int dim;
 6    double *body;
 7  public:
 8    Vector(int n) {
 9      dim = n; body = new double [dim];
10    }
11    Vector() { dim = 0; body = 0; }
12    ~Vector() { delete [] body; }
13    int dimension() const { return dim; }
14    double& element(int i) const { return body[i]; }
15    // コピーコンストラクター
16    Vector(const Vector&);
17    // 代入演算子
18    Vector& operator=(const Vector &);
19  };
20
21  // コピーコンストラクター
22  Vector::Vector(const Vector& x)
23  {
24    dim = x.dim;
25    body = new double [dim];
26    for (int i = 0; i < dim; i++)
27      body[i] = x.element(i);
28  }
29
30  // 代入演算子
31  Vector& Vector::operator=(const Vector &x)
32  {
33    // 自己代入のチェック
34    if (&x != this) {
35      delete [] body;
36      dim = x.dimension();
37      body = new double [dim];
38      for (int i = 0; i < dim; i++)
39        body[i] = x.element(i);
40    }
41    // 次は代入演算子の定跡
42    return *this;
43  }
44
45  int main()
46  {
47    int i;
48    const int n = 3;
49    Vector a(n);
50    for (i = 0; i < n; i++) a.element(i) = i;
51    Vector b, c(n);
52    b = a;
53    c = a;
54    for (i = 0; i < n; i++) cout << a.element(i) << endl;
55    for (i = 0; i < n; i++) cout << b.element(i) << endl;
56    for (i = 0; i < n; i++) cout << c.element(i) << endl;
57  }
```

D.11 valarray

valarray1.C

```
// mytest.C

#include <iostream>
#include <valarray>
#include <slice>

using namespace std;

#include "Slice_iter.h"
#include "Matrix.h"

void print_valarray(valarray<double> &a)
{
  int i;
  cout << "size=" << a.size() << endl;
  for (i = 0; i < a.size(); i++)
    cout << a[i] << " ";
  cout << endl;
}

double sup_norm(valarray<double> &a)
{
  // valarray<double> t = a.apply(abs);
  valarray<double> t = abs(a);
  return t.sum();
}

double naiseki(valarray<double> &a, valarray<double> &b)
{
  valarray<double> ab = a * b;
  return ab.sum();
}

int main()
{
  int i, N;
  valarray<double> y;
  cin >> N;
  valarray<double> x(1.0, N), z(N);
  y.resize(N);
  for (i = 0; i < N; i++)
    y[i] = i;

  cout << "vector x" << endl; print_valarray(x);
  cout << "vector y" << endl; print_valarray(y);
  z = x + y;
  cout << "vector z:=x+y" << endl; print_valarray(z);

  x *= 0.2;
  cout << "vector x:=0.2*x" << endl; print_valarray(x);

  cout << "sup norm of x=" << sup_norm(x) << endl;

  x = 2;
  y = 3;
  cout << "vector x" << endl; print_valarray(x);
  cout << "vector y" << endl; print_valarray(y);

  cout << "naiseki of x, y=" << naiseki(x, y) << endl;
```

Slice_iter.h

```
template<class T> class Slice_iter {
  valarray<T> *v;
  slice s;
  size_t curr;

  T &ref(size_t i) const { return (*v)[s.start()+i*s.stride()]; }
public:
  Slice_iter(valarray<T> *vv, slice ss) : v(vv), s(ss), curr(0) { }
  Slice_iter end()
  {
    Slice_iter t = *this;
    t.curr = s.size();
    return t;
  }
  Slice_iter& operator++() { curr++; return *this; }
  Slice_iter  operator++(int) { Slice_iter t = *this; curr++; return *t; }
  T& operator[](size_t i) { return ref(curr=i); }
  T& operator()(size_t i) { return ref(curr=i); }
  T& operator*() { return ref(curr); }
  // 関係演算子 ==, !=, < を定義
  template<class T> bool operator==
  (const Slice_iter<T> &p, const Slice_iter<T> &q)
  {
    return p.curr==q.curr && p.s.stride==q.s.stride && p.s.start==q.s.start;
  }
  template<class T> bool operator!=
  (const Slice_iter<T> &p, const Slice_iter<T> &q)
  {
    return  !(p==q);
  }
  template<class T> bool operator<
  (const Slice_iter<T> &p, const Slice_iter<T> &q)
  {
    return p.curr<q.curr && p.s.stride==q.s.stride && p.s.start==q.s.start;
  }
}
```

Matrix.h

```
#include "Slice_iter.h"

class Matrix {
  valarray<double> *v;
  size_t d1, d2;

public:
  Matrix(size_t x, size_t y);
  Matrix(const Matrix &);
  Matrix &operator=(const Matrix &);
  ~Matrix();

  size_t size() const { return d1 * d2; }
  size_t dim1() const { return d1; }
  size_t dim2() const { return d2; }
  Slice_iter<double> row(size_t i);
  Cslice_iter<double> row(size_t i) const;
  Slice_iter<double> column(size_t i);
  Cslice_iter<double> column(size_t i) const;
  double &operator()(size_t x, size_t y);
  double operator()(size_t x, size_t y);
  Slice_iter<double> operator() (size_t i) { return row(i); }
  Cslice_iter<double> operator() (size_t i) const { return row(i); }
  Slice_iter<double> operator[] (size_t i) { return row(i); }
  Cslice_iter<double> operator[] (size_t i) const { return row(i); }
  Matrix& operator*=(double);
  valarray<double>& array() { return *v; }
}

Matrix::Matrix(size_t x, size_t y)
{
  d1 = x;
  d2 = y;
  v = new valarray<double>(x*y);
}

double Matrix::operator()(size_t x, size_t y)
{
  return row(x)[y];
}

double mul(const valarray<double> &v1, const valarray<double> &v2)
{
  double res = 0;
  for (int i = 0; i < v1.size(); i++) res += v1[i] * v2[i];
  return res;
}

valarray<double> operator*(const Matrix &m, const valarray<double> &v)
{
  valarray<double> res(m.dim1());
  for (int i = 0; i < m.dim1(); i++) res(i) = mul(m.row(i), v);
  return res;
}

Matrix &Matrix::operator*=(double d)
{
  (*v) *= d;
  return *this;
}

inline Slice_iter<double> Matrix::row(size_t i)
{
```

D.12 注目しているソフトウェア

D.12.1 FreeFEM

D.12.2 Blitz

D.12.3 MATPACK++

D.13 URL

http://www.asahi-net.or.jp/~uc3k-ymd/
『C/C++ プログラミング』
面白い。

http://www.interq.or.jp/jazz/iijima/equations/equations00.html
『C++で連立方程式を解いてみよう！』

http://ccinfo.ims.ac.jp/cpp/cpp.html
『C++ Programmer's Page at Computer Center of IMS』

http://sklab-www.pi.titech.ac.jp/~hase/soft/ptm/PTM.html
『ポータブル テンプレート 行列クラスライブラリ』

http://www.is.titech.ac.jp/~kando9/
『』

http://www.is.titech.ac.jp/~kando9/MTAbstract.html
『名古屋大学 工学研究科 情報工学専攻 修士学位論文
「数値計算ライブラリのインターフェースに関する研究」(1993 年 2 月提出) の要旨』

http://www.asahi-net.or.jp/~uc3k-ymd/Lesson/Section03/section03_10.html
『3.10 最適化 (1999.09.04 初版)』

http://www.seigyo.numazu-ct.ac.jp/~hase/subject/S3_PL/s3_pl_jap.html
『NCT Department of Control & Computer Engineering
S3_PL - 演習：プログラミング言語 C ++』

http://www.uopmu.ees.osakafu-u.ac.jp/~yabu/soft/c-cal.html
『C, C++ による数値計算プログラミング』

http://www.ns.kogakuin.ac.jp/~em01014/bbs/minibbs.cgi?log=log1

http://www.asahi-net.or.jp/~uc3k-ymd/Lesson/Section02/except.html

`catch,throw` など

D.14 参考文献

Stroustrup [3] の 22 章「数値演算」は必読であろう。自分でクラスを作るには、Koenig and Moo [1] の？章を読むべきである。

関連図書

[1] Andrew Koenig and Barbara E.Moo, Ruminations on C++, Addison-Wesley (1997).

Andrew Koenig, Barbara E.Moo 著, 小林 健一郎 訳, C++ 再考, アジソン・ウェスレイ・パブリッシャーズ・ジャパン (1998).

[2] P.J.Plauger, The Draft Standard C++ Library, Prentice Hall (1995).

P.J. プラウガー著, 蓬莱尚幸, ドラフト標準 C++ ライブラリ, プレンティスホール出版発行, 株式会社トッパン発売 (1995).

[3] Bjarne Stroustrup, The C++ Programming Language Third Edition, Addison Wesley (1997).

Bjarne Stroustrup, プログラミング言語 C++ 第 3 版, アスキー (1998).

[4] 小林 健一郎, これならわかる C++, 講談社 (2001).

付 録E　Java もやってみよう