常微分方程式の初期値問題の数値計算入門

桂田祐史

since 1995〜2011 年 4 月 29 日, 2021 年 6 月 6 日

(以前は「…数値解法入門」という題だったけれど、変更しました。)

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

目 次

1 はじめに: この文書は何か

数学科 2 年生向けの「情報処理 II」という講義 (1995 年) の中で、常微分方程式の初期値問題の数値計算入門を行いました。そのときの資料をまとめたものです。解法としては、Euler 法と Runge-Kutta 法しか説明していませんが、それだけでも結構色々なことが出来ます。

なんと言っても古いので、「今なら違う説明にする」、「ちょっとおかしいかな、修正すべきかな」と思うところがないわけではないけれど、直し始めるとキリがないので、そのママにしてあります。(書いた当時はそれなりに頑張ったので、作り直すにもかなり手間がかかる、ということです。)

プログラムについては、当時使っていた自作グラフィックス・ライブラリィを、現在でも利用可能な GLSC を使うように書き換えました。プログラミング言語として、C 言語を使っています。これについては、今ならば他の選択肢がたくさんあるところで、個人的には Julia または Crystal 推しですが、そういうのを使う例は別に提示しようと言うことで、C 言語バージョンはこのままの形で残します。

(2021/6/6 追記) フラストレーションがたまるので、やはり新しいのを書くことにしました。でも、この文書は大体このままの形で残します。

2 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)

2.1 はじめに

数学科・現象数理学科の学生向けに、常微分方程式の初期値問題の数値計算法について解説する。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ることが目標である。

2.2 目的とする問題

常微分方程式の初期値問題

$$\frac{dx}{dt} = f(t,x) \tag{1}$$

$$x(a) = x_0 \tag{2}$$

を考える[1]。つまり f, a, x_0 が与えられたとき、(1), (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 $\mathbb{R} \times \mathbb{R}^n = \mathbb{R}^{n+1}$ のある開集合 Ω 上定義され、$\mathbb{R}^n$ に値を取る関数である: $f\colon \Omega \to \mathbb{R}^n$
- $x_0 \in \mathbb{R}^n$
- $(a, x_0) \in \Omega$

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。

2.3 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** (the discrete variable method) 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

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

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[2]。

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。

---

[1] (1) は $\frac{dx}{dt}(t) = f(t, x(t))$ と書く方が誤解がないかもしれないが、(t) は省略されることが多い。

[2] 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。

2.4 Euler 法

微分係数の定義より、h が十分小さければ

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

と近似できる。

そこで

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

から $\{x_j\}_{j=0}^N$ に関する方程式

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

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)。

(4) を整理して、

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

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**前進 Euler 法**である[3]。前進 Euler 法は素朴であるが、次の意味で「うまく働く」。

> f が連続かつ x について Lipschitz 条件を満たす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。

(一方で、後退 Euler 法は、無条件に安定となるため、しばしば利用される。)

2.5 Runge-Kutta 法

漸化式

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

ただし、

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

で $\{x_j\}_{j=1}^N$ を計算する方法を **Runge-Kutta 法**という[4]。

Runge-Kutta 法は、適度に簡単で、そこそこの効率を持つ方法であるため、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、

---

[3]単に Euler 法と呼ばれることも多いが、**後退 Euler 法**というものがあるので、それと区別するために**前進 Euler 法**と呼ばれる。

[4]Runge-Kutta 法にはたくさんの親戚がいるので、ここで紹介したものを、「古典的 Runge-Kutta 法」、「4 次の Runge-Kutta 法」と呼ぶこともある。

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。

2.6 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。

2.6.1 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

```
/*
 * prog1array-ansi.c
 */

#include <stdio.h>

#define N 100

int main(void)
{
    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

```
/*
 * prog1non-array.c
 */

#include <stdio.h>

#define N 100

int main(void)
{
    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;
}
```

2.6.2 F がベクトル値の場合

F がベクトル値 $F = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$ 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、ベクトルを扱えるプログラミング言語 (C++, Python, Ruby, Julia, ⋯) を用いれば、2.6.1 と同じような感じでプログラムが書ける。)

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

```
/*
 * prog2wrong.c
 */

#include <stdio.h>

#define N 100

int main(void)
{
    int j;
    double a, b, F1(double, double), F2(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 = F1(a, b);
        b = F2(a, b);
        printf("(a[%d],b[%d])=(%g,%g)\n", j+1, j+1, a, b);
    }
    return 0;
}

double F1(double x, double y)
{
    return x / 2 + y / 3 + 1.0;
}

double F2(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

```
/*
 * prog2right.c
 */

#include <stdio.h>

#define N 100

int main(void)
{
    int j;
    double a, b, newa, newb, F1(double, double), F2(double, double);

    printf("a[0], b[0]: "); scanf("%lf %lf", &a, &b);
    for (j = 0; j < N; j++) {
        newa = F1(a, b);
        newb = F2(a, b);
        a = newa;
        b = newb;
        printf("(a[%d],b[%d])=(%g,%g)\n", j+1, j+1, a, b);
    }
    return 0;
}

double F1(double x, double y)
{
    return x / 2 + y / 3 + 1.0;
}

double F2(double x, double y)
{
    return - x / 3 + y / 2 - 0.5;
}
```

3 常微分方程式の初期値問題の数値解法

ここからの 3 節は、ある時期、数学科 2 年生向けの講義『情報処理 II』で講義していた内容である (「1995 年度情報処理 II」[5])。ですます調で、統一が取れないが、そのままマージしておく。

コンパイルに gcc を使っているが、Mac の場合は cc で構わない。(その場合 `-lm` は指定する必要はない)。

3.1 はじめに

3.1.1 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 $\frac{dx}{dt}, \frac{d^2x}{dt^2}, \cdots, \frac{d^kx}{dt^k}$ 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 $\frac{dx}{dt} = x'$, $\frac{d^2x}{dt^2} = x''$, $\frac{d^3x}{dt^3} = x^{(3)}$, $\cdots$ のような書き方もします。)

(例 1) $x'(t) = f(t)$ (f は既知関数)

[5] http://nalab.mind.meiji.ac.jp/~mk/syori2-1995/

(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。

3.1.2 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[6]、数学それ自体にとっても非常に重要です[7]。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。これはコンピューターによる数値シミュレーション[8]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。

3.2 数値解法 (1)

3.2.1 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[9]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

$$\frac{dx}{dt} = f(t, x) \quad (t \in (a, b)) \tag{8}$$

[6]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。

[7]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 (これはいつの学習指導要領かによる)、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。

[8]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。

[9]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。

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

$$x(a) = x_0 \quad (x_0 \text{ は既知定数}) \tag{9}$$

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**[10]を作り、それを解く。」

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。

3.2.2 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

$$x_{j+1} = x_j + hf(t_j, x_j). \tag{10}$$

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[11]。ここでは簡単な例で収束を確かめてみましょう。

3.2.3 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 (10) 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、C 言語では自然な発想です。例えば

```
#define MAXN (1000)

  double x[MAXN+1];
```

のように配列 "x" を用意しておいて

[10]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。
[11]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。

```
x[0] = x0;
t = a;
for (j = 0; j < N; j++) {
  x[j+1] = x[j] + h * f(t,x[j]);
  t += h;
}
```

とするわけです。Fortran だったら、

Fortran の場合

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

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
```

という具合いです。

補足的注意　C 言語の場合は、配列の代わりに、ポインターと malloc() を使って

```
#include <stdlib.h> // malloc()
...
  double *x;
...
  x = malloc(sizeof(double) * (N+1));
  if (x == NULL) {
    // エラー処理
  }
```

のように動的にメモリーを取得することも出来ます。この後は (x が配列の場合と) 同様に使えます。

配列を使わないですませる方法　漸化式 (10) 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[12]。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;
for (j = 0; j < N; j++) {
  x += h * f(t,x);
  t += h;
}
```

のようなプログラムで計算が出来ます。Fortran だったら次のようになります。

[12]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。

Fortran の場合

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

重箱の隅をつつく注意: “`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;
```

3.2.4 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 $x_N = \left(1 + \dfrac{1}{N}\right)^N$ 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.c

```
/*
 * reidai5-1.c -- 微分方程式の初期値問題を Euler 法で解く
 * http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/reidai5-1.c
 */

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

int main(void)
{
  /* 開始時刻と終了時刻 */
  double a = 0.0, b = 1.0;
  /* 変数と関数の宣言 */
  int N, j;
  double t,x,h,x0,f(double, double);
  /* 初期値 */
  x0 = 1.0;
  /* 区間の分割数 N を入力してもらう */
  printf("N="); scanf("%d", &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;
  printf("t=%f, x=%f\n", t, x);
  /* Euler 法による計算 */
  for (j = 0; j < N; j++) {
    x += h * f(t,x);
    t += h;
    printf("t=%f, x=%f\n", t, x);
  }

  return 0;
}

/* 微分方程式 x'=f(t,x) の右辺の関数 f の定義 */
double f(double t, double x)
{
  return x;
}
```

このプログラムをコンパイルして[13]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.c" 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。

[13]コンパイルすると、"reidai5-1.c:34: warning: unused parameter '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)。

3.2.5 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.c

```
/*
 * reidai5-2.c --- 常微分方程式の初期値問題を Euler 法で解く
 * http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/reidai5-2.c
 */

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

int main(void)
{
  /* 開始時刻と終了時刻 */
  double a = 0.0, b = 1.0;
  /* 初期値 */
  double x0;
  /* 変数と関数の宣言 */
  double t,x,h,f(double,double),e;
  int N0,N1,r,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、r をかけて増える N に */
  printf("# FIRSTN,LASTN,r=");
  scanf("%d%d%d", &N0,&N1,&r);
  /* 計算の開始 */
  for (N = N0; N<= N1; N *= r) {
    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 */
    for (i = 0; i < N; i++) {
      x += h * f(t,x);
      t += h;
    }
    printf("%d %e\n", N, fabs(e-x));
  }
  return 0;
}

/* 微分方程式 x'=f(t,x) の右辺の関数 f の定義 */
double f(double t, double x)
{
  return x;
}
```

コンパイルして実行すると “ FIRSTN,LASTN,r=” と尋ねてきます。例えば “10 1280 2” と答えると、10 から始めて 2 倍ずつしていって 1280 を超えないまでの値 (つまり 10, 20, 40, 80, 160, 320,

640, 1280) 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。

コンパイル&実行

```
oyabun% gcc -o reidai5-2 reidai5-2.c -lm
oyabun% ./reidai5-2
# FIRSTN,LASTN,r=10 1000 2
10 1.245394e-01
20 6.498412e-02
40 3.321799e-02
80 1.679689e-02
160 8.446252e-03
320 4.235185e-03
640 2.120621e-03
1280 1.061069e-03
oyabun%
```

出力結果を保存して作ったファイル “reidai5-2.data” の内容を gnuplot でグラフにするには、以下のようにします。**両側対数目盛**によるグラフを描く機能を用いています。

```
oyabun% ./reidai5-2 > reidai5-2.data
10 1280 2
oyabun%

これで計算結果を reidai5-2.data に記録できた。この内容を gnuplot でプロットする。

oyabun% gnuplot

        G N U P L O T
        Unix version 3.7
        (以下色々なメッセージ…省略)

gnuplot> set logscale xy
gnuplot> plot "reidai5-2.data" with linespoints

  (以下印刷用のデータ作り)
gnuplot> set term postscript eps color
Terminal type set to 'postscript'
Options are 'eps noenhanced color dashed defaultplex "Helvetica-Ryumin"
14'
gnuplot> set output "reidai5-2.eps"
gnuplot> replot
gnuplot> quit
oyabun%
```

(2021/4/1 追記: ここでは EPS (Encapsulated PostScript) 形式にしていますが、現在では `set term png` や `set term pdf` とする方が良いかも。)

このグラフから、誤差 $= O(N^{-1})$ $(N \to +\infty)$ 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…

3.3 Runge-Kutta 法

3.2.5 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

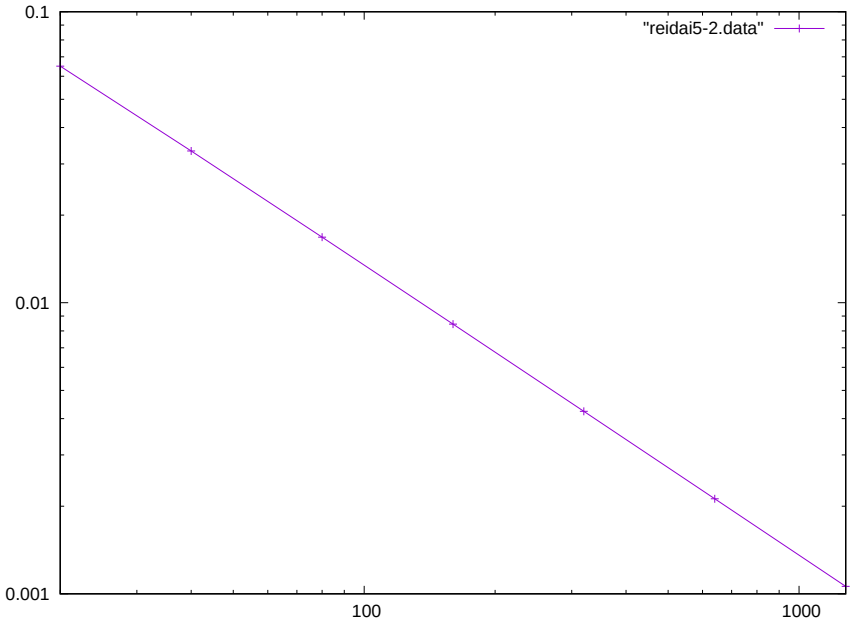

図 1: Euler 法の誤差 (横軸: 分割数 N, 縦軸: 誤差の絶対値)

の計算をする) 必要があります。特別な場合[14]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。

[14]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.c

```
/*
 * reidai5-3.c --- 常微分方程式の初期値問題を Runge-Ketta 法で解く
 * http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/reidai5-3.c
 */

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

int main(void)
{
  /* 開始時刻と終了時刻 */
  double a = 0.0, b = 1.0;
  /* 初期値 */
  double x0;
  /* 変数と関数の宣言 */
  int N,j;
  double t,x,h,f(double,double),k1,k2,k3,k4;
  /* 初期値の設定 */
  x0 = 1.0;
  /* 区間の分割数 N を入力してもらう */
  printf("N="); scanf("%d", &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;
  printf("%f %f\n", t, x);
  /* Runge-Kutta 法による計算 */
  for (j = 0; j < N; j++) {
    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 += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    t += h;
    printf("%f %f\n", t, x);
  }
  printf("%f %17.15f\n", t, x);
  return 0;
}

/* 微分方程式 x'=f(t,x) の右辺の関数 f の定義 */
double f(double t, double x)
{
  return x;
}
```

コンパイル・実行の結果は次のようになるはずです。

```
oyabun% gcc reidai5-3.c
oyabun% ./a.out
N=10
0.000000 1.000000
0.100000 1.105171
0.200000 1.221403
0.300000 1.349858
0.400000 1.491824
0.500000 1.648721
0.600000 1.822118
0.700000 2.013752
0.800000 2.225540
0.900000 2.459601
1.000000 2.718280
1.000000 2.718279744135166              ← たくさんの桁数を表示
oyabun%
```

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$=|e - 2.718279744135166|/e \fallingdotseq 7.67 \times 10^{-7}$)。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.c” で、大きな N に対してどうなるか、実験しなさい。

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 (3.2.5 の “reidai5-2.c” の真似をする)。Euler 法と比べてどうなるか？

4　定数係数線形常微分方程式

インターネットに接続されたWWWブラウザーがあれば、http://nalab.mind.meiji.ac.jp/~mk/labo/java/prog/ODE1.html にアクセスすると実験出来る (かもしれません) [15]。

4.1　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

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

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。

[15] これは Java で書かれたアプレットです。ソースは http://nalab.mind.meiji.ac.jp/~mk/labo/java/prog/ODE1.java にあります。 — (2021/4/1 追記) 現在では、セキュリティ上の理由で、このような Java アプレットの利用は認められなくなりました。残念ながら実行できなくなっています。

注意 1　この問題は、コンピューター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。

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

とおくと[16]、$\vec{x} = \vec{x}(t)$ 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 $\mathbb{R}^2$ 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 $\mathbb{R}^2$) を相空間 (phase space)[17]と呼びます。

$f(t, \vec{x}) := Ax$ 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、

$$y := \frac{dx}{dt}, \quad \vec{x} = \begin{pmatrix} x \\ y \end{pmatrix}, \quad A = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix}$$

と置くことにより、(1) の形に帰着されます。同様の置き換えで、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

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

[16] 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。

[17] “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。「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しょう。

4.2 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。

4.2.1 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。

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

ターミナルでプログラム入手・コンパイル・実行

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/reidai7-1-glsc.c
cglsc reidai7-1-glsc.c
./reidai7-1-glsc
```

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。$(a, b, c, d) = (0, 1, -1, 0)$ は単振動 $(m = 1,\ k = 1)$、$(a, b, c, d) = (0, 1, -1, -1)$ は減衰振動 $(m = 1,\ k = 1,\ \gamma = 1)$ となります。

行列の成分を入力

```
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指定するメニュー

```
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。

初期値をキーボードから数値入力

```
0                          ← 0 番を選択する。
初期値 x0,y0=0.5 0.5       → x0,y0 の入力の催促。← 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。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すると、その点を初期値として解を計算して、解軌道を描きます。

初期値をマウスで指定

```
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)。
 -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)
1    ← マウスで初期値を指定
(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 が減少する方向に解きます (過去にさかのぼる)。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。

(2021/4/1 追記) Mac で 3 ボタン・マウスを接続して使っている人は少数派でしょう。そういう場合にも、XQuartz の [環境設定][入力] で、「3 ボタンをエミュレート」にチェックを入れることで、このプログラムを使うことができます。

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。

ターミナルで (入力データファイルを入手して解く)

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/reidai7-1.data
cat reidai7-1.data | ./reidai7-1-glsc
```

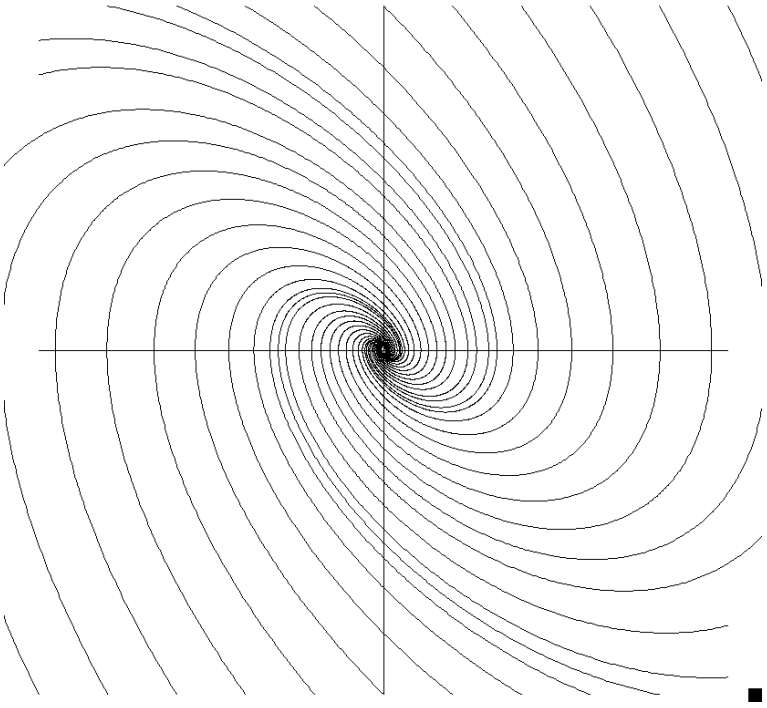

図 2: `cat reidai7-1.data | ./reidai7-1-glsc`

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。

4.2.2　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？)

$$\begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix}, \quad \begin{pmatrix} -\frac{4}{5} & -\frac{3}{5} \\ \frac{2}{5} & -\frac{11}{5} \end{pmatrix}, \quad \begin{pmatrix} \frac{8}{5} & -\frac{9}{5} \\ \frac{6}{5} & -\frac{13}{5} \end{pmatrix}, \quad \begin{pmatrix} \frac{2}{5} & \frac{9}{5} \\ -\frac{1}{5} & \frac{8}{5} \end{pmatrix}, \quad \begin{pmatrix} -1 & 2 \\ -1 & 1 \end{pmatrix}.$$

(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。)

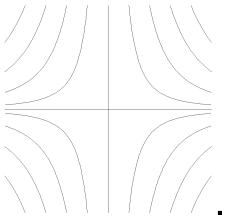

図 3: $\begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix}$

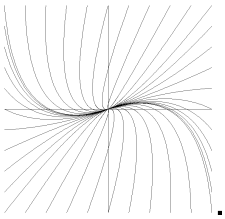

図 4: $\begin{pmatrix} -\frac{4}{5} & -\frac{3}{5} \\ \frac{2}{5} & -\frac{11}{5} \end{pmatrix}$ (吸)

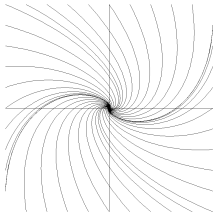

図 5: $\begin{pmatrix} \frac{8}{5} & -\frac{9}{5} \\ \frac{6}{5} & -\frac{13}{5} \end{pmatrix}$ (湧)

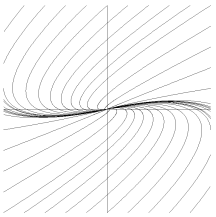

図 6: $\begin{pmatrix} \frac{2}{5} & \frac{9}{5} \\ -\frac{1}{5} & -\frac{8}{5} \end{pmatrix}$ (湧)

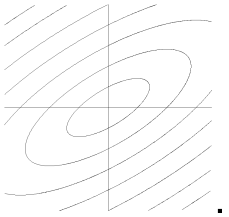

図 7: $\begin{pmatrix} -1 & 2 \\ -1 & 1 \end{pmatrix}$

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。

4.2.3　ソースプログラム reidai7-1-glsc.c

(情報処理 II の授業で使っていたコンピューター環境は古いので、当時使っていたプログラムを GLSC を利用するように書き換えたもの。)

```
/*
 * reidai7-1-glsc.c --- reidai7-1.c を GLSC を用いるように書き換えた
 * http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/reidai7-1-glsc.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 <stdlib.h> // system()
#include <math.h>
#include <glsc.h>

// 係数行列 A の成分
double a, b, c, d;
// ウィンドウに表示する範囲
double xleft, xright, ybottom, ytop;

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

void g_dump(char *fname, Display *display, Window wid)
{
  char command[256];
  sprintf(command, "import -window %lu %s", wid, fname);
  system(command);
}

int main(void)
{
  // 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 but;
  //
  double win_width, win_height, w_margin, h_margin;
  //
  Display *display;
  Window wid; // Window ID
  // 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
  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);
  g_device(G_BOTH);
  g_def_scale(0, xleft, xright, ybottom, ytop,
              w_margin, h_margin, win_width, win_height);
  g_sel_scale(0);
```

```
  // x 軸、y 軸を描く
  g_move(xleft, 0.0); g_plot(xright, 0.0);
  g_move(0.0, ybottom); g_plot(0.0, ytop);
  // メイン・ループの入口
  do {
    // 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:初期値のキーボード入力, 1:初期値のマウス入力\n");
    printf("   2: 刻み幅, 追跡時間変更 (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) {
      do {
        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");
        g_mouse_sence(&x0, &y0, &but);
        printf("x0=%g, y0=%g, but=%d\n", x0, y0, but);
        if (but == 1) {
          printf("左ボタン, (x0,y0)=%f %f\n", x0,y0);
          draworbit(x0,y0,h,tlimit);
        }
        else if (but == 2) {
          printf("中ボタン, (x0,y0)=%f %f\n", x0,y0);
          draworbit(x0,y0,-h,tlimit);
        }
      }
      while (but != 3);
      printf("右ボタンがクリックされました。マウスによる初期値の入力を打ち切ります。\n");
    }
    else if (cmd == 2) {
      // 時間刻み幅、追跡時間の変更
      printf("時間刻み幅 h (%g), 追跡時間 T (%g): ", h, tlimit);
      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);
      }
    }
  }
  while (cmd != -1);

  g_dump("reidai7-1.png", g_get_display(), g_get_window());

  printf("GLSC ウィンドウを左ボタンでクリックして下さい\n");
  g_sleep(-1.0);
}

// 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)
{
  int in;
  double x, y, fx(double, double), fy(double, double);
  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
  if ((in = (xleft <= x && x <= xright && ybottom <= y && y <= ytop)) != 0)
    g_move(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
    if (in) {
      if ((in = (xleft <= x && x <= xright && ybottom <= y && y <= ytop)) != 0)
        g_plot(x,y);
    }
    else {
      if ((in = (xleft <= x && x <= xright && ybottom <= y && y <= ytop)) != 0)
        g_move(x,y);
    }
    // 時刻を 1 ステップ分進める
    t += h;
  }
  while (fabs(t) <= fabs(tlimit)); // まだ範囲内かどうかチェック
}

// 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;
}
```

4.3 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。

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

4.3.2 $N = 2$ 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 $x_0 \in \mathbb{R}^2$ は

$$x_0 = c_1u_1 + c_2u_2$$

と u_1, u_2 の線形結合で表すことが出来る。これから

$$A^nx_0 = A^n(c_1u_1 + c_2u_2) = c_1A^nu_1 + c_2A^nu_2 = c_1{\lambda_1}^nu_1 + c_2{\lambda_2}^nu_2 = {\lambda_1}^n(c_1u_1) + {\lambda_2}^n(c_2u_2),$$

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

(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 $\mathbb{R}^2 \neq \ker(\lambda_0 I - A)$ であり、$u_2 \in \mathbb{R}^2 \setminus \ker(\lambda_0 I - A)$ 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 $\lambda = \alpha \pm i\beta$ $(\alpha, \beta \in \mathbb{R}, i = \sqrt{-1})$ を持つ場合

$\alpha + i\beta$ に属する固有ベクトルの一つを $x + iy$ $(x, y \in \mathbb{R}^2)$ 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。

5 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。

5.1 力学系と Poincaré のリミット・サイクル

5.1.1 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

$$\frac{dx}{dt} = f(x) \tag{11}$$

という形の方程式を**力学系** (**dynamical**[18]**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。

[18] “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。

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[19]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。

5.1.2 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コンピューター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。

例題 8-1 次の力学系の流れの様子を $-4 \le x, y \le 4$ 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)。「$-4 \le x, y \le 4$ 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、

ターミナルで実行 (入手, コンパイル, 実行)

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/reidai8a-glsc.c
cglsc reidai8a-glsc.c
./reidai8a-glsc.c
範囲 (xleft,ybottom,xright,ytop)?
-4 -4 4 4
```

この後は reidai7-1 と同様に操作できます。あるいは、こちらが用意したサンプル・データ `rei8a.data` を使って

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/rei8a.data
cat rei8a.data | ./reidai8a-glsc
```

としても 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

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

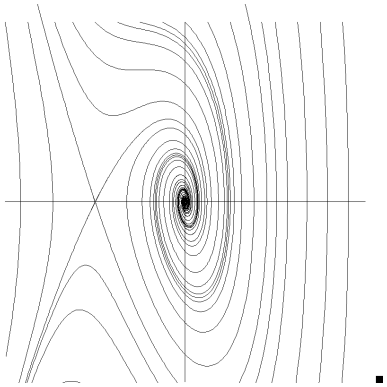

図 8: $\frac{d}{dt}\binom{x}{y} = \binom{y}{-6x-y-3x^2}$

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？)。

```
% ./reidai8a-glsc
 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。)

5.1.3　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[20] 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-glsc.c` 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図 9 です。

試してみよう

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/reidai8b-glsc.c
cglsc reidai8b-glsc.c
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/rei8b.data
cat rei8b.data | ./reidai8b-glsc
```

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

[20] ファンデルポール、と読みます。

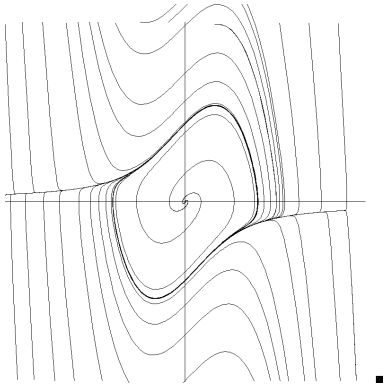

図 9: van der Pol の方程式の相図

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。

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

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。それはコンピューター・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、コンピューター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。

5.2　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 $g \approx 9.8\mathrm{m/sec}^2$ 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。

6　他のプログラミング言語でのプログラム例

2021 年現在、MATLAB や Julia, Crystal, Python などが良い選択肢と考えているが、参考まで。

6.1　十進 BASIC

(仮称) 十進 BASIC というプログラミング言語は、気軽に試せる、1000 桁演算モードがある、などの特徴があって、プログラミング入門用に一時期勧めていました。

```
REM dampedoscillation.bas
REM http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/dampedoscillation.BAS
DEF fx(t,x,y)=a*x+b*y
DEF fy(t,x,y)=c*x+d*y
LET maxn=1000
DIM x(0 TO maxn),y(0 TO maxn)

SUB runge(t0,x0,y0,h,n)
   LET t=t0
   LET x(0)=x0
   LET y(0)=y0
   PLOT LINES : x(0),y(0);
   PRINT t,x(0),y(0)
   FOR j=0 TO n-1
      LET k1x=h*fx(t,x(j),y(j))
      LET k1y=h*fy(t,x(j),y(j))
```

```
      LET k2x=h*fx(t+h/2,x(j)+k1x/2,y(j)+k1y/2)
      LET k2y=h*fy(t+h/2,x(j)+k1x/2,y(j)+k1y/2)
      LET k3x=h*fx(t+h/2,x(j)+k2x/2,y(j)+k2y/2)
      LET k3y=h*fy(t+h/2,x(j)+k2x/2,y(j)+k2y/2)
      LET k4x=h*fx(t+h,x(j)+k3x,y(j)+k3y)
      LET k4y=h*fy(t+h,x(j)+k3x,y(j)+k3y)
      LET x(j+1)=x(j)+(k1x+2*k2x+2*k3x+k4x)/6
      LET y(j+1)=y(j)+(k1y+2*k2y+2*k3y+k4y)/6
      LET t=t+h
      PLOT LINES : x(j+1),y(j+1);
      PRINT t,x(j+1),y(j+1)
   NEXT j
   PLOT LINES
END SUB

LET a=0
LET b=1
LET c=-1
LET d=-0.1

SET WINDOW -1,1,-1,1
DRAW grid

LET Ts=0
LET Te=100
LET N=1000
LET h=(Te-Ts)/N
SET LINE COLOR 2
CALL runge(Ts, 0.7, 0.5, h, N)
END
```

6.2 Java

Java の良いところは、GUI のプログラムがほどほどの手間で書けて、それはシミュレーションの便利さに通じる (例えばパラメーターを調節するとか) ことでしょうか。
(2021/4/13 追記) 以下のサンプル・プログラムは、Java アプレットであるが、最近の Java ではアプレットは廃止されてしまった。こういうのが WWW で試せるのは面白いと思ったのだけれど…(直す気はなくなった。)

```
/*
 * ConstLinear2D.java --- 2 次元定数係数線型常微分方程式
 */

// <APPLET code="ConstLinear2D.class" width=500 height=500> </APPLET>

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

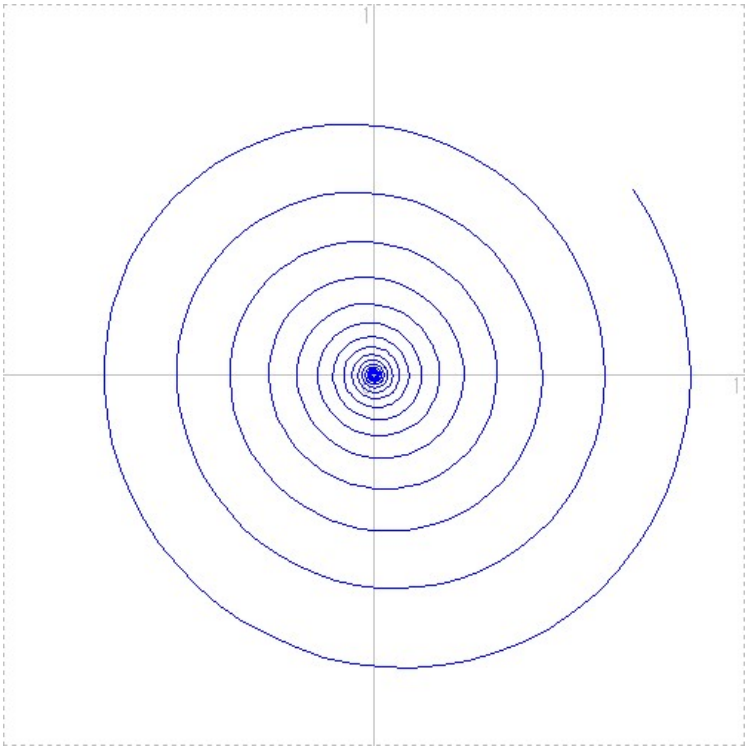

図 10: 減衰振動 (仮称十進 BASIC プログラムによる)

```
class GraphCanvas extends Canvas {

  static final boolean DEBUG = false; // true;
  private static final String message = "graph of a function with 1 variable";
  // 問題に取って基本的なパラメーター
  // 係数行列
  private double a, b, c, d;
  // 描画範囲
  private double x_max =   1.0, x_min = - 1.0, y_max = 1.0, y_min = - 1.0;
  private double x_margin = (x_max - x_min) / 10;
  private double y_margin = (y_max - y_min) / 10;

  // 座標系の変換のためのパラメーター
  private int CanvasX = 400, CanvasY = 400;
  private double ratiox, ratioy, X0, Y0;

  // コンストラクター
  public GraphCanvas() {
    super();
  }
  public GraphCanvas(int cx, int cy) {
    super();
    CanvasX = cx; CanvasY = cy;
  }

  public void compute(double A, double B, double C, double D) {
    a = A; b = B; c = C; d = D;
    repaint();
  }

  private boolean IsIn(double x, double y) {
    return (x_min <= x && x <= x_max && y_min <= y && y <= y_max);
  }
  // 座標変換の準備
  private void space(double x0, double y0, double x1, double y1) {
    X0 = x0; Y0 = y0;
```

```
  ratiox = CanvasX / (x1 - x0);
  ratioy = CanvasY / (y1 - y0);
}
// ユーザー座標（ワールド座標系）をウィンドウ座標（デバイス座標系）
private int wx(double x) {
  return (int)(ratiox * (x - X0));
}
// ユーザー座標（ワールド座標系）をウィンドウ座標（デバイス座標系）
private int wy(double y) {
  return CanvasY - (int)(ratioy * (y - Y0));
}
// 力学系の右辺 f=(fx, fy)
private double fx(double x, double y) {
  return a * x + b * y;
}
private double fy(double x, double y) {
  return c * x + d * y;
}
// (x0,y0) を初期値とする解の軌道（trajectory）を描く
private void drawTrajectory(Graphics g, double x0, double y0, double T) {
  double x, y, new_x, new_y;
  double k1x, k1y, k2x, k2y, k3x, k3y, k4x, k4y;
  double h = 0.01 / Math.sqrt(a * a + b * b + c * c + d * d);
  if (T < 0.0)
    h = - h;
  int iter = (int)Math.rint(Math.abs(T / h));
  x = x0;
  y = y0;
  for (int i = 1; i <= iter; i++) {
    k1x = h * fx(x, y);
    k1y = h * fy(x, y);
    k2x = h * fx(x + k1x / 2, y + k1y / 2);
    k2y = h * fy(x + k1x / 2, y + k1y / 2);
    k3x = h * fx(x + k2x / 2, y + k2y / 2);
    k3y = h * fy(x + k2x / 2, y + k2y / 2);
    k4x = h * fx(x + k3x, y + k3y);
    k4y = h * fy(x + k3x, y + k3y);
    new_x = x + (k1x + 2 * k2x + 2 * k3x + k4x) / 6;
    new_y = y + (k1y + 2 * k2y + 2 * k3y + k4y) / 6;
    if (IsIn(x, y) && IsIn(new_x, new_y))
        g.drawLine(wx(x), wy(y), wx(new_x), wy(new_y));
    x = new_x; y = new_y;
  }
}

public void paint(Graphics g) {
  //
  space(x_min - x_margin, y_min - y_margin,
        x_max + x_margin, y_max + y_margin);
  //
  setBackground(Color.blue);
  //
  g.setColor(Color.black);
  g.drawLine(wx(x_min), wy(0.0), wx(x_max), wy(0.0));
  g.drawLine(wx(0.0), wy(y_min), wx(0.0), wy(y_max));
  //
  g.setColor(Color.yellow);
  int n = 36;
  double dt = 2 * Math.PI / n;
  double Time = 10.0 / Math.sqrt(a * a + b * b + c * c + d * d);
  for (int i = 0; i < n; i++) {
    double t = i * dt;
    drawTrajectory(g, Math.cos(t), Math.sin(t),   Time);
```

```
      drawTrajectory(g, Math.cos(t), Math.sin(t), - Time);
    }
  }
}

public class ConstLinear2D extends Applet implements ActionListener {

  private int N = 20;
  private double lambda = 0.5;
  private double Tmax = 0.5;
  // ユーザーとのインターフェイス（パラメーターの入力）
  private Label label_a, label_b, label_c, label_d;
  private TextField input_a, input_b, input_c, input_d;
  private double a = 1.0, b = 0.0, c = 0.0, d = 1.0;
  private Button startB;
  //
  private GraphCanvas gc;

  private void ReadFields() {
    a = Double.valueOf(input_a.getText()).doubleValue();
    b = Double.valueOf(input_b.getText()).doubleValue();
    c = Double.valueOf(input_c.getText()).doubleValue();
    d = Double.valueOf(input_d.getText()).doubleValue();
  }
  // 準備（変数の用意と座標系の初期化など）
  public void init() {
    // ナル・レイアウト
    setLayout(null);
    // a, b, c, d を入力するためのテキスト・フィールド
    add(label_a = new Label("a=")); label_a.setBounds(100, 30, 40, 30);
    add(label_b = new Label("b=")); label_b.setBounds(250, 30, 40, 30);
    add(label_c = new Label("c=")); label_c.setBounds(100, 70, 40, 30);
    add(label_d = new Label("d=")); label_d.setBounds(250, 70, 40, 30);
    add(input_a = new TextField("" + a)); input_a.setBounds(150, 30, 100, 30);
    add(input_b = new TextField("" + b)); input_b.setBounds(300, 30, 100, 30);
    add(input_c = new TextField("" + c)); input_c.setBounds(150, 70, 100, 30);
    add(input_d = new TextField("" + d)); input_d.setBounds(300, 70, 100, 30);
    // 再計算ボタン
    startB = new Button("Restart");
    add(startB);
    startB.setBounds(420, 45, 50, 30);
    startB.addActionListener(this);

    // キャンバス
    gc = new GraphCanvas();
    add(gc);
    gc.setBounds(50, 100, 400, 400);
    ReadFields();
    gc.compute(a, b, c, d);
  }

  // ボタンを押されたら、テキスト・フィールドの内容を読み取って、再描画
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == startB) {
      ReadFields();
      gc.compute(a, b, c, d);
    }
  }
}
```

初期値の選択、時間の逆転、マウスで初期値、など拡張したい。

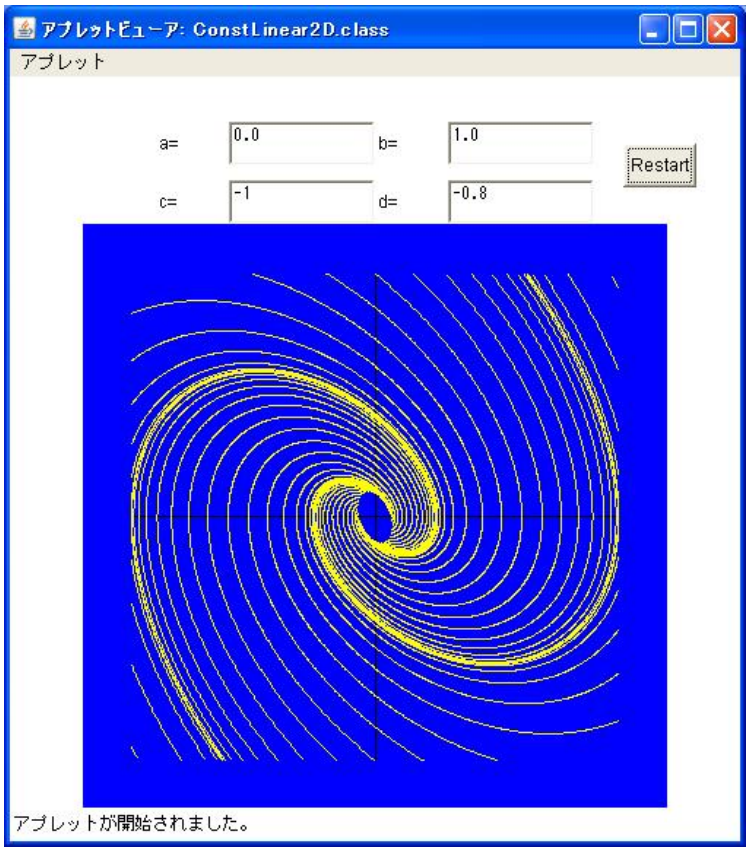

図 11: 定数係数線形常微分方程式を Java アプレットで解く

6.3 C++ & Eigen

C++ の良いところは、C に近いので C に慣れているとやりやすいと言う点と、ベクトルが扱えるところ、実行効率が高いところでしょう。

6.3.1 Eigen のインストール

Eigen[21] から、eigen-3.3.9.tar.gz を入手して次のようにしてインストールする。

/usr/include へのインストール

```
tar xzf eigen-3.3.9.tar.gz
cd eigen-3.3.9
tar cf - Eigen | (cd /usr/local/include; sudo tar xzf -)
```

6.3.2 ball.cpp

```
/*
 * ball.cpp --- はねるボール
 *   http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/ball.cpp
 *   cc -I /usr/local/include ball.cpp
 *   ./a.out > ball.data
 *   gnuplot で plot "ball.data" with lp
 */

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

[21] http://eigen.tuxfamily.org/

```
#include <Eigen/Dense>
using namespace Eigen;

double m, g, Gamma, e;

VectorXd f(double t, VectorXd x)
{
  VectorXd y(4);
  y(0) = x(2);
  y(1) = x(3);
  y(2) = - Gamma / m * x(2);
  y(3) = - g - Gamma / m * x(3);
  return y;
}

int main(void)
{
  int n, N;
  double tau, Tmax, t,pi;
  VectorXd x(4),k1(4),k2(4),k3(4),k4(4);

  pi = 4 * atan(1.0);
  m = 100;
  g = 9.8;
  Gamma = 1.0;
  e = 1.0;

  Tmax = 20;
  N = 1000;
  tau = Tmax / N;
  x << 0,0,50*cos(pi*50/180),50*sin(pi*50/180);
  for (n = 0; n < N; n++) {
    t = n * tau;
    k1 = tau * f(t, x);
    k2 = tau * f(t+tau/2, x+k1/2);
    k3 = tau * f(t+tau/2, x+k2/2);
    k4 = tau * f(t+tau, x+k3);
    x = x + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    if (x(1)<0) {
      x(1) = - x(1);
      x(3) = - x(3);
    }
    std::cout << x(0) << " " << x(1) << std::endl;
  }
}
```

コンパイル&実行

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/ball.cpp
c++ -O -I /usr/local/include ball.cpp
./a.out > ball.data
```

gnuplot で描画

```
gnuplot> plot "ball.data" with lp
```

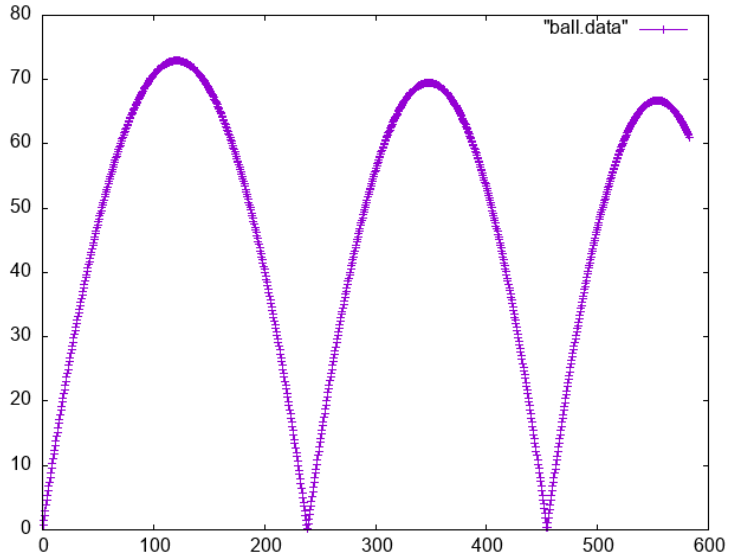

図 12: 弾むボール

7　参考文献

この文書のオリジナルが書かれたときにはなかったが、今では、常微分方程式の数値解法の有名な専門書の和訳 (ハイラー・ネルセット・ヴァンナー [1], ハイラー・ヴァンナー [2]) が出版されている。

よりコンパクトな解説としては、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] あるいは [1], [2] 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 [5], 戸川 [6] も奨めたい。特に戸川 [6] 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 [7] も読むに価する。

参考文献

[1] E. ハイラー, S. P. ネルセット, G. ヴァンナー：常微分方程式の数値解法 I 基礎編, シュプリンガー・ジャパン (2007), 三井斌友監訳.

[2] E. ハイラー, G. ヴァンナー：常微分方程式の数値解法 II 発展編, シュプリンガー・ジャパン (2008), 三井斌友監訳.

[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：常微分方程式の数値解法, 岩波書店 (2003), 「微分方程式の数値解法 I」岩波講座応用数学 (1993) の単行本化.

[5] 一松信 (ひとつまつしん)：数値解析, 朝倉書店 (1982/10).

[6] 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基礎篇 C 言語版, 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たけれど、今はない？

[7] 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 (1989 (1982?), 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。