常微分方程式の初期値問題を解くプログラムの書き方

桂田 祐史

2021 年 4 月 23 日, 2023 年 6 月 26 日

http://nalab.mind.meiji.ac.jp/~mk/labo/text/intro-ode-simulation/

目 次

0　「はじめに」の前に — Euler 法は使わないで

常微分方程式の初期値問題の解を Euler 法で計算する人が後を絶たない (うわぁ)。そうなっているのは、説明の仕方がまずいのだろう。かっこうは悪いけれど、最初に注意しておくことにしました。

この文書は、常微分方程式の初期値問題の数値シミュレーションを体験するための案内をする、という目的で用意しました。数値計算法としては、「定番」の (古典的, 4 次の) Runge-Kutta 法を選択しました。なぜそれが定番で、そこから始めるのが良いか、というのは後述します (予定…)。

Runge-Kutta 法の利用を勧めるのに、なぜ Euler 法の説明をするかというと、それは (離散変数法の) 原理を理解するには役立つだろう、偏微分方程式のシミュレーションなどでは Euler 法 (普通は後退 Euler 法だけど) は普通に採用されているから、頭の片隅に入れておいてもらうと良いだろう、と考えたからです。同じように考えているかどうかは分かりませんが、実際に Euler 法と Runge-Kutta 法の解説をしてお終い、というテキストはとても多い。(Euler 法と Runge-Kutta 法の「中間」の Heun 法の解説をしてあるテキストもあったりしますが、多分「原理を理解してもらう」という目的だと思います。)

一方で、最初から高性能な方法を紹介して勧める、という本もなくはないです (最近手に取った小川・宮路 [1] はそういうノリでした)。実際に Mathematica や MATLAB に “お任せ” していると、(利用者本人が知らないうちに) そういう方法を使うことになるので、「伝統的な」教え方を見直すときに来ているのかな、とも考えています。G 節はその考えに基づいて書き始めました。

1　はじめに

1.1　この文書の目的

ずっと以前 (20 世紀 — 笑)、常微分方程式の数値シミュレーションの方法について、桂田 [2] という解説を書いた。現実のコンピューター, ソフトウェアが出て来るので、今となっては、さすがに古くなってしまっている。

この文書はそのリニューアルである、という面がある。([2] は授業資料が元になっていて、この文書はゼミで使用することを念頭に書かれているので、趣きが違っている、ということはある。)

[2] では、C 言語プログラムの例を紹介したが、図を描くために GLSC というライブラリィを用いていて、今となってはそれに慣れていない人も多い (そもそも 2021 年度の 3 年生の Mac には、GLSC がインストールされていなかった)。

現在の私は、C 言語でプログラムを書くことを推奨する気はほとんどなく、この文書でも、C++ や Julia の例を載せてある。しかし、学生は C 言語に慣れている人も (が) 多いので、C 言語で数値計算して可視化には gnuplot を用いる例を、なるべく多く載せることにした。

(2021/9/11 追記) 自分は使わない気もするけれど、Python のことも書いておくべきかも。試してみたことのメモを「Python で ode」[1] に書いた。

[1] http://nalab.mind.meiji.ac.jp/~mk/knowhow-2021/node26.html

1.2 常微分方程式の数値計算に親しもう

モデリング、アナリシス、シミュレーションを三本柱とする現象数理学を学習・研究する者にとって、微分方程式のシミュレーションを行う技術は基本的である。

幸い常微分方程式の初期値問題に限れば、比較的簡単な方法で、非常に広い範囲の問題に応用できる。研究テーマを選ぶ際にも選択肢が多くなるということである。

ここでは有名な常微分方程式をいくつか集めて分類し、例題を選んでそれに対するサンプル・プログラムを示すことによって、計算の仕方を説明する。

出て来る微分方程式そのものについては、ほとんど説明しない。有名なものが多く、読者が既に知っている可能性も高いし、わざわざここに中途半端な説明を書くよりも、他のしっかりした資料を当たってもらうのが良い、と考えるからである。知らない人は、良い機会だから自分で調べてみることをお勧めする。探してみたけれど、適当な資料が見つからない、というときは、質問して下さい。

1.3 参考情報

数値計算のやり方について、既に述べたように (さすがに賞味期限切れのような気がして来たけれど、全面的に書き直すまでは)、桂田 [2] をあげておく。

この文書は、とにかく数値計算できるようになることを目標にしているが、使っている数値解法の理論的側面の説明は省略している。それを学びたい場合、入門的部分は桂田 [3] で読める (そこから先は、[3] で紹介してある専門書を見て下さい)。

常微分方程式の入門的部分は、授業で習っているはずだが、桂田 [4], [5] で読める (後者は内輪向けでパスワード付きです)。

C 言語については記憶があいまい、と言う人が珍しくないけれど、使うことは多くないので、必要なことをその都度復習すれば大丈夫。桂田 [6] の 3 節 (ポインターのところは飛ばして構わない) で十分である。

2 Euler 法、Runge-Kutta 法入門 — 1 次元の問題

2.1 説明用の問題　マルサスの法則

$k, c \in \mathbb{R}$ とする。常微分方程式

$$\frac{dx}{dt} = kx \tag{1}$$

に初期条件

$$x(0) = c \tag{2}$$

を課した初期値問題の解は

$$x(t) = ce^{kt}.$$

この微分方程式は、人口論のマルサスの法則 (その場合 $k > 0$)[2], 放射性元素の崩壊 (その場合 $k < 0$), 冷却など現実の現象のモデルとしても登場する。

[2] T. R. Malthus (1766—1834) は英国の経済学者で、1798 年に『人口の原理』を著わし、人口は幾何級数的 (≒等比数列的=指数関数的) に増加しうるが、生存手段は算術級数的 (=等差数列的= 1 次関数的) にしか増加しないと論じた。これを書いている 2021 年は COVID19 が流行している。感染者は指数関数的に増加するが、病床は…、だから感染を抑えることが重要である、ということを理解できない人が少なくない。そういえばウィキペディアのマルサスの項目を見ても、マルサスが何度も書いて強調していることが載っていない。

2.2 離散変数法による近似解

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 variablemethod) 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[3]。分点は、特に理由がなければ N 等分点にとる。すなわち $h = (b-a)/N$ として $t_j = a + jh$ $(j = 0, 1, 2, \cdots, N)$ とする。(問題によっては、刻み幅を調節するのが望ましい場合も多い。それについては…どうしようかな。)

2.2.1 前進 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

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

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 **Euler 法**である[4]。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 (精度をあげようとすると計算量が非常に大きくなる)、実際に使われることはまれである。

2.2.2 後退 Euler 法

(準備中)

---

[3] 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」なわけである。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。

[4] **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**とも呼ばれる。

2.2.3 Runge-Kutta 法

漸化式

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

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[5]。

(k_1, k_2, k_3, k_4 は j に依存するので、本来は例えば $k_{1,j}$, $k_{2,j}$, $k_{3,j}$, $k_{4,j}$ のように、j を添えて書くべきであるが、j を添えるのは省略されることが多い。この辺はプログラミングのノリかもしれない。)

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、

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述する (つもり…約束はしない)。

[5] 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。

2.3 C 言語のプログラム例

2.3.1 Euler 法の C プログラム例

euler1ex1.c

```
/*
 * euler1ex1.c (Euler method for Malthusian model)
 */

#include <stdio.h>

double k = 1.0;

int main(void)
{
  int i, N;
  double t, x, dt;
  double f(double, double), x0;
  double Tmax;
  // 初期値
  x0 = 1.0;
  // 最終時刻
  Tmax = 1.0;
  // 時間刻み
  printf("# N: "); scanf("%d", &N);
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  printf("# t x\n");
  printf("%f %f\n", t, x);
  // Euler 法
  for (i = 0; i < N; i++) {
    x += dt * f(x, t);
    t = (i + 1) * dt;
    printf("%f %f\n", t, x);
  }
  return 0;
}

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

cc でコンパイル&実行

```
% cc euler1ex1.c
% ./a.out
# N: 100
# t x
0.000000 1.000000
中略
1.000000 2.704814
%
```

cglsc でコンパイル&実行

```
% cglsc euler1ex1.c
% ./euler1ex1
# N: 100
(以下略)
%
```

解曲線を描いてみよう。この場合は gnuplot が簡単である。

解曲線を描く

```
% cc euler1ex1.c
% ./a.out > euler1ex1.data
100
% gnuplot
(ここで gnuplot の起動メッセージが表示されるが省略)
gnuplot> plot "euler1ex1.data" with lp
(これでグラフが表示される。以下は画像の保存。)
gnuplot> set term png
gnuplot> set output "euler1ex1.png"
gnuplot> replot
gnuplot> quit
%
```

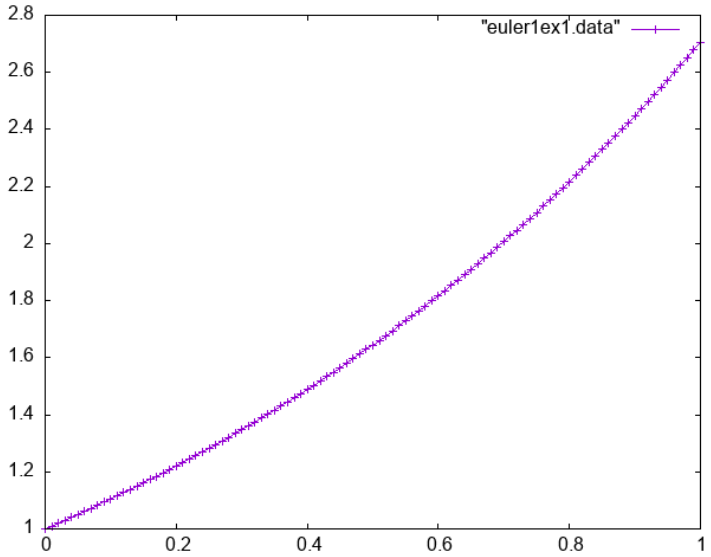


図 1: $dx/dt = x$, $x(0) = 1$ に対する Euler 法の解曲線

gnuplot については、ネットに解説があふれている (例えば桂田 [7])。この問題について役に立ちそうなことをいくつか説明しておく。

gnuplot の細かな工夫に関する注意

- gnuplot の機能 (数値データをファイルから読んでプロットする) を用いている。`euler1ex1.data` には、`# N:` と `# t x` という、数値データでない行が含まれている。これらの行の先頭に `#` を入れておくことにより、gnuplot にとっては注釈行になっている。

- gnuplot で縦軸の範囲を $0 \leq x \leq 3$ に指定したければ、`plot [] [0:3] "euler1ex1.data" with lp` とすればよい。
- `with lp` は `with linespoints` の略で、線分でつなぎ、マーカーを描くことの指定である。線分でつなぐだけなら `with l` (`with lines` の略) で良い。
- gnuplot へのコマンドの入力が面倒ならば、プログラムにしてしまうという手がある。

test1.gp

```
plot "euler1ex1.data" with lp
set term png
set output "euler1ex1.png"
replot
```

というファイルを作っておけば

```
% gnuplot test1.gp
```

で実行できる。どうやって (どのデータを用いて) 画像ファイルを作ったかの記録になるし、再現もしやすいので、レポートや論文のためのデータを作る場合はこうするのがお勧めである。

問 1　上の実行例で最後に出力した数値 `2.704814` の誤差はいくらか。N を変えることで誤差がどのように変わるか調べよ (両側対数目盛でプロットすることを勧める)。

問 2　上の実行例では、出力をリダイレクトすることでデータファイルを作っているが、`fopen()`, `fclose()`, `fprintf()` を使ってデータファイルを作れ (参考 「簡単なファイル入出力」[6], 解答は付録 F.1 を見よ)。

問 3　C のプログラムの中で `popen()` という関数を使うと、外部プログラムを起動して通信ができる。gnuplot を起動して解曲線を描く C プログラムを作れ。(解答は付録 F.2 を見よ)。

問 4　現象数理学科 Mac では、`cglsc` コマンドでコンパイルすることによって、GLSC というグラフィックス・ライブラリィが利用できる。それを用いて解曲線を描く C プログラムを作れ。

複数の初期値についての解を表示してみよう。gnuplot に数値データを読ませているとき、`e` 1 文字からなる行と空行 (行の最初で改行する) があると、データの区切りになり、その後は別のプロットをすることになる。

[6] `http://nalab.mind.meiji.ac.jp/~mk/labo/text/nantoka-c++/node10.html`

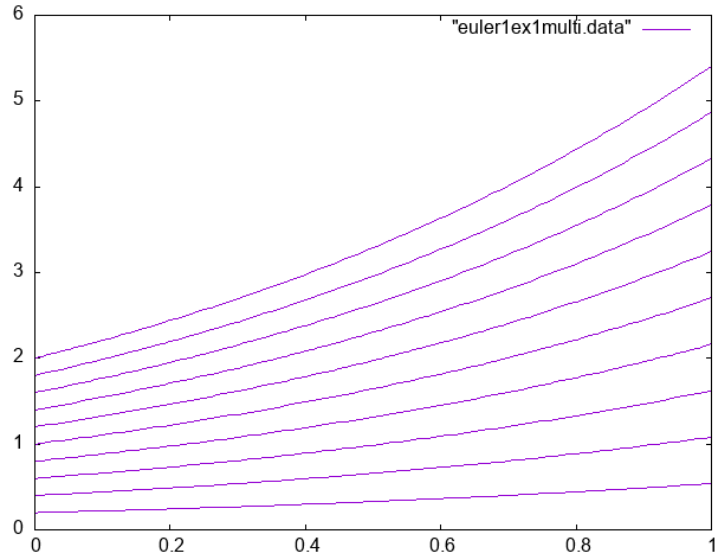


図 2: $dx/dt = x$, $x(0) = 1$ に対する Euler 法の解曲線 ($x(0)$ を 0.2 刻みで変化)

euler1ex1multi.c

```
/*
 * euler1ex1multi.c (Euler method for Malthusian model)
 *   複数の初期条件に対応する解を計算する
 */

#include <stdio.h>

double k = 1.0;

int main(void)
{
  int i, N;
  double t, x, dt;
  double f(double, double), x0;
  double Tmax;
  // 最終時刻
  Tmax = 1.0;
  // 時間刻み
  printf("# N: "); scanf("%d", &N);
  dt = Tmax / N;
  // 初期値 x(0)=x0 を 0.2 から 2.0 まで 0.2 刻みで変更する
  for (x0 = 0.2; x0 <= 2.0; x0 += 0.2) {
    // x(0)=x0 を初期条件とする。
    t = 0.0;
    x = x0;
    printf("# t x\n");
    printf("%f %f\n", t, x);
    // Euler 法
    for (i = 0; i < N; i++) {
      x += dt * f(x, t);
      t = (i + 1) * dt;
      printf("%f %f\n", t, x);
    }
    printf("e\n\n"); // e 1 文字の行と空行
  }
  return 0;
}

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

2.3.2 Runge-Kutta 法のプログラム例

Euler 法のプログラムを Runge-Kutta 法を用いるように書き換えるのは簡単である。

rk1ex1.c

```
/*
 * rk1ex1.c (Runge-Kutta method for Malthusian model)
 */

#include <stdio.h>

double k = 1.0;

int main(void)
{
  int i, N;
  double t, x, dt, k1, k2, k3, k4;
  double f(double, double), x0;
  double Tmax;
  // 初期値
  x0 = 1.0;
  // 最終時刻
  Tmax = 1.0;
  // 時間刻み
  printf("# N: "); scanf("%d", &N);
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  printf("# t x\n");
  printf("%f %f\n", t, x);
  // Runge-Kutta 法
  for (i = 0; i < N; i++) {
    k1 = dt * f(x, t);
    k2 = dt * f(x + k1/2, t + dt / 2);
    k3 = dt * f(x + k2/2, t + dt / 2);
    k4 = dt * f(x + k3, t + dt);
    x += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    t = (i + 1) * dt;
    printf("%f %20.14f\n", t, x);
  }
  return 0;
}

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

cc でコンパイル&実行

```
% cc rk1ex1.c
% ./a.out
# N: 100
# t x
0.000000 1.000000
中略
1.000000 2.718282
%
```

解曲線の図を描くための手順は Euler 法のときと同様 (データファイルの名前を変えるくらい) なので省略する。

問 5　2.718282 は小数点以下 6 位までしか表示していないが、小数点以下 15 位まで表示するように C プログラムを書き直して実行せよ (`printf()` の書式の選び方の問題, 「浮動小数点数の入出力と四則演算」[7])。そうすると 2.718281828234403 となる。この 2.718281828234403 の誤差はいくらか。N を変えることで誤差がどのように変わるか調べよ (両側対数目盛でプロットすることを勧める, 「対数グラフを描く」[8])。

2.4　課題

2.4.1　ロジスティック方程式 (logistic equation)

ロジスティック方程式の初期値問題

$$\frac{dx}{dt} = (k - \lambda x)x \tag{8}$$

$$x(0) = c \tag{9}$$

(ここで k, λ, c は正の定数) を解くプログラムを作成し、解曲線を描け。複数の解曲線 (初期値を変える) を同時に描いてみよ。ロジスティック方程式について資料を探して調べてみよ。

$c \neq \frac{k}{\lambda}$ のとき、解は次式で与えられる。

$$x(t) = \frac{kc}{\lambda c + (k - \lambda c)e^{-kt}}. \tag{10}$$

($c = \frac{k}{\lambda}$ のときは、$x(t) = \frac{k}{\lambda}$ である。)

以前、資料調べをしたことがあった (原典中心なので、現代的な解説は微分方程式の教科書的な本を当たると良いだろう)。

> 「マルサスの法則, フェルフルストのロジスティック方程式, ロトカ・ヴォルテラの方程式」[9]

平衡点やその安定性を調べることを勧める。(用語の定義は、付録 D, D.2, D.3 を見よ。そこに載っている定理で安定性の判定が可能であるが、この場合は解が得られているので、それから安定性を判定することもできる。)

3　2 次元の問題

3.1　ターゲット問題 van der Pol の方程式

$$\frac{dx}{dt} = y, \tag{11}$$

$$\frac{dy}{dt} = -x + \mu(1 - x^2)y \tag{12}$$

$$x(0) = x_0, \quad y(0) = y_0 \tag{13}$$

μ は正定数。

[7] http://nalab.mind.meiji.ac.jp/~mk/labo/text/cminimum/node10.html
[8] http://nalab.mind.meiji.ac.jp/~mk/labo/howto/intro-gnuplot/node8.html
[9] http://nalab.mind.meiji.ac.jp/~mk/labo/library/lotka-volterra/malthus-verhulst-lotka-volterra/

3.2 Euler 法, Runge-Kutta 法はベクトル値関数でも OK

Euler 法の場合は

$$\vec{x}_{i+1} = \vec{x}_i + \Delta t \vec{f}(\vec{x}_i, t_i). \tag{14}$$

もしベクトルを扱えるプログラミング言語ならば、この形のままでプログラムが書ける。

そうでない場合もこの式を成分表示した

$$x_{i+1} = x_j + \Delta t f_x(x_i, y_i, t_i) \tag{15a}$$

$$y_{i+1} = y_j + \Delta t f_y(x_i, y_i, t_i) \tag{15b}$$

を使えば良い。(ここで f_x, f_y は $\vec{f}$ の第 1 成分 (x 成分)、第 2 成分 (y 成分) を表す。偏微分ではない。)

実際には、x, y は配列 `x[]`, `y[]` ではなく、スカラー変数 `x`, `y` に記憶することが多いであろうから、次のどちらかになるだろうか？[10]

```
dx = dt * fx(x,y,t);
dy = dt * fy(x,y,t);
x += dx;
y += dy;
```

```
newx = x + dt * fx(x,y,t);
newy = y + dt * fy(x,y,t);
x = newx;
y = newy;
```

(要点は `x`, `y` の更新は、f の値の計算が終わってから、ということである。)

Runge-Kutta 法の場合は

$$\vec{k}_1 = \Delta t \vec{f}(\vec{x}_i, t_i), \tag{16a}$$

$$\vec{k}_2 = \Delta t \vec{f}(\vec{x}_i + \vec{k}_1/2, t_i + \Delta t/2), \tag{16b}$$

$$\vec{k}_3 = \Delta t \vec{f}(\vec{x}_i + \vec{k}_2/2, t_i + \Delta t/2), \tag{16c}$$

$$\vec{k}_4 = \Delta t \vec{f}(\vec{x}_i + \vec{k}_3, t_i + \Delta t) \tag{16d}$$

で $\vec{k}_1$, $\vec{k}_2$, $\vec{k}_3$, $\vec{k}_4$ を計算して[11]

$$\vec{x}_{i+1} = \vec{x}_i + \frac{1}{6}\left(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4\right) \tag{17}$$

とすれば良い。やはりベクトルを扱えるプログラミング言語ならば、この形のままでプログラムが書ける。

[10] もっと効率的に出来なくはないけれど、ここでは分かりやすさ優先のコードにした。

[11] $\vec{k}_1$, $\vec{k}_2$, $\vec{k}_3$, $\vec{k}_4$ は i に依存するので、本来は $\vec{k}_{1,i}$ のように i を含める記号を用いるべきだがサボっている。

ベクトルが扱えないプログラミング言語ならば、(16a)–(16d) を成分表示した

$$k_{1,x} = \Delta t \vec{f}_x(x_i, y_i, t_i), \tag{18a}$$
$$k_{1,y} = \Delta t \vec{f}_y(x_i, y_i, t_i), \tag{18b}$$
$$k_{2,x} = \Delta t \vec{f}_x(x_i + k_{1,x}/2, y_i + k_{1,y}/2, t_i + \Delta t/2), \tag{18c}$$
$$k_{2,y} = \Delta t \vec{f}_y(x_i + k_{1,y}/2, y_i + k_{1,y}/2, t_i + \Delta t/2), \tag{18d}$$
$$k_{3,x} = \Delta t \vec{f}_x(x_i + k_{2,x}/2, y_i + k_{2,y}/2, t_i + \Delta t/2), \tag{18e}$$
$$k_{3,y} = \Delta t \vec{f}_y(x_i + k_{2,y}/2, y_i + k_{2,y}/2, t_i + \Delta t/2), \tag{18f}$$
$$k_{4,x} = \Delta t \vec{f}_x(x_i + k_{3,x}/2, y_i + k_{3,y}/2, t_i + \Delta t/2), \tag{18g}$$
$$k_{4,y} = \Delta t \vec{f}_y(x_i + k_{3,y}/2, y_i + k_{3,y}/2, t_i + \Delta t/2), \tag{18h}$$

で $k_{1,x}$, $k_{1,y}$, $k_{2,x}$, $k_{2,y}$, $k_{3,x}$, $k_{3,y}$, $k_{4,x}$, $k_{4,y}$ を計算して

$$x_{i+1} = x_i + \frac{1}{6}\left(k_{1,x} + 2k_{2,x} + 2k_{3,x} + k_{4,x}\right), \tag{19a}$$
$$y_{i+1} = y_i + \frac{1}{6}\left(k_{1,y} + 2k_{2,y} + 2k_{3,y} + k_{4,y}\right) \tag{19b}$$

とすれば良い。— これはやっていることはベクトルを成分表示しただけなのだが、私の経験によると間違える人がとても多い。間違えることを見込んで、対応するための時間を授業で用意したこともあるが、今は「そういうプログラミング言語を使うのは拙い、プログラミング言語を換えるべき」と考えている。

この例では $\vec{f}$ が 2 次元であるから、(18a)–(18h) という 8 つの式で済んでいるが、ベクトル $\vec{x}$ の次元 n が高くなると面倒になる。その場合は、$\vec{x}$, $\vec{k}_1$, $\vec{k}_2$, $\vec{k}_3$, $\vec{k}_4$ などを配列で取り扱うプログラムが多い (この文書ではそれは説明しない)。

3.3 C 言語によるプログラム例

3.3.1 Euler 法の C プログラム例

euler2ex1.c

```
/*
 * euler2ex1.c (Euler method for van der Pol equation)
 */

#include <stdio.h>

double mu = 1.0;

int main(void)
{
  int i, N;
  double t, x, y, dt, dx, dy;
  double fx(double, double, double), fy(double, double, double), x0, y0;
  double Tmax;
  // 初期値
  x0 = 0.1; y0 = 0.1;
  // 最終時刻
  Tmax = 50.0;
  // 時間刻み
  printf("# N: "); scanf("%d", &N);
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  y = y0;
  printf("# t x y\n");
  printf("%f %f %f\n", t, x, y);
  // Euler 法
  for (i = 0; i < N; i++) {
    dx = dt * fx(x, y, t);
    dy = dt * fy(x, y, t);
    x += dx;
    y += dy;
    t = (i + 1) * dt;
    printf("%f %f %f\n", t, x, y);
  }
  return 0;
}

double fx(double x, double y, double t)
{
  return y;
}

double fy(double x, double y, double t)
{
  return - x + mu * (1.0- x * x) * y;
}
```

解曲線を描く

```
% cglsc euler2ex1.c
% ./euler2ex1 > euler2ex1.data
1000
% gnuplot
(ここで gnuplot の起動メッセージが表示されるが省略)
gnuplot> splot "euler2ex1.data" with l
(以下はグラフの保存)
gnuplot> set term png
gnuplot> set output "euler2ex1_txy.png"
gnuplot> replot
gnuplot> quit
%
```

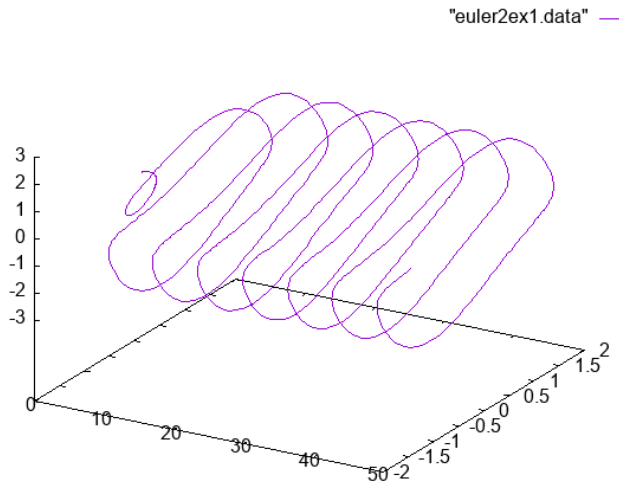


図 3: van der Pol 方程式の解曲線

同じデータファイル `euler2ex1.data` を使って、色々なグラフが描ける。gnuplot の `using a:b` (a 列目と b 列目のデータを使う) という指示を利用する。

x, y の時間変化を描く

```
% gnuplot
(ここで gnuplot の起動メッセージが表示されるが省略)
gnuplot>  plot "euler2ex1.data" with l, "euler2ex1.data" using 1:3 with l
(以下はグラフの保存)
gnuplot> set term png
gnuplot> set output "euler2ex1_tx_ty.png"
gnuplot> replot
gnuplot> quit
%
```

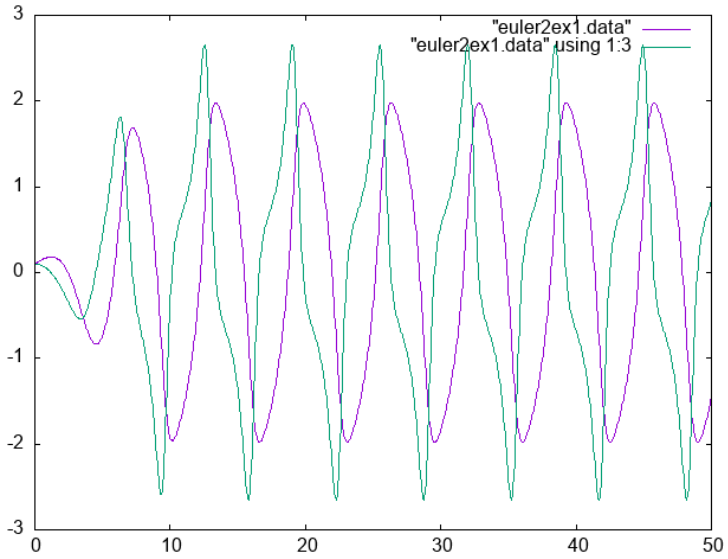


図 4: van der Pol 方程式の x, y の時間変化 (カラーでないと分かりにくい)

解軌道を描く

```
% gnuplot
(ここで gnuplot の起動メッセージが表示されるが省略)
gnuplot>  plot "euler2ex1.data" using 2:3 with l
(以下はグラフの保存)
gnuplot> set term png
gnuplot> set output "euler2ex1_xy.png"
gnuplot> replot
gnuplot> quit
%
```

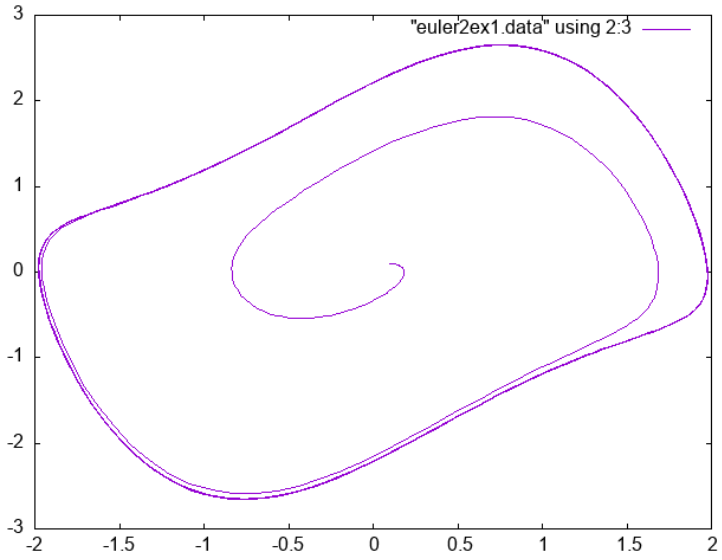


図 5: van der Pol 方程式の解軌道

問 6　上の実行例では、N (区間をいくつの小区間に分割するか) を 1000 にしたが、N が 100 のときどうなるか試してみよ (数値的安定性の話)。

3.3.2 Runge-Kutta 法の C プログラム

Euler 法のプログラムをたたき台にして Runge-Kutta 法のプログラムを作る。2 次元であるから、それほど複雑ではない (しかしエラーなしでやるのは結構難しい)。

rk2ex1.c

```
/*
 * rk2ex1.c (Runge-Kutta method for van der Pol equation)
 */

#include <stdio.h>

double mu = 1.0;

int main(void)
{
  int i, N;
  double t, x, y, dt, k1x, k1y, k2x, k2y, k3x, k3y, k4x, k4y;
  double fx(double, double, double), fy(double, double, double), x0, y0;
  double Tmax;
  // 初期値
  x0 = 0.1; y0 = 0.1;
  // 最終時刻
  Tmax = 50.0;
  // 時間刻み
  printf("# N: "); scanf("%d", &N);
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  y = y0;
  printf("# t x y\n");
  printf("%f %f %f\n", t, x, y);
  // Runge-Kutta 法
  for (i = 0; i < N; i++) {
    k1x = dt * fx(x, y, t);
    k1y = dt * fy(x, y, t);
    k2x = dt * fx(x + k1x / 2, y + k1y / 2, t + dt / 2);
    k2y = dt * fy(x + k1x / 2, y + k1y / 2, t + dt / 2);
    k3x = dt * fx(x + k2x / 2, y + k2y / 2, t + dt / 2);
    k3y = dt * fy(x + k2x / 2, y + k2y / 2, t + dt / 2);
    k4x = dt * fx(x + k3x, y + k3y, t + dt);
    k4y = dt * fy(x + k3x, y + k3y, t + dt);
    x += (k1x + 2 * k2x + 2 * k3x + k4x) / 6;
    y += (k1y + 2 * k2y + 2 * k3y + k4y) / 6;
    t = (i + 1) * dt;
    printf("%f %f %f\n", t, x, y);
  }
  return 0;
}

double fx(double x, double y, double t)
{
  return y;
}

double fy(double x, double y, double t)
{
  return - x + mu * (1.0 - x * x) * y;
}
```

使い方 (コンパイル、実行、可視化) は、Euler 法のプログラム euler2ex1.c と同じである。

rk2ex1.data という名前のファイルにデータを記録したとして

test4a.gp

```
splot "rk2ex1.data" with l
set term png
set output "rk2ex1_txy.png"
replot
```

test4b.gp

```
plot "rk2ex1.data" with l, "rk2ex1.data" using 1:3 with l
set term png
set output "rk2ex1_tx_ty.png"
replot
```

test4c.gp

```
plot "rk2ex1.data" using 2:3 with l
set term png
set output "rk2ex1_xy.png"
replot
```

3.4 C++言語 & Eigen によるプログラム例

C++ は C 言語と似ているので、C 言語を知っている人には、敷居が低いであろう (Mac で、C 言語のコンパイラーを使えるようにしてある人は、同時に C++ のコンパイラーも使えるようになっているはず)。便利なライブラリィが豊富にある。ベクトル・行列計算のためには、Eigen というライブラリィがおすすめ。

Eigen のインストールは簡単 (A を見よ。5 分程度で済むので、うまく行かなければ気軽に質問して下さい。)。

C++は、入出力が C 言語とは違うが、それについては、「なんとか C++ を使う」[12] を見ると良い。

Eigen には、VectorXd というベクトルを扱うクラスがある。

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

  VectorXd(2) x;
```

とすると、2 次元ベクトル x が定義できる。ベクトル x の成分 x(0), x(1) が使えるようになる。これだけだと配列と変わらないが、ベクトル演算ができるのが大きな利点である。ここでは 2 次元であるが、何次元でも

```
    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 += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
```

により Runge-Kutta 法の計算ができる。

[12] http://nalab.mind.meiji.ac.jp/~mk/labo/text/nantoka-c++/

rk2ex1++.cpp

```
/*
 * rk2ex1++.cpp (Runge-Kutta method for van der Pol equation)
 *   c++ -I /usr/local/include rk2ex1++.cpp
 *   ./a.out > rk2ex1.data （分割数 N を入力．例えば 1000）
 *   gnuplot で
 *    splot "rk2ex1.data" with l
 *    plot "rk2ex1.data" with l, "rk2ex1.data" using 1:3 with l
 *    plot "rk2ex1.data" using 2:3 with l
 */

#include <iostream>
#include <math.h>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;

double mu = 1.0;

int main(void)
{
  int i, N;
  double t, dt, pi;
  VectorXd x(2),k1(2),k2(2),k3(2),k4(2),f(double,VectorXd),x0(2);
  double Tmax;
  // 最終時刻
  Tmax = 50.0;
  // 時間刻み
  cout << "# N: "; cin >> N;
  dt = Tmax / N;
  // 初期値
  x0 << 0.1,0.1;
  t = 0.0;
  x = x0;
  cout << "# t x" << endl;
  cout << fixed; // この後は C 言語の %f の形式で出力
  cout << t << x(0) << x(1) << endl;
  // Runge-Kutta 法
  for (i = 0; i < N; i++) {
    k1 = dt * f(t, x);
    k2 = dt * f(t+dt/2, x+k1/2);
    k3 = dt * f(t+dt/2, x+k2/2);
    k4 = dt * f(t+dt, x+k3);
    x = x + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    t = (i + 1) * dt;
    cout << t << " " << x(0) << " " << x(1) << endl;
  }
}

VectorXd f(double t, VectorXd x)
{
  VectorXd y(2);
  y(0) = x(1);
  y(1) = - x(0) + mu * (1.0 - x(0) * x(0)) * x(1);
  return y;
}
```

コンパイル&実行

```
% c++ -O3 -I /usr/local/include -o rk2ex1 rk2ex1++.cpp
% ./rk2ex1 > rk2ex1++.data
1000
%
```

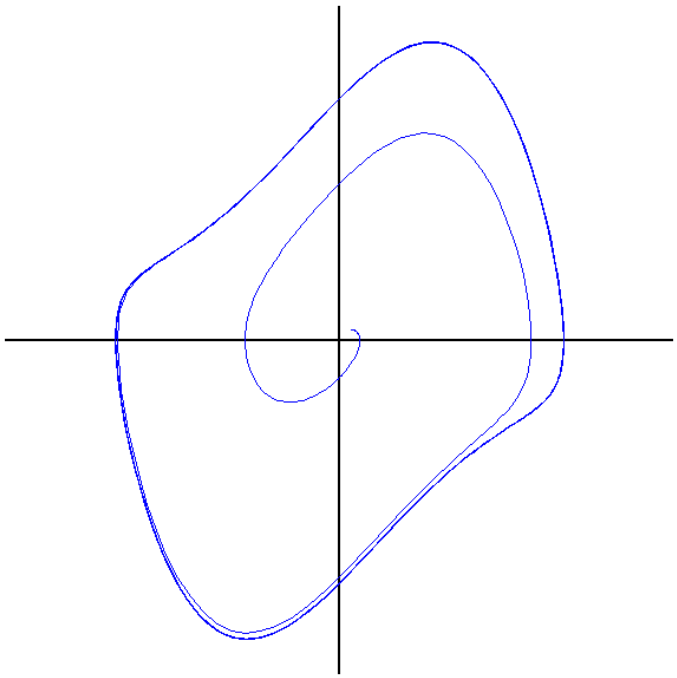
この後は、これまでと同様にして、gnuplot を用いて可視化ができる。

ところで、GLSC (付録 C を見よ) を用いると、C++のプログラムからグラフィックスが使える。以下のプログラム中で、名前が g_で始まっている関数が、GLSC の関数である。初期化や座標系の設定などはごちゃごちゃしているが、グラフを描く部分は g_move(x,y), g_plot(x,y) の 2 命令しか使っていないので、難しくはないだろう。

rk2ex1++glsc.cpp

```
/*
 * rk2ex1++glsc.cpp (Runge-Kutta method for van der Pol equation)
 *   c++ -I /usr/local/include -I/opt/X11/include -o rk2ex1++glsc \
 *       rk2ex1++glsc.cpp -L/usr/local/lib -lglscd -L/opt/X11/lib -lX11 -lm
 *   ./rk2ex1++glsc  (分割数 N を入力. 例えば 1000)
 */

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

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

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

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

double mu = 1.0;

int main(void)
{
  int i, N;
  double t, dt, pi;
  VectorXd x(2),k1(2),k2(2),k3(2),k4(2),f(double,VectorXd),x0(2);
  double Tmax;
  double a = -3.0, b = 3.0, c = -3.0, d = 3.0;
  double win_width, win_height, w_margin, h_margin;
  // 最終時刻
  Tmax = 50.0;
  // 時間刻み
  cout << "# N: "; cin >> N;
  dt = Tmax / N;
  // 初期値
  x0 << 0.1,0.1;
  t = 0.0;
  x = x0;
  cout << "# t x" << endl;
  cout << fixed; // この後は C 言語の %f の形式で出力
  cout << t << " " << x(0) << " " << x(1) << endl;
  // GLSC の初期化、座標系の設定
  win_width = 150.0; win_height = 150.0; w_margin = 10.0; h_margin = 10.0;
  g_init((char *)"LOTKA_VOLTERRA",
         win_width  + 2 * w_margin, win_height + 2 * h_margin);
  g_device(G_BOTH);
  g_def_scale(0, a, b, c, d,
      w_margin, h_margin, win_width, win_height);
  g_sel_scale(0);
  // 線種の定義
  g_def_line(0, G_BLACK, 2, G_LINE_SOLID);
  g_def_line(1, G_BLUE,  1, G_LINE_SOLID);
  /* 座標軸を描く */
  g_sel_line(0);
  g_move(a, 0.0);  g_plot(b, 0.0);
  g_move(0.0, c);  g_plot(0.0, d);
  // ここからグラフ描き
  g_sel_line(1);
  g_move(x(0), x(1));
  // Runge-Kutta 法
  for (i = 0; i < N; i++) {
```

このプログラムを実行すると、画面にウィンドウが現れて、そこに解軌道が描かれる。計算終了後にそのウィンドウをクリックすると、画像をファイル (`rk2ex1++glsc.png`) に保存してからプログラムが終了する。



■

図 6: van der Pol 方程式の解軌道

3.5 Julia 言語によるプログラム例

Julia は 2012 年に公開された新しいプログラミング言語で、色々な特徴があるが、特に数値計算に向いているとされている。

Julia の処理系のインストールの仕方は、付録 B を見よ。

参考のため 1 次元の問題の Julia によるプログラムを掲げる。

rk1ex1.jl

```
# rk1ex1.jl --- dx/dt=x (0<t<1), x(0)=1 を Runge-Kutta 法で解く

using Printf

function testrungekutta(Tmax=1.0,N=10)
  # 初期値
  t0=0.0
  x0=1.0
  #
  @printf("#N=%d Tmax=%f\n", N, Tmax)
  @printf("# t x\n")
  # Runge-Kutta 法
  dt=Tmax/N
  t=t0
  x=x0
  for j=1:N
    k1=dt*f(t,x)
    k2=dt*f(t+dt/2, x+k1/2)
    k3=dt*f(t+dt/2, x+k2/2)
    k4=dt*f(t+dt, x+k3)
    x += (k1+2*k2+2*k3+k4) / 6
    t = j * dt
    @printf("%.4f %.7f\n", t, x)
  end
end

function f(t,x)
  x
end

# julia rk1ex1.jl と実行した場合に testrungekutta() を実行する。
if abspath(PROGRAM_FILE) == @__FILE__
    testrungekutta()
end
```

実行の仕方 1

```
% julia rk1ex1.jl


#N=10 Tmax=1.000000
# t x
0.1000 1.1051708
0.2000 1.2214026
0.3000 1.3498585
0.4000 1.4918242
0.5000 1.6487206
0.6000 1.8221180
0.7000 2.0137516
0.8000 2.2255396
0.9000 2.4596014
1.0000 2.7182797
%
```

実行の仕方 2

```
% julia

julia> include("rk1ex1.jl")

julia> testrungekutta(1,100)

#N=10 Tmax=1.000000
# t x
0.1000 1.1051708
0.2000 1.2214026
0.3000 1.3498585
(中略)
0.9800 2.6644562
0.9900 2.6912345
1.0000 2.7182818

julia>
```

次は van der Pol 方程式の問題の Julia によるプログラムである。Tmax, N を実行時に入力できるような工夫を入れてみた。

rk2ex1.jl

```
# rk2ex1.jl (Runge-Kutta method for van der Pol equation)
#
# 使い方
#   julia rk2ex1.jl              デフォルト Tmax=50, N=1000
#   julia rk2ex1.jl 100 2000     Tmax=100, N=2000
#   echo 'plot "rk2ex1.data" using 2:3 with l"' | gnuplot

using Printf

mu=1.0

# Runge-Kutta 法の 1 ステップ
function rungekutta(f,t,x,dt)
  k1=dt*f(t,x)
  k2=dt*f(t+dt/2, x+k1/2)
  k3=dt*f(t+dt/2, x+k2/2)
  k4=dt*f(t+dt, x+k3)
  x + (k1 + 2 * k2 + 2 * k3 + k4) / 6
end

# van der Pol 方程式の右辺
function f(t,x)
  y=similar(x)
  y[1]=x[2]
  y[2]=-x[1]+mu*(1.0-x[1]*x[1])*x[2]
  y
end

function testrungekutta(Tmax=50.0,N=1000)
  # 初期値
  t0=0.0
  x0=[0.1,0.1]
  #
  of=open("rk2ex1.data","w")
  # Runge-Kutta 法
  dt=Tmax/N
  t=t0
  x=x0
  for i=1:N
    x=rungekutta(f,t,x,dt)
    t=i*dt
    s=@sprintf "%f %f %f\n" t x[1] x[2]
    print(s)
    print(of,s)
  end
  close(of)
end

# main
argc=length(ARGS)
if argc==0
        testrungekutta()
elseif argc==1
        testrungekutta(parse(Float64,ARGS[1]))
elseif argc==2
        testrungekutta(parse(Float64,ARGS[1]), parse(Int,ARGS[2]))
end
```

実行の仕方

```
% julia rk2ex1.jl
% gnuplot
gnuplot> plot "rk2ex1.data" using 2:3 with l
gnuplot> quit

% julia rk2ex1.jl 100 2000
```

(Tmax を 100, N を 2000 として実行)

Julia には、グラフィックスのためのパッケージが色々ある。Plots というパッケージを利用してグラフを描く機能を追加したのが、次のプログラムである。

rk2ex1plot.jl

```
# rk2ex1plot.jl (Runge-Kutta method for van der Pol equation)
#
# 使い方
#   julia> include("rk2ex1plot.jl")
#   julia> testrungekutta()
#   julia> testrungekutta(10)
#   julia> testrungekutta(100,10000)

using Printf
using Plots

gr() # デフォルトが GR なので不要という説もある
mu=1.0

# Runge-Kutta 法の 1 ステップ
function rungekutta(f,t,x,dt)
  k1=dt*f(t,x)
  k2=dt*f(t+dt/2, x+k1/2)
  k3=dt*f(t+dt/2, x+k2/2)
  k4=dt*f(t+dt, x+k3)
  x + (k1 + 2 * k2 + 2 * k3 + k4) / 6
end

function f(t,x)
  y=similar(x)
  y[1]=x[2]
  y[2]=-x[1]+mu*(1.0-x[1]*x[1])*x[2]
  y
end

function testrungekutta(Tmax=50.0,N=1000)
  tv=zeros(N+1,1)
  xv=zeros(N+1,1)
  yv=zeros(N+1,1)
  # 初期値
  t0=0.0
  x0=[0.1,0.1]
  #
  of=open("rk2ex1.data","w")
  s="# Lotka-Volterrra equation"; println(s); println(of,s)
  s="# t x y"; println(s); println(of, s)
  # Runge-Kutta 法
  dt=Tmax/N
  t=t0
  x=x0
  s=@sprintf "%f %f %f\n" t x[1] x[2]
  print(s)
  print(of,s)
  # 記録
  tv[1]=t; xv[1]=x[1]; yv[1]=x[2]
  # 時間を進める
  for i=1:N
    x=rungekutta(f,t,x,dt)
    t=i*dt
    tv[i+1]=t; xv[i+1]=x[1]; yv[i+1]=x[2]
    s=@sprintf "%f %f %f\n" t x[1] x[2]
    print(s)
    print(of,s)
  end
  close(of)
  p=plot(xv,yv,title="van der Pol",
         xaxis=("x"),yaxis=("y"),xlims=(-3,3),ylims=(-3,3),legend=false)
  savefig(p,"rk2ex1plot.png")
  display(p)
end

# main
```

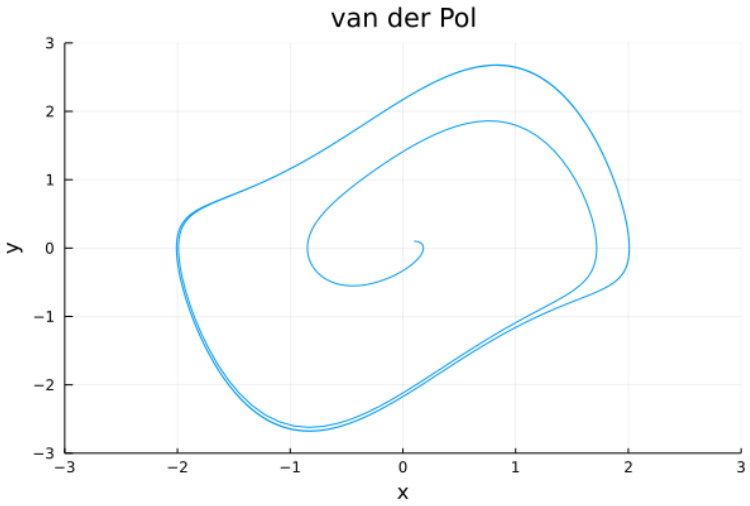


図 7: van der Pol 方程式の解軌道

3.6 Python 言語によるプログラム例

(ここは工事中。前項の Julia 言語によるプログラムと同じことをするプログラムを実現する。)

rk1ex1.py

```
# rk1ex1.py --- dx/dt=x (0<t<1), x(0)=1 を Runge-Kutta 法で解く
# Julia プログラムの Python バージョン
# rk1ex1.jl(http://nalab.mind.meiji.ac.jp/~mk/labo/text/intro-ode-simulation/node25.html)
# Runge-Kutta 法であるが、応用が効きにくい形。参考プログラム。
# シェルから起動したときコマンド・ライン引数を見るあたりは、参考になるかも。

import sys

def testrungekutta(Tmax=1.0, N=10):
    t0=0.0
    x0=1.0
    print('#N=%d Tmax=%f' % (N,Tmax))
    print('# t x')
    #Runge-Kutta 法
    dt=(Tmax-t0)/N
    t=t0
    x=x0
    for j in range(N):
        k1=dt*f(t,x)
        k2=dt*f(t+dt/2,x+k1/2)
        k3=dt*f(t+dt/2,x+k2/2)
        k4=dt*f(t+dt,x+k3)
        x += (k1+2*k2+2*k3+k4) / 6.0
        t = t0 + (j+1) * dt
        print('%.4f %.7f' % (t, x))

def f(t,x):
    return x

if __name__ == "__main__":
    argc = len(sys.argv)
    if argc == 1:
        testrungekutta()
    elif argc == 2:
        testrungekutta(float(sys.argv[1]))
    elif argc == 3:
        testrungekutta(float(sys.argv[1]), int(sys.argv[2]))
```

実行の仕方

```
% python rk1ex1.py


#N=10 Tmax=1.000000
# t x
0.1000 1.1051708
0.2000 1.2214026
0.3000 1.3498585
0.4000 1.4918242
0.5000 1.6487206
0.6000 1.8221180
0.7000 2.0137516
0.8000 2.2255396
0.9000 2.4596014
1.0000 2.7182797
%
```

次は van der Pol 方程式の問題の Julia によるプログラムである。Tmax, N を実行時に入力できるような工夫を入れてみた。また計算結果を rk2ex1.data というファイルに出力している。

rk2ex1.py

```
# rk2ex1.py --- Runge-Kutta method for van der Pol equation
# Julia プログラムの Python バージョン
# rk2ex1.jl(http://nalab.mind.meiji.ac.jp/~mk/labo/text/intro-ode-simulation/node25.html)
# お勧めという訳でもないが、それなりに応用が効く。

import sys
import numpy as np

def rungekutta(f,t,x,dt,args=()):
    k1=dt*f(t,x,args=args)
    k2=dt*f(t+dt/2, x+k1/2,args=args)
    k3=dt*f(t+dt/2, x+k2/2,args=args)
    k4=dt*f(t+dt, x+k3,args=args)
    return x + (k1 + 2 * k2 + 2 * k3 + k4) / 6

# van der Pol 方程式
def f(t, x, args=(1.0,)):
    # return np.array([x[1],-x[0]+mu*(1.0-x[0]*x[0])*x[1]])
    mu=args[0]
    y=np.empty(np.size(x))
    y[0]=x[1]
    y[1]=-x[0]+mu*(1.0-x[0]*x[0])*x[1]
    return y

def testrungekutta(Tmax=50.0, N=1000):
    mu=1.0
    t0=0.0
    x0=np.array([0.1,0.1])
    print('#N=%d Tmax=%f' % (N,Tmax))
    print('# t x')
    #Runge-Kutta 法
    dt=(Tmax-t0)/N
    t=t0
    x=x0
    with open('rk2ex1.data', mode='w') as fout:
        print('%f %f %f' % (t, x[0], x[1]))
        print('%f %f %f' % (t, x[0], x[1]), file=fout)
        for j in range(N):
            x=rungekutta(f,t,x,dt,args=(mu,))
            t = t0 + (j+1) * dt
            print('%f %f %f' % (t, x[0], x[1]))
            print('%f %f %f' % (t, x[0], x[1]), file=fout)

if __name__ == "__main__":
    argc = len(sys.argv)
    if argc == 1:
        testrungekutta()
    elif argc == 2:
        testrungekutta(float(sys.argv[1]))
    elif argc == 3:
        testrungekutta(float(sys.argv[1]), int(sys.argv[2]))
```

実行の仕方

```
% python rk2ex1.py
% gnuplot
gnuplot> plot "rk2ex1.data" using 2:3 with l
gnuplot> quit

% python rk2ex1.jl 100 2000
```

(Tmax を 100, N を 2000 として実行)

gnuplot を使わず、解軌道を直接描くのも簡単である。

rk2ex1plot.py

```
# rk2ex1.py --- Runge-Kutta method for van der Pol equation
# Julia プログラムの Python バージョン
# rk2ex1.jl(http://nalab.mind.meiji.ac.jp/~mk/labo/text/intro-ode-simulation/node25.html)
# お勧めという訳でもないが、それなりに応用が効く。

import sys
import numpy as np
import matplotlib.pyplot as plt

def rungekutta(f,t,x,dt,args=()):
    k1=dt*f(t,x,args=args)
    k2=dt*f(t+dt/2, x+k1/2,args=args)
    k3=dt*f(t+dt/2, x+k2/2,args=args)
    k4=dt*f(t+dt, x+k3,args=args)
    return x + (k1 + 2 * k2 + 2 * k3 + k4) / 6

# van der Pol 方程式
def f(t, x, args=(1.0,)):
    # return np.array([x[1],-x[0]+mu*(1.0-x[0]*x[0])*x[1]])
    mu=args[0]
    y=np.empty(np.size(x))
    y[0]=x[1]
    y[1]=-x[0]+mu*(1.0-x[0]*x[0])*x[1]
    return y

def testrungekutta(Tmax=50.0, N=1000):
    mu=1.0
    t0=0.0
    x0=np.array([0.1,0.1])
    print('#N=%d Tmax=%f' % (N,Tmax))
    print('# t x')
    #Runge-Kutta 法
    dt=(Tmax-t0)/N
    t=np.linspace(t0,Tmax,N+1)
    x=np.empty((N+1,2))
    x[0]=x0
    print('%f %f %f' % (t[0], x[0,0], x[0,1]))
    for j in range(N):
        x[j+1]=rungekutta(f,t,x[j],dt,args=(mu,))
        print('%f %f %f' % (t[j+1], x[j+1,0], x[j+1,1]))

    plt.plot(x[:,0],x[:,1])
    plt.show()

if __name__ == "__main__":
    argc = len(sys.argv)
    if argc == 1:
        testrungekutta()
    elif argc == 2:
        testrungekutta(float(sys.argv[1]))
    elif argc == 3:
        testrungekutta(float(sys.argv[1]), int(sys.argv[2]))
```

3.7 課題

3.7.1 定数係数線形常微分方程式

実数の定数 a, b, c, d に対して、

$$\frac{dx}{dt}(t) = ax(t) + by(t), \tag{20a}$$

$$\frac{dy}{dt}(t) = cx(t) + dy(t) \tag{20b}$$

という連立微分方程式を考える。初期条件として

$$x(0) = x_0, \quad y(0) = y_0 \tag{21}$$

の形のものを考える。これを解くプログラムを作成し、解軌道 (曲線 $(x(t), y(t))$) を描け。 複数の解軌道を同時に描いてみよ。

この微分方程式は、多くの現象の数理モデルとなる。力学においては、つりあいの位置の近くの微小振動を表す多くの微分方程式がこの形となる。また、力学系の平衡点 (不動点) の線形安定性解析の基礎となるので、その観点からも重要である。

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

とおくと、(20a), (20b) は

$$\frac{d\boldsymbol{x}}{dt}(t) = A\boldsymbol{x}(t), \tag{22}$$

(21) は

$$\boldsymbol{x}(0) = \boldsymbol{x}_0 \tag{23}$$

と表せる。(22) の形の微分方程式は、定数係数線形常微分方程式と呼ばれる。

(22), (23) の解は、行列の指数関数を用いると、次のように得られる。

$$\boldsymbol{x}(t) = e^{tA}\boldsymbol{x}_0. \tag{24}$$

これは基本的な問題であり、微分方程式のテキストはもちろんであるが、線形代数のテキストで言及されている場合もある。笠原 [8] を推奨する (説明を端折る本が多い中で、親切な記述が嬉しい)。あるいは、ちょっと雑であるけれど、桂田 [2] (4 節「定数係数線型常微分方程式」特に §4.2.2) はアクセスしやすいかも。

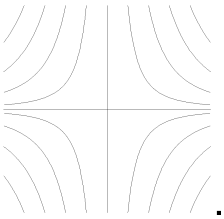


図 8: $\begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix}$

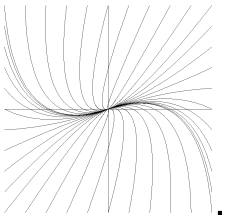


図 9: $\begin{pmatrix} -\frac{4}{5} & -\frac{3}{5} \\ \frac{2}{5} & -\frac{11}{5} \end{pmatrix}$
(吸)

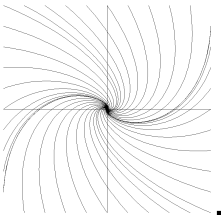


図 10: $\begin{pmatrix} \frac{8}{5} & -\frac{9}{5} \\ \frac{6}{5} & -\frac{13}{5} \end{pmatrix}$
(湧)

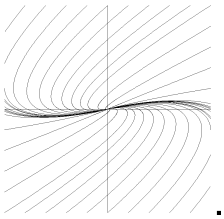


図 11: $\begin{pmatrix} \frac{2}{5} & \frac{9}{5} \\ -\frac{1}{5} & -\frac{8}{5} \end{pmatrix}$
(湧)

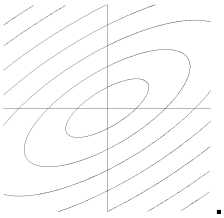


図 12: $\begin{pmatrix} -1 & 2 \\ -1 & 1 \end{pmatrix}$

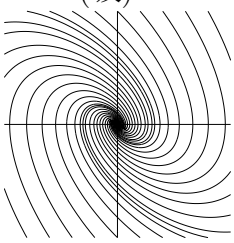


図 13: $\begin{pmatrix} 0 & 1 \\ -1 & -1 \end{pmatrix}$

3.7.2 Lotka-Volterra の方程式

Lotka-Volterra の方程式の初期値問題

$$\frac{dx}{dt} = ax - bxy, \tag{25}$$

$$\frac{dy}{dt} = -cy + dxy \tag{26}$$

$$x(0) = x_0, \quad y(0) = y_0 \tag{27}$$

を解くプログラムを作成し、解軌道 (曲線 $(x(t), y(t))$) を描け。複数の解軌道を同時に描いてみよ。Lotka-Volterra の方程式について資料を探して調べてみよ。

(まったく何も紹介しないのも何なので、ブラウン [9] をあげておく。「第 1 次世界大戦中、漁に出ることが少なかったにもかかわらず、食用魚が減って、食用に適さないサメなどの軟骨魚類が増えたのはなぜか」というミステリーに対するヴォルテラの謎解きが説明されている。)

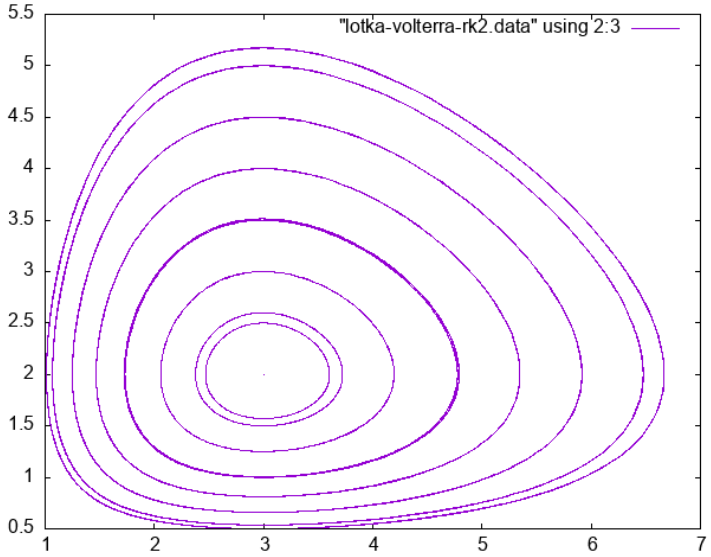


図 14: Lotka-Volterra 方程式 ($a = 2$, $b = 1$, $c = 3$, $d = 1$) の解軌道

この微分方程式は、色々な一般化がある。参考書としては、今・竹内 [10] がある (丸善 eBook で読むことが出来る)。

卒業研究レポートである長谷川 [11] では、May-Leonard [12] のモデルも扱っている。

3.7.3 SIR モデルの方程式

SIR モデルの微分方程式の初期値問題

$$\frac{dS}{dt}(t) = -\beta S(t)I(t), \tag{28}$$

$$\frac{dI}{dt}(t) = \beta S(t)I(t) - \gamma I(t), \tag{29}$$

$$S(0) = N - I_0, \quad I(0) = I_0 \tag{30}$$

を解くプログラムを作成し、S, I, $R = N - S - I$ の時間変化、曲線 $(S(t), I(t))$ を描け。

SIR モデルは重要で、資料が豊富にある。まずは、佐藤 [13](オリジナルの Kermack-McKendrick の論文を読み解いたもの) を見ることを勧める。それから、ゼミで学生の相手をしたときに作ったメモである桂田 [14] もあげておく。

3.7.4　2 変数版 BZ 反応 (準備中)

$$\varepsilon \frac{dx}{dt} = x(1-x) - fz\frac{x-q}{x+q}, \tag{31}$$

$$\frac{dz}{dt} = x - z. \tag{32}$$

池田・末松 [15]、それから、いわゆる手前味噌ということになるけれど、桂田研の卒業研究レポートである小林 [16] をあげておく。

[16] へのコメントに書いておいたけれど、素朴な Runge-Kutta 法でのシミュレーションは難しいかもしれない。RKF45 の話はこの文書にも書いておくべきか。

4　2 階微分方程式

4.1　はじめに

物理学に疎い人にはピンと来ないかもしれないが、Newton の運動方程式が 2 階の微分方程式である、という話をする。

力学においては、Newton の運動の三法則というのが基本である。そのうちの第二法則は、Newton の運動方程式とも呼ばれるもので、言葉で書くと「質量かける加速度 = 力」という内容である。

質点の運動の場合、時刻 t における質点の位置を $\boldsymbol{x}(t)$ で表すと、加速度は $\frac{d^2\boldsymbol{x}}{dt^2}(t)$ であるから、運動方程式は

$$m\frac{d^2\boldsymbol{x}}{dt^2}(t) = \boldsymbol{f}(t) \tag{33}$$

と表せる。ただし質点が受ける力を $\boldsymbol{f}(t)$ と表した。

$\boldsymbol{f}$ が既知とすれば、これは 2 階の微分方程式である。

この後に説明する、2 階の微分方程式を 1 階の微分方程式に書き直す方法が適用できる。ある意味でフライングになるが、それをやってみよう。

$$\boldsymbol{y}(t) := \frac{d\boldsymbol{x}}{dt}(t), \quad \boldsymbol{X}(t) := \begin{pmatrix} \boldsymbol{x}(t) \\ \boldsymbol{y}(t) \end{pmatrix} \tag{34}$$

とおくと (33) は次の方程式に書き換えらる。

$$\frac{d\boldsymbol{X}}{dt}(t) = \begin{pmatrix} \frac{d\boldsymbol{x}}{dt}(t) \\ \frac{d\boldsymbol{y}}{dt}(t) \end{pmatrix} = \begin{pmatrix} \boldsymbol{y}(t) \\ \frac{d^2\boldsymbol{x}}{dt^2}(t) \end{pmatrix} = \begin{pmatrix} \boldsymbol{y}(t) \\ \frac{1}{m}\boldsymbol{f}(t) \end{pmatrix}.$$

$$\boldsymbol{F}(\boldsymbol{X}, t) := \begin{pmatrix} \boldsymbol{y} \\ \frac{1}{m}\boldsymbol{f}(t) \end{pmatrix} \tag{35}$$

とおくと、

$$\frac{d\boldsymbol{X}}{dt}(t) = \boldsymbol{F}(\boldsymbol{X}(t), t). \tag{36}$$

4.2 ターゲット問題 1 自由落下

(2 階の微分方程式といえば、やはり運動方程式は外せないし、運動方程式といえば、簡単すぎると言う人がいるかもしれないが、やはり最初は自由落下ではないだろうか…)

$$\frac{d^2x}{dt^2} = -g, \tag{37}$$

$$x(0) = 100, \quad x'(0) = 0. \tag{38}$$

ここで g は重力加速度と呼ばれる正定数 (SI 単位系では、9.8 $\mathrm{m/s^2}$) である。(100 m の高さから落ちると…)

4.3 1 階の方程式への変換法

$$y := \frac{dx}{dt} \tag{39}$$

とおくと

$$\frac{dx}{dt} = y, \quad \frac{dy}{dt} = \frac{d}{dt}\frac{dx}{dt} = \frac{d^2x}{dt^2} = -g.$$

そこで

$$\vec{x} := \begin{pmatrix} x \\ y \end{pmatrix}, \quad \vec{f}(\vec{x}, t) := \begin{pmatrix} f_x(x, y, t) \\ f_y(x, y, t) \end{pmatrix} = \begin{pmatrix} y \\ -g \end{pmatrix}, \quad \vec{x}_0 := \begin{pmatrix} 100 \\ 0 \end{pmatrix}$$

とおくと (f_x, f_y は $\vec{f}$ の x 成分, y 成分という意味で、偏微分ということではない)

$$\frac{d\vec{x}}{dt} = \vec{f}(\vec{x}, t), \quad \vec{x}(0) = \vec{x}_0.$$

1 階の微分方程式 (ただし 2 次元のベクトル値関数についての微分方程式) の初期値問題に変換できた。

4.4 C 言語による Runge-Kutta 法のプログラム例

1 階の微分方程式 (2 次元) に帰着されたので、前節の解説に従えば良いわけだが、1 つくらいプログラム例をあげておく。

rk2ex2.c

```
/*
 * rk2ex2.c (Runge-Kutta method for free fall)
 */

#include <stdio.h>

double g = 9.8;

int main(void)
{
  int i, N;
  double t, x, y, dt, k1x, k1y, k2x, k2y, k3x, k3y, k4x, k4y;
  double fx(double, double, double), fy(double, double, double), x0, y0;
  double Tmax;
  // 初期値 (100m の高さから初速 0 で降りる)
  x0 = 100.0; y0 = 0.0;
  // 分割数, 最終時刻 -> 時間刻み
  printf("# N, Tmax: "); scanf("%d%lf", &N, &Tmax);
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  y = y0;
  printf("# t x y\n");
  printf("%f %f %f\n", t, x, y);
  // Runge-Kutta 法
  for (i = 0; i < N; i++) {
    k1x = dt * fx(x, y, t);
    k1y = dt * fy(x, y, t);
    k2x = dt * fx(x + k1x / 2, y + k1y / 2, t + dt / 2);
    k2y = dt * fy(x + k1x / 2, y + k1y / 2, t + dt / 2);
    k3x = dt * fx(x + k2x / 2, y + k2y / 2, t + dt / 2);
    k3y = dt * fy(x + k2x / 2, y + k2y / 2, t + dt / 2);
    k4x = dt * fx(x + k3x, y + k3y, t + dt);
    k4y = dt * fy(x + k3x, y + k3y, t + dt);
    x += (k1x + 2 * k2x + 2 * k3x + k4x) / 6;
    y += (k1y + 2 * k2y + 2 * k3y + k4y) / 6;
    t = (i + 1) * dt;
    printf("%f %f %f\n", t, x, y);
  }
  return 0;
}

double fx(double x, double y, double t)
{
  return y;
}

double fy(double x, double y, double t)
{
  return - g;
}
```

コンパイル&実行

```
% cc -O3 -o rk2ex2 rk2ex2.c
% ./rk2ex2 > rk2ex2.data
500 4.55
%
```

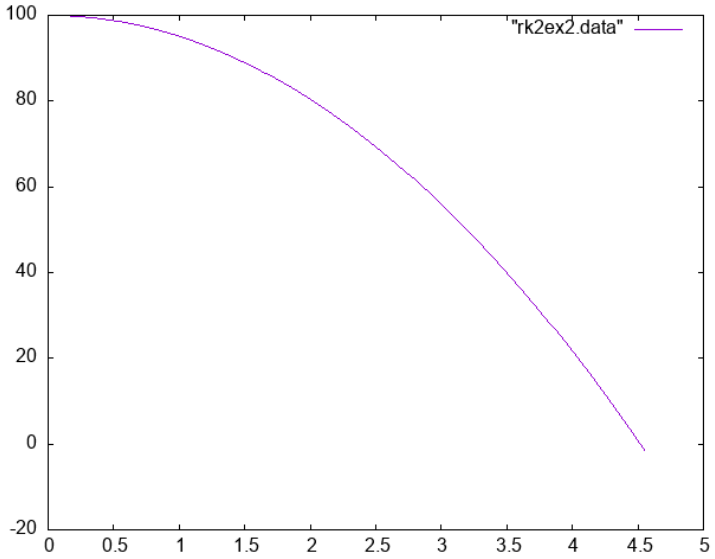


図 15: 高さ 100m からの自由落下 (横軸時刻, 縦軸高さ)

4.5 課題

運動方程式の例はいくらでもあげることが出来る。ここでは $x(t)$ が 1 次元の場合、つまり 1 階方程式に変換した際に 2 次元になる場合のみ取り上げる (質点系の運動では、次元が上がることになる)。

4.5.1 速度に比例する空気抵抗を受けるバネ振り子

$$m\frac{d^2x}{dt^2} = -kx - \gamma\frac{dx}{dt},$$
$$x(0) = 1, \quad x'(0) = 0.$$

$m > 0$, $k > 0$, $\gamma \geq 0$ は定数である。$\gamma = 0$ とすると空気抵抗がない場合で、いわゆる単振動になる。

4.5.2 単振り子

質量 n, ひもの長さ ℓ, 重力加速度 g, 時刻 t においてひもが鉛直線となす角 $\theta(t)$ とすると

$$m\ell\frac{d^2\theta}{dt^2}(t) = -mg\sin\theta(t).$$

$$\omega := \sqrt{\frac{g}{\ell}}$$

とおくと

$$\theta''(t) = -\omega^2\sin\theta(t). \tag{40}$$

振幅が小さいとき、単振動の方程式 $x'' + \omega^2 x = 0$ に近いが、 振幅が大きいとずれてくる。有名なので、色々な本に載っている。桂田 [17] というノートもある。

4.5.3 強制振動

ω を実定数として

$$\frac{d^2x}{dt^2}(t) + x(t) = \sin \omega t, \tag{41}$$

$$x(0) = x_0, \quad x'(0) = 0.$$

5 その他の課題

この節の課題は、3 次元以上の問題を集めてみた。

現状は (準備中) ばかり。有名な話も多いので、調べれば分かるだろう。さわりの部分で良ければ、桂田 [5] の第 3 章に書いてある。

5.1 Lorenz アトラクター

カオスで有名。有名なので授業で教わったこともあるであろう。また情報を得やすいだろう、ということで詳しい説明は後回し (準備中)。

$$\begin{cases} \dfrac{dx}{dt} = -\sigma x(t) + \sigma y(t) \\ \dfrac{dy}{dt} = Rx(t) - y(t) - x(t)z(t) \\ \dfrac{dz}{dt} = -bz(t) + x(t)y(t) \end{cases} \tag{42}$$

ここで σ, R, b は正定数である。Lorenz が選んだという値 $(\sigma, R, b) = (10, 28, 8/3)$ で実験したのが下の図である。

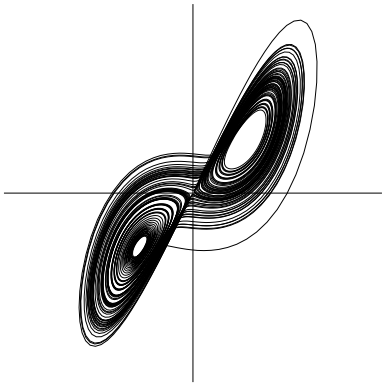


図 16: Lorent アトラクター ($\sigma = 10$, $R = 28$, $b = \frac{8}{3}$)

5.2 Kepler 運動

太陽と惑星が、万有引力に従って運動する場合に、惑星がどういう軌道を描くかという問題 (二体問題, the two-body problem) が Newton によって鮮やかに解かれた。

次に掲げる、いわゆる Kepler の三法則を満たすことが証明できた。

1. 惑星の軌道は、太陽を一つの焦点とする楕円である (1609 年)。

2. 面積速度は一定である (1609 年)。
   (太陽と惑星を結ぶ線分が単位時間に描く扇形状の領域の面積は一定である。)

3. 軌道の長半径の 3 乗と公転周期の 2 乗は比例する (1619 年)。

(Newton は、ペストの流行のために大学が休みになり、2 年ほど故郷に帰っているときにこの発見をしたという。逆二乗の法則の力を受ける天体の軌道は何かという (ハレー彗星で有名な天文学者) ハレーの問に、「それは楕円だ」と Newton が即答し、驚いたハレーが Newton にプリンキピア (「自然哲学の数学的原理」, 1687 年) を書かせた、というのは有名な話である。)

実は、惑星より一般の「万有引力を受ける天体」は、楕円以外の軌道を描く場合もあるが、例外的な場合を除くと、楕円、放物線、双曲線という円錐曲線 (二次曲線と言っても良い) になる。美しい結果と思う。

多くの本で (部分的に) 取り上げられているが、坂井 [18] (pp. 9–14) がよくまとまっていてお勧めである。

ところが、3 つの天体になると非常に手強い問題になる (いわゆる三体問題)。いくつかの有名な特殊解のシミュレーションをするのは面白い課題ではないだろうか。最近出版された浅田 [19] は、学生にも読みやすいと思われる。見かけたらページをめくってみることを勧める。

この問題は古典的ではあるが、天体の数をうんと増やすと、銀河生成のシミュレーションなど、現代的な話題になる。もっとも、これは (ゼミなどで) 簡単には追試できそうにないが。

5.3 ボールの運動

ボール投げは、入り口は高校物理程度で簡単であるが、意外に発展があり (空気から受ける力を考慮して変化球の数理にしたり、ボウリングやビリヤードのような問題にしたり)、奥深く面白い。

原理的なところから解明しようとすると、流体力学の問題になる (Reynolds 数が大きくなるので、数値計算は 2 次元でも大変である)。

マグナス力など近似法則を用いて常微分方程式で解析するのでも十分面白い。野球のボールについては、アデア [20] という文献があり、それに載っている測定データを用いてシミュレーションするのを、誰かやらないだろうか。

5.4 二重振り子

二重振り子は、比較的単純な系にもかかわらず、カオス現象が見られて興味深い (YouTube で色々動画を見ることが出来る)。

以前資料探しをしたことがあって

> 卒研資料室「二重振り子」[13]

に少し残っている。Mathematica のコードもあるし、割ときちんとした論文 (カオス絡み) もあるし、誰か卒業研究のテーマとして挑戦しないかな、と思っています (最初は論文を読んで、シミュレーションの追試をすることを目標にする)。

[13] http://nalab.mind.meiji.ac.jp/~mk/labo/library/double-pendulum/

5.5 渦糸系

N 個の渦糸からなる渦糸系の運動は、k 番目の渦糸の座標を $(x_k(t), y_k(t))$, 渦の強さを Γ_k として、

$$\frac{dx_k}{dt} = \sum_{\substack{j=1 \\ j\neq k}}^{N} \frac{-\Gamma_j}{2\pi} \frac{y_k(t) - y_j(t)}{(x_k(t) - x_j(t))^2 + (y_k(t) - y_j(t))^2}$$

$$\frac{dy_k}{dt} = \sum_{\substack{j=1 \\ j\neq k}}^{N} \frac{\Gamma_j}{2\pi} \frac{x_k(t) - x_j(t)}{(x_k(t) - x_j(t))^2 + (y_k(t) - y_j(t))^2}$$

という微分方程式で表わされる。これは Hamilton 力学系であり、数学的にはきれいな方程式であると言える。

参考書としては、岡本 [21]、今井 [22] 第 6 章 (§44) 、また中木 [23] も見ると良い。

稲垣亜希子・栗田智昭・田中賢史 [24] という卒業研究レポートがある。

(ひとり言) 竜巻とか、台風の藤原効果とか、渦関係で面白そうな話題は色々あるが、チャレンジするには適度に簡単化しないと困難だと思われる。渦糸はかなり大胆な簡単化だけれど、それでも自明でない構造を持っているらしい。渦糸のシミュレーションのアニメーションは、見ていてとても楽しい。

5.6 剛体の力学のシミュレーション

ずっと以前、テニス・ボールのバウンドに取り組んだ人がいたけれど、実はボールが “つぶれる” ので、とても難しいことが判明した。なるべく単純な問題と考えると、剛体として扱えるものが浮かんでくる。ボウリング、ビリヤード。考えてみると、質点や質点系の運動はシミュレーション・プログラムを良く目にするが、剛体の運動はそのものズバリを目にしたことがない。

学生に卒研で取り組んでもらったけれど (片倉ボウリング [25])、残念ながらシミュレーションまでは辿りつかなかった。その時に集めた資料は

卒研資料室「ボールの運動の数理」[14]

に置いてある。

5.7 蔵本モデル (準備中)

A Eigen のインストール

Eigen は、C++言語用の、ベクトル・行列演算用のクラス・ライブラリィである。

Eigen[15] から `eigen-3.3.9.zip` のような zip ファイルを入手して (2021/7/1 現在最新の stable release は 3.3.9 である)、ターミナルで以下のようなコマンドを実行する。

[14] `http://nalab.mind.meiji.ac.jp/~mk/labo/library/ball/`
[15] `https://eigen.tuxfamily.org/`

```
unzip eigen-3.3.9.zip
cd eigen-3.3.9
sudo cp -pr Eigen /usr/local/include
cd ..
```

こうしてインストールすると、/usr/local/include にファイルがコピーされるので、コンパイルするには、インクルード・ファイルのディレクトリィを-I /usr/local/include と指示すれば良い。

```
c++ -O -I/usr/local/include nantoka.cpp
```

B　Juliaのインストール、情報入手

B.1　インストール

- 「The Julia Programming Language」[16] の「Download Julia」[17] から macOS 用のファイル (julia-1.6.0-mac64.dmg のようなファイル名) を入手して、ダブル・クリックして

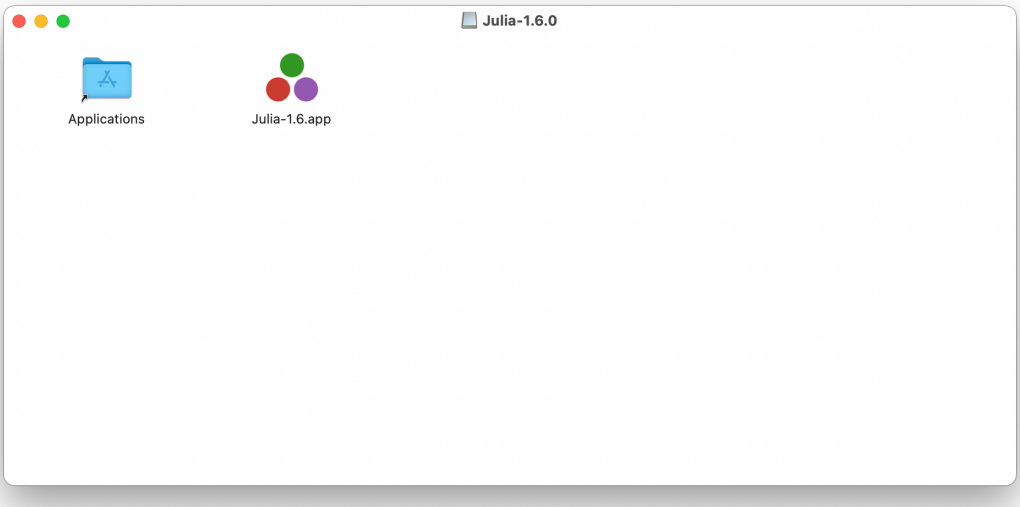


図 17: Julia-1.6 を Applications までドラッグして入れれば OK

これで、 アプリケーション ディレクトリィに、Julia-1.6.app と言うディレクトリィが出来る。ダブル・クリックすると、 ターミナルが起動してその中で Julia が起動する。

ターミナルで使うためにシンボリック・リンクを張ることを勧める。

シンボリック・リンクを張っておく

```
sudo rm -f /usr/local/bin/julia
sudo ln -s /Applications/Julia-1.6.app/Contents/Resources/julia/bin/julia /usr/local/bin
```

こうしておけば、ターミナルで julia と打つと julia が起動する。

[16] https://julialang.org/
[17] https://julialang.org/downloads/

- 私が用意するサンプル・プログラムでは、Printf, Plots を使うので、サンプル・プログラムを試すには、次のようにしてインストールしておくと良い。julia> と言うプロンプトに対して、

Printf, Plots の準備

```
% julia

(Julia の起動メッセージが出るが省略)

julia> using Pkg
julia> Pkg.add("Printf")
julia> Pkg.add("Plots")
```

実は Julia には、DifferentialEquations.jl[18] という有名なパッケージがあります。

```
julia> Pkg.add("DifferentialEquations")
```

B.2 情報の入手

公式の「Julia 1.6 Documentation」[19] は、英語で書かれている点は残念に感じる人が多いかもしれないし (英語はいざとなったら読むと覚悟しよう)、品質にばらつきがあるようにも思われる (私の独断です) が、十分な量の情報がある。

和書はあまりなく、2021 年 3 月時点で進藤・佐藤 [26] くらい。この本は必要のあるゼミ生には購入して配布するので、申し出ること。明治大学図書館の eBook で読むことも可能である。

私が書いた「Julia メモ」[20] が参考になるかも (実例に数値計算が多いので)。

C GLSC

GLSC が何か、どうやってインストールするか、とりあえず「GLSC についてまとめておく (2021 年版)」[21] を見て下さい。
試してみたいけれど、インストールする自信がない、という人は気軽に相談して下さい (10 分程度でインストール出来るはずです)。

プログラム例が見たいという人には、とりあえず 3.4 の rk2ex1++glsc.cpp を紹介しておきますが、解読するためには、桂田 [27] を見て下さい。

D 力学系についてメモ

平衡点、安定、不安定などの言葉が良く出て来る。基本的なことは知っておくべきである。

[18] https://diffeq.sciml.ai/stable/
[19] https://docs.julialang.org/en/v1/
[20] http://nalab.mind.meiji.ac.jp/~mk/labo/text/julia-memo/
[21] http://nalab.mind.meiji.ac.jp/~mk/knowhow-2021/node16.html

D.1　力学系とは

この文書では、微分方程式として

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

という形のもの (1 階正規形微分方程式) を考えた。特別な場合として、関数 $f(x,t)$ が t に依らない場合がある。その場合は単に $f(x)$ と書けば良いので、微分方程式は

$$\frac{dx}{dt}(t) = f(x(t)) \quad (\text{手短に書くと } \frac{dx}{dt} = f(x)) \tag{43}$$

という形になる。

これを**自励系微分方程式** (autonomous differential equation, autonomous system) という。力学系 (dynamical system) と呼ばれることも多い。

一つ注意が必要である。力学系という言葉は、微分方程式でない場合にも使われる (広い意味を持っている、ということである)。ネットで検索したりすると混乱するかもしれない。

上で「特別な場合」と書いたが、応用上出て来る微分方程式で力学系であるものは非常に多い。つまり「完全に一般ではなく、特別ではあるが、応用上は十分一般性が高い」。実際、この文書に現れる微分方程式で力学系でないものは、強制振動の方程式 (41) くらいである。従って、力学系に焦点を当てた議論は学ぶ価値がある。

D.2　平衡点とは

a が (43) の**平衡点** (equilibrium point, equilibrium solution) または**不動点** (fixed point) であるとは、

$$f(a) = 0$$

を満たすことをいう。

このとき

$$x(t) := a \quad (t \in \mathbb{R})$$

で定めた x は、(43) の解である。

相空間で a を通る解軌道が 1 点だけからなる、ということになる。

D.3　平衡点の安定性、漸近安定性

定義 D.1 (平衡点の安定、不安定) (43) の平衡点 a が (リャプノフの意味で) **安定** (stable) であるとは、

$$(\forall \varepsilon > 0)(\exists \delta > 0)(\forall x : (43)\text{ の解} \wedge |x(0) - a| < \delta)(\forall t \geq 0)\, |x(t) - a| < \varepsilon$$

を満たすことをいう。安定でないことを「不安定である」という。

つまり、任意の正の数 ε に対して、a に十分近いところから出発した任意の解は、a から距離 ε の範囲に止まる、ということである。

(私が学生で、このあたりのことを勉強したとき、「リャプノフの意味で」というのを見て、「そうでない意味で安定というのは、例えばどういうの？」と気になった。でもそういうのを

目にすることはなく、最近は「リャプノフの意味で」というのは省略されるのが多くなった。時間が経って、言葉が定着したということなのだろう。)

定義 D.2 (平衡点の漸近安定性) (43) の平衡点 a が**漸近安定** (asymptotically stable) であるとは、a が (リャプノフの意味で) 安定であり、かつ

$$(\exists \delta > 0)(\forall x : (43) \text{ の解} \wedge |x(0) - a| < \delta) \quad \lim_{t \to \infty} x(t) = a$$

を満たすことをいう。

判定法として、次の定理が使われることが非常に多い。

定理 D.3 f は C^1 級とする。(43) の平衡点 a について、f の a におけるヤコビ行列 $f'(a)$ のすべての固有値の実部が負ならば、a は漸近安定である。固有値の一つでも正の実部を持つならば、a は不安定である。

「ヤコビ行列って何ですか？」(学生が持っている教科書の索引にヤコビ行列がない…ぶつぶつ) $\Omega \subset \mathbb{R}^n$, $a \in \Omega$, $f \colon \Omega \to \mathbb{R}^m$ は微分可能とするとき、f の a におけるヤコビ行列とは、$m \times n$ 型の行列

$$f'(a) = \left(\frac{\partial f_i}{\partial x_j}(a) \right)$$

のことをいう。

力学系 $\frac{dx}{dt} = f(x)$ においては $m = n$ であることに注意しよう (微分方程式の左辺は n 次元, 右辺は m 次元なので)。ゆえに $f'(a)$ は n 次正方行列で、(重複度を込めて) n 個の固有値を持つ。行列 $f'(a)$ の成分は実数であるが、固有値には虚数が現れることもある。

この定理が、$n = 1$ の場合にも使えることを注意しておく。$n = 1$ のとき、f の a におけるヤコビ行列は、f の a における微分係数 $f'(a) \overset{\text{def.}}{=} \lim_{h \to 0} \dfrac{f(a+h) - f(a)}{h}$ そのものである。またその固有値は、$f'(a)$ (これは実数) である (一般に実数 A を、1×1 型の実行列とみなすとき、$Ax = Ax$ が $x = 1$ に対して成り立つので、A は A の固有値で、1 が固有ベクトルである。)。

ゆえに、定理を $n = 1$ の場合に限定すると、「$f'(a) < 0$ ならば a は漸近安定、$f'(a) > 0$ ならば a は不安定」ということになる。

E　公式の選択

E.1　はじめに

この文書は、古典的 (4 次の) Runge-Kutta 法推しの立場で書いた。とりあえず Runge-Kutta 法を勧める理由は、手間と精度のバランスがまあまあで、実際上現れる多くの問題をちゃんと解くことが出来るからである。Runge-Kutta 法を勧めるのは、多くのテキストで採用されている「定番のやり方」とみなしていている。

公式の良し悪しを判断するために、色々な観点がある。

(a) 次数と段数

(b) 誤差の推定、刻み幅の自動調節

(c) 数値的安定性 (特に硬い方程式対策)

(d) 構造の保存 (例えばエネルギー保存)

この文書では (a) のみ解説する (もしかすると、そのうち (b) について何か書くかも)。(c), (d) については省略する。

E.2 次数と段数 — 次数の高い方法が良いかも

以下、古典的な Runge-Kutta 法を単に Runge-Kutat 法と呼ぶ。

局所離散化誤差、全離散化誤差という言葉の定義は省略する (桂田 [3] などを見よ)。

Euler 法は 1 次、Runge-Kutta 法は 4 次の公式である。これは (刻み幅を h と書くことにして、滑らかな解を持つ問題に適用したとき)

- 局所離散化誤差 (local truncation error) がそれぞれ $O(h^2)$, $O(h^5)$ である
- 全離散化誤差 (total accumulated error) が (誤差の拡大がおきない限り) それぞれ $O(h)$, $O(h^4)$ である

ことを意味する。例えば、ある刻み幅 h で計算したときの誤差を $\frac{1}{10000}$ にしたければ (精度を 10 進法で 4 桁あげたければ)、ステップ数 (従って計算量) を、Euler 法では 10000 倍にしなければならないところ、Runge-Kutta 法では 10 倍にすれば良いと期待できる、ということになる。

数値例が見たければ

- [2] の §3.2.5, §3.3
- [3] の例 3.3

次数を高くするためには、より多くの手間をかける必要があるのが普通である。手間の目安としては、公式の段数が使われることが多い。それは時刻を 1 ステップ勧めるのに、微分方程式の右辺に現れる関数 f を何回計算するかを表している。Euler 法は 1 段、Runge-Kutta 法は 4 段である。粗い言い方をすると、ステップ数を同じにする場合、Runge-Kutta 法は Euler 法の 4 倍の計算量が必要ということである。

注意 E.1 「滑らかな解を持つ問題に適用したとき」と書いた。公式の次数 (位数, order) が m 次であるとは、C^m 級の一意解を持つ場合に適用して、局所離散化誤差が $O(h^{m+1})$ となること、とある。解が C^m 級にならない場合、m より小さい ℓ に対して $O(h^{\ell+1})$ となることしか期待できない、と言うことである。

この辺はどういう問題に適用するかによる。この文書で例として取り上げた微分方程式の問題では、(f が C^∞ 級であるので) 解は C^∞ 級になるので心配は無用であるが、いつでもそうであるとは限らないことは注意が必要である。

E.3 Euler 法の “意義”

とりあえず (最低でも) Runge-Kutta 法を使うようにということであるが、この文書でも、多くの本でも、Euler 法が解説されている。

Euler 法を解説する理由の一つは、この種の解法 (離散変数法？) の原理が分かりやすいからであろう。いきなり Runge-Kutta 法の公式を見せられたら、訳が分からなくなる可能性が高い。

しかし Runge-Kutta 法のような方法があるのだから、Euler 法を採用する理由はほとんどない (実用性はない) と言って過言ではない。時々 Runge-Kutta 法が面倒だから Euler 法で計算するという人がいる。非常におかしなことである (昭和だったら「○○△△□□□」と言うところだ)。

注意 E.2 (偏微分方程式に対する Euler 法の有効性) (そのうち書きます)

E.3.1 Heun の方法

F 問の解答

F.1 問 2

C 言語の `fopen()`, `flose()`, `fprintf()`, `fscanf()` を使うことでファイル入出力ができる。「簡単なファイル入出力」[22] を見よ。

[22] `http://nalab.mind.meiji.ac.jp/~mk/labo/text/nantoka-c++/node10.html`

toi2.c

```
/*
 * toi2.c --- euler1ex1.c (Euler method for Malthusian model)
 */

#include <stdio.h>
#include <stdlib.h>

double k = 1.0;

int main(void)
{
  int i, N;
  double t, x, dt;
  double f(double, double), x0;
  double Tmax;
  FILE *of;

  if ((of = fopen("euler1ex1.data", "w")) == NULL) {
    fprintf(stderr, "cannot open euler1ex1.data");
    exit(1);
  }
  // 初期値
  x0 = 1.0;
  // 最終時刻
  Tmax = 1.0;
  // 時間刻み
  printf("# N: "); scanf("%d", &N);
  fprintf(of, "# N = %d\n", N);
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  printf("# t x\n");
  fprintf(of, "# t x\n");
  printf("%f %f\n", t, x);
  fprintf(of, "%f %f\n", t, x);
  // Euler 法
  for (i = 0; i < N; i++) {
    x += dt * f(x, t);
    t = (i + 1) * dt;
    printf("%f %f\n", t, x);
    fprintf(of, "%f %f\n", t, x);
  }
  fclose(of);
  return 0;
}

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

C++言語の場合は、ifstream, ofstream, >>, << などを使ってファイル入出力ができる。「簡単なファイル入出力」[23] を見よ。

[23] http://nalab.mind.meiji.ac.jp/~mk/labo/text/nantoka-c++/node10.html

toi2.cpp

```
/*
 * toi2.cpp --- euler1ex1.c (Euler method for Malthusian model)
 */

#include <iostream>
#include <fstream>
using namespace std;

double k = 1.0;

int main(void)
{
  int i, N;
  double t, x, dt;
  double f(double, double), x0;
  double Tmax;
  ofstream ofs("euler1ex1.data");
  if (!ofs) {
    cerr << "cannot open euler1ex1.data" << endl;
    exit(1);
  }
  // 初期値
  x0 = 1.0;
  // 最終時刻
  Tmax = 1.0;
  // 時間刻み
  cout << "# N: "; cin >> N;
  ofs << "# N = " << N << endl;
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  cout << "# t x" << endl;
  ofs  << "# t x" << endl;
  cout << fixed;
  cout << t << " " << x << endl;
  ofs  << fixed;
  ofs << t << " " << x << endl;
  // Euler 法
  for (i = 0; i < N; i++) {
    x += dt * f(x, t);
    t = (i + 1) * dt;
    cout << t << " " << x << endl;
    ofs  << t << " " << x << endl;
  }
  ofs.close();
  return 0;
}

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

F.2 問 3

「C から gnuplot を呼び出す」[24]

[24] http://nalab.mind.meiji.ac.jp/~mk/labo/howto/intro-gnuplot/node21.html

toi3.c

```
/*
 * toi3.c --- euler1ex1.c (Euler method for Malthusian model)
 */

#include <stdio.h>
#include <stdlib.h>

double k = 1.0;

int main(void)
{
  int i, N;
  double t, x, dt;
  double f(double, double), x0;
  double Tmax;
  FILE *of;

  if ((of = fopen("euler1ex1.data", "w")) == NULL) {
    fprintf(stderr, "cannot open euler1ex1.data");
    exit(1);
  }
  // 初期値
  x0 = 1.0;
  // 最終時刻
  Tmax = 1.0;
  // 時間刻み
  printf("# N: "); scanf("%d", &N);
  fprintf(of, "# N = %d\n", N);
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  printf("# t x\n");
  fprintf(of, "# t x\n");
  printf("%f %f\n", t, x);
  fprintf(of, "%f %f\n", t, x);
  // Euler 法
  for (i = 0; i < N; i++) {
    x += dt * f(x, t);
    t = (i + 1) * dt;
    printf("%f %f\n", t, x);
    fprintf(of, "%f %f\n", t, x);
  }
  fclose(of);
  if ((of = popen("gnuplot", "w")) == NULL) {
    fprintf(stderr, "cannot open pipe.\n");
    exit(1);
  }
  fprintf(of, "plot \"euler1ex1.data\" with l\n");
  fflush(of);
  pclose(of);
  return 0;
}

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

もちろん gnuplot は事前にインストールしておく必要がある。コンパイル&実行すると、N の値を尋ねてくるので、例えば 100 と入力すると、gnuplot が起動されてグラフが描かれる

はず。

popen(), pclose() は、C++ でも使えるので (別の関数を使うべきか？)、次でとりあえず動く。

toi3.cpp

```
/*
 * toi3.cpp --- euler1ex1.c (Euler method for Malthusian model)
 */

#include <iostream>
#include <fstream>
using namespace std;

double k = 1.0;

int main(void)
{
  int i, N;
  double t, x, dt;
  double f(double, double), x0;
  double Tmax;
  ofstream ofs("euler1ex1.data");
  FILE *of;
  if (!ofs) {
    cerr << "cannot open euler1ex1.data" << endl;
    exit(1);
  }
  // 初期値
  x0 = 1.0;
  // 最終時刻
  Tmax = 1.0;
  // 時間刻み
  cout << "# N: "; cin >> N;
  ofs << "# N = " << N << endl;
  dt = Tmax / N;
  // 初期値
  t = 0.0;
  x = x0;
  cout << "# t x" << endl;
  ofs  << "# t x" << endl;
  cout << fixed;
  cout << t << " " << x << endl;
  ofs  << fixed;
  ofs << t << " " << x << endl;
  // Euler 法
  for (i = 0; i < N; i++) {
    x += dt * f(x, t);
    t = (i + 1) * dt;
    cout << t << " " << x << endl;
    ofs  << t << " " << x << endl;
  }
  ofs.close();

  if ((of = popen("gnuplot", "w")) == NULL) {
    fprintf(stderr, "cannot open pipe.\n");
    exit(1);
  }
  fprintf(of, "plot \"euler1ex1.data\" with l\n");
  fflush(of);
  pclose(of);

  return 0;
}

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

G　専用の関数を使ってみる — Python, Julia

常微分方程式を数値的に解くためのコードは、Runge-Kutta 法でよければ、自作するのはそれほど難しくない、それを実行してみよう、ということで、以前書いた文書のリニューアルの意味を込めてこの文書を書いたわけだが、そもそも 2021 年にもなってまだその方針でやっているのが間違っている、という気もする。

今では、Mathematica, MATLAB, Python, Julia, … などなど、専用の関数が用意されている処理系は簡単に利用可能である。

(以前でも、固有値問題などは専門家の作ったライブラリィを利用していたし。常微分方程式を解く手段も、そういうのに切り替えるべきなのかもしれない。)

ここでは、そういうやり方を紹介してみる。

例題として、無次元化した SIR モデル

$$\begin{aligned}\frac{dS}{dt} &= -R_0 SI,\\ \frac{dI}{dt} &= R_0 SI - I,\\ \frac{dR}{dt} &= I\end{aligned}$$

を取り上げる。ここで R_0 は基本再生算数と呼ばれる正の定数である。

> 参考「SIR モデルについてのメモ」
> http://nalab.mind.meiji.ac.jp/~mk/labo/text/sir.pdf
> 特に §3「方程式の無次元化と解の性質」の §3.2

初期条件として

$$S(0) = 1 - I_0, \quad I(0) = I_0, \quad R(0) = 0. \tag{44}$$

ここで I_0 は $0 < I_0 \ll 1$ を満たす数とする。ある感染症について、免疫を持たない集団に少数の感染者が侵入してどうなるか、という話。

G.1　Python

(しばらく工事中)

とりあえず、以前書いたメモ「Python で ode」[25] にリンクを張っておく。

scipy の `odeint()` は、定評のある ODEPACK を利用しているそうである。
「scipy.integrate.odeint」[26]

[25] http://nalab.mind.meiji.ac.jp/~mk/knowhow-2021/node26.html
[26] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html

testsir2.py

```
# testsir2.py
#
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# x=(S,I,R)
def sir(x, t, R0):
    return [-R0*x[0]*x[1], R0*x[0]*x[1]-x[1], x[1]]

R0=2.5
I0=0.001
x0=[1.0-I0,I0,0.0]
n=1000
t=np.linspace(0.0, 20.0, n+1)
x=odeint(sir, x0, t, args=(R0,))

plt.plot(t,x[:,0],'b', label='S')
plt.plot(t,x[:,1],'g', label='I')
plt.plot(t,x[:,2],'r', label='R')
plt.legend(loc='best')
plt.xlabel('t')
plt.grid()
plt.show()
```

簡単に書けるのは嬉しい。ついでにSI平面上で解の軌道を描いてみる。

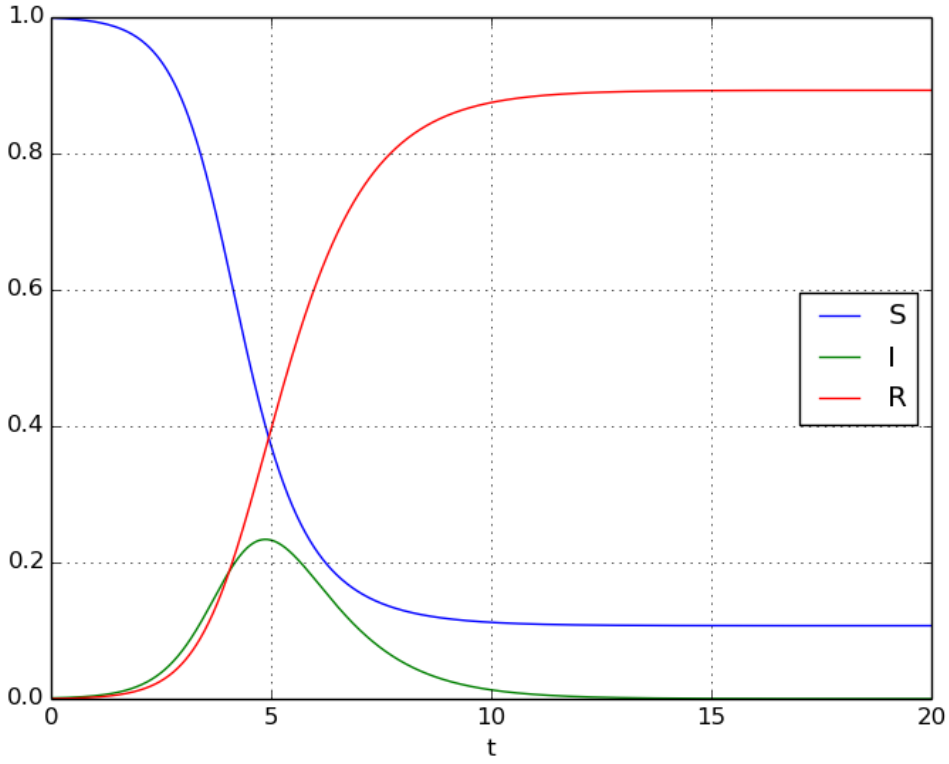


図 18: 無次元化 SIR モデルの解曲線 ($R_0 = 2.5$, $(S(0), I(0), R(0)) = (0.99, 0.01, 0)$)

testsir3.py

```
# testsir3.py --- SIR model
# coding: utf-8
#
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# x=(S,I,R)
def sir(x, t, R0):
    return [-R0*x[0]*x[1], R0*x[0]*x[1]-x[1], x[1]]

R0=2.5
I0=0.001
x0=[1.0-I0,I0,0.0]
n=1000
t=np.linspace(0.0, 20.0, n+1)
x=odeint(sir, x0, t, args=(R0,))

plt.figure(figsize=(10,5))

plt.subplot(121)
plt.plot(t,x[:,0],'b', label='S')
plt.plot(t,x[:,1],'g', label='I')
plt.plot(t,x[:,2],'r', label='R')
plt.legend(loc='best')
plt.xlabel('t')
```

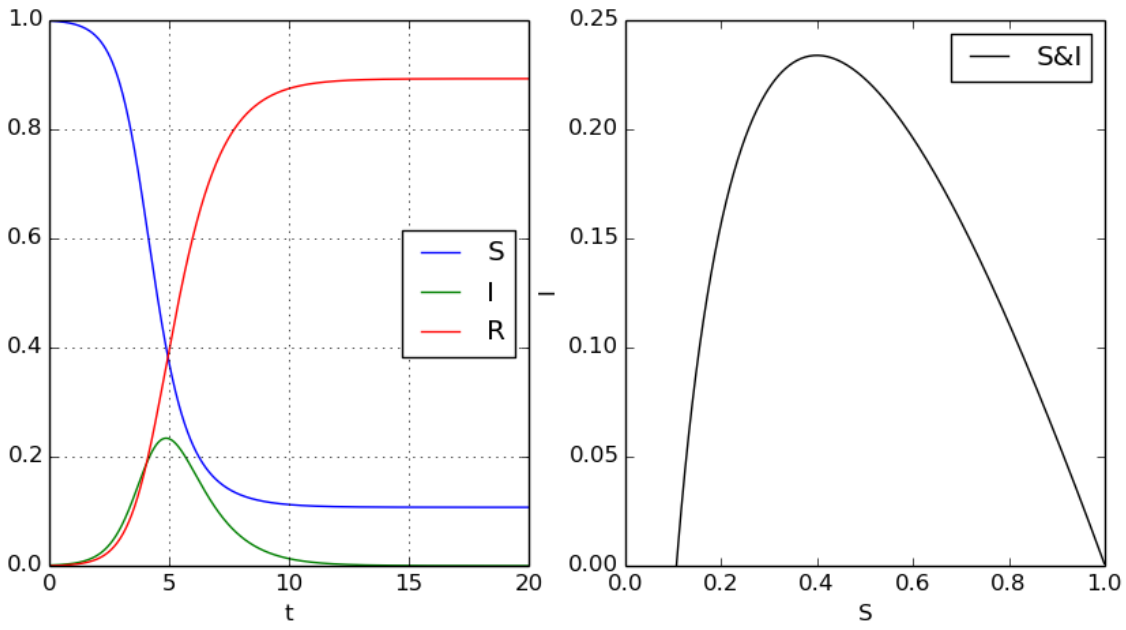


図 19: 無次元化 SIR モデルの解曲線と解軌道 $(R_0 = 2.5,\ (S(0), I(0), R(0)) = (0.99, 0.01, 0))$

新しく用意した、という `scipy.integrate.solve_ivp` を利用するプログラムは、以下のようになる。

testsir4.py

```
# testsir4.py --- SIR model
# coding: utf-8
#
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# x=(S,I,R)
def sir(t, x, R0):
    return [-R0*x[0]*x[1], R0*x[0]*x[1]-x[1], x[1]]

R0=2.5
I0=0.001
x0=[1.0-I0,I0,0.0]
T=20.0
sol=solve_ivp(sir, [0.0, T], x0, args=(R0,),
            dense_output=True, rtol=1e-10,atol=1e-10)
n=1000
t=np.linspace(0.0, T, n+1)
x = sol.sol(t)

plt.figure(figsize=(10,5))

plt.subplot(121)
plt.plot(t,x.T)
plt.xlabel('t')
plt.legend(['S', 'I', 'R'], shadow=True)
plt.title('SIR model (t-S,I,R)')

plt.subplot(122)
plt.xlim(0.0,1.0)
plt.plot(x[0], x[1], "-")
plt.xlabel("S")
plt.ylabel("I")
plt.title("SIR")
plt.show()
```

sir() の引数の順番は odeint() のときとは変える必要がある。 dense_output=True, rtol=1e-10,atol=1e-10 のあたりは、実は見様見真似で、どうするべきか、まだよく理解していない。

そうだ、流行曲線というのも描いてみよう。これは新規感染者数の時間経過を表すものである。$-\frac{dS}{dt}$, つまり $R_0S(t)I(t)$ を描けば良い。

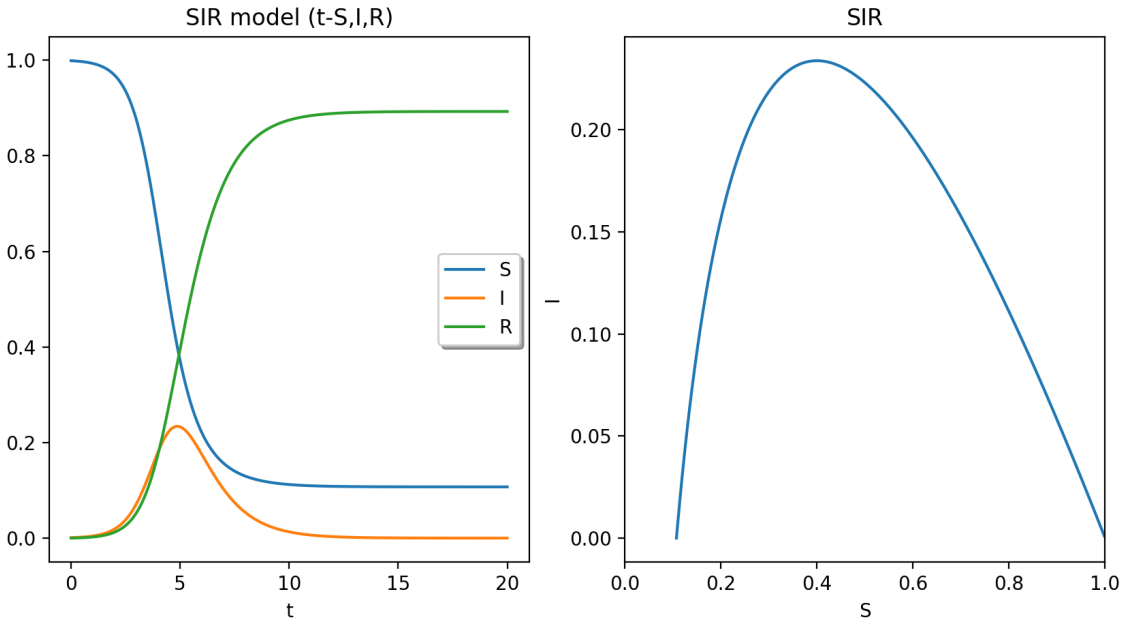


図 20: 無次元化 SIR モデルの解曲線と解軌道 $(R_0 = 2.5,\ (S(0), I(0), R(0)) = (0.99, 0.01, 0))$

testsir5.py

```
# testsir5.py --- SIR model
# coding: utf-8
#
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# x=(S,I,R)
def sir(t, x, R0):
    return [-R0*x[0]*x[1], R0*x[0]*x[1]-x[1], x[1]]

R0=2.5
I0=0.001
x0=[1.0-I0,I0,0.0]
T=20.0
sol=solve_ivp(sir, [0.0, T], x0, args=(R0,),
            dense_output=True, rtol=1e-10,atol=1e-10)
n=1000
t=np.linspace(0.0, T, n+1)
x = sol.sol(t)

plt.figure(figsize=(15,5))

plt.subplot(131)
plt.plot(t,x.T)
plt.xlabel('t')
plt.legend(['S', 'I', 'R'], shadow=True)
plt.title('SIR model (t-S,I,R)')

plt.subplot(132)
```

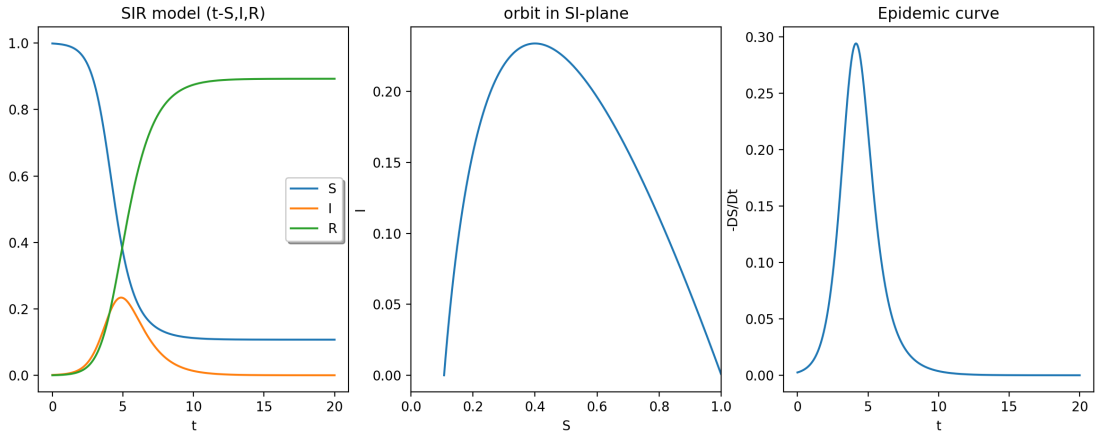


図 21: 無次元化 SIR モデルの解曲線, 解軌道, 流行曲線 ($R_0 = 2.5$, $(S(0), I(0), R(0)) = (0.99, 0.01, 0)$)

古い Python 環境しかインストールしていなくて、`solve_ivp()` が使えない人が、結構大勢いることに気づいた (何か勧めるときは、アップデートの仕方を教えてあげるべきだ)。ともあれ、`odeint()` で上と同じことをするには、次のようなプログラムを使えば良い。

testsir5b.py

```
# testsir5b.py --- SIR model
# coding: utf-8
#
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# x=(S,I,R)
def sir(x, t, R0):
    return [-R0*x[0]*x[1], R0*x[0]*x[1]-x[1], x[1]]

R0=2.5
I0=0.001
x0=[1.0-I0,I0,0.0]
n=1000
t=np.linspace(0.0, 20.0, n+1)
x=odeint(sir, x0, t, args=(R0,))

plt.figure(figsize=(15,5))

plt.subplot(131)
plt.plot(t,x[:,0],'b', label='S')
plt.plot(t,x[:,1],'g', label='I')
plt.plot(t,x[:,2],'r', label='R')
plt.legend(loc='best')
plt.xlabel('t')
plt.title('SIR model (t-S,I,R)')
plt.grid()

plt.subplot(132)
plt.xlim(0.0,1.0)
plt.plot(x[:,0],x[:,1],'k',label='S&I')
plt.legend(loc='best')
plt.xlabel('S')
plt.ylabel('I')
plt.title("orbit in SI-plane")

plt.subplot(133)
plt.plot(t,R0*x[:,0]*x[:,1],'b', label='S')
plt.xlabel('t')
plt.ylabel('-DS/Dt')
plt.title("Epidemic curve")

plt.show()
```

ついでに 3 次元相空間での軌道も描いてみる。

testsir6.py

```
# testsir6.py --- SIR model
# coding: utf-8
#
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# x=(S,I,R)
def sir(x, t, R0):
    return [-R0*x[0]*x[1], R0*x[0]*x[1]-x[1], x[1]]

#

R0=2.5
I0=0.001
x0=[1.0-I0,I0,0.0]
n=1000
t=np.linspace(0.0, 20.0, n+1)
x=odeint(sir, x0, t, args=(R0,))

fig=plt.figure(figsize=(20,5))

# S,I,R の時間変化
plt.subplot(141)
plt.plot(t,x[:,0],'b', label='S')
plt.plot(t,x[:,1],'g', label='I')
plt.plot(t,x[:,2],'r', label='R')
plt.legend(loc='best')
plt.xlabel('t')
plt.title('SIR model (t-S,I,R)')
plt.grid()

# SI 平面での軌道
plt.subplot(142)
plt.xlim(0.0,1.0)
plt.plot(x[:,0],x[:,1],'k',label='S&I')
plt.legend(loc='best')
plt.xlabel('S')
plt.ylabel('I')
plt.title("orbit in SI-plane")

# 流行曲線
plt.subplot(143)
plt.plot(t,R0*x[:,0]*x[:,1],'b', label='S')
plt.xlabel('t')
plt.ylabel('-DS/Dt')
plt.title("Epidemic curve")

# SIR 空間での軌道
```

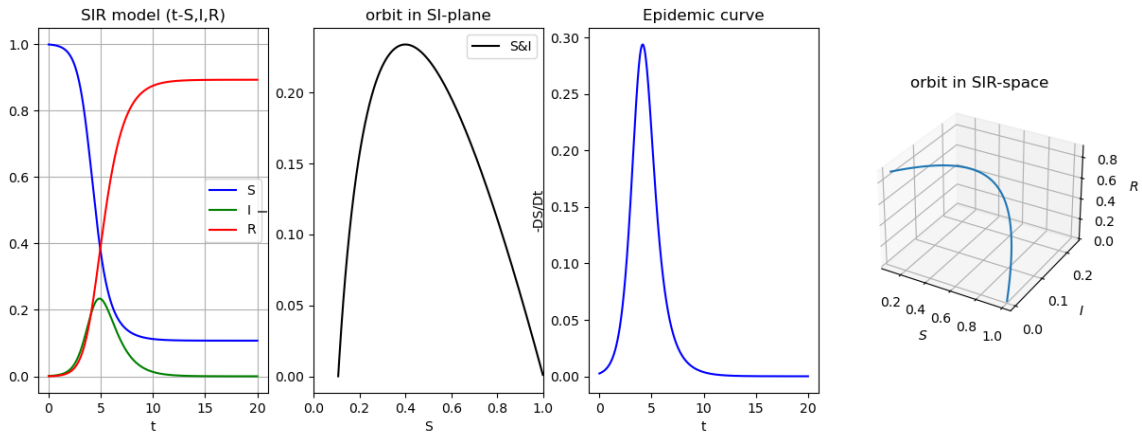


図 22: 無次元化 SIR モデルと 4 つの可視化

参考文献

[1] 小川知之, 宮路智行：数理モデルとシミュレーション, サイエンス社 (2020/12/19).

[2] 桂田祐史：常微分方程式の初期値問題の数値解法入門, https://m-katsurada.sakura.ne.jp/labo/text/num-ode.pdf (1995〜2011, 2021 小改訂).

[3] 桂田祐史：常微分方程式の初期値問題の数値解法, https://m-katsurada.sakura.ne.jp/labo/text/numerical-ode.pdf (1994〜).

[4] 桂田祐史：微分方程式入門, https://m-katsurada.sakura.ne.jp/lecture/kiso4/kiso4ode.pdf, 基礎数学 IV を講義したときの講義ノート (2004).

[5] 桂田祐史：常微分方程式ノート, https://m-katsurada.sakura.ne.jp/labo/text/members/ODE.pdf (2003).

[6] 桂田祐史：C 言語これくらいは覚えよう, https://m-katsurada.sakura.ne.jp/labo/text/cminimum/ (2005 年〜).

[7] 桂田祐史：gnuplot 入門, https://m-katsurada.sakura.ne.jp/labo/howto/intro-gnuplot/ (2002〜).

[8] 笠原晧司(こうじ)：微分方程式の基礎, 数理科学ライブラリー, 朝倉書店 (1982).

[9] M. ブラウン：微分方程式 下, シュプリンガー・フェアラーク東京 (2001).

[10] 今隆助, 竹内康博：常微分方程式とロトカ・ヴォルテラ方程式, 共立出版 (2018), https://elib.maruzen.co.jp/elib/html/Viewer/Id/3000086446.

[11] 長谷川成実：生物の増殖についての常微分方程式 (2011 年度桂田研卒業研究レポート), https://m-katsurada.sakura.ne.jp/labo/report/pdf/2011-hasegawa.pdf (2012).

[12] May, R. M. and Leonard, W. J.: Nonlinear Aspects of Competition Between Three Species, *SIAM Journal on Applied Mathematics*, Vol. 29, pp. 243–253 (1975).

[13] 佐藤總夫（ふさお）：自然の数理と社会の数理 II, 日本評論社 (1987), 版元在庫切れだったりしますが、図書館で借りるとか、 読む方法はあると思います。ゼミ生は見つからなければ相談して下さい。

[14] 桂田祐史:SIR モデルについてのメモ, https://m-katsurada.sakura.ne.jp/labo/text/sir.pdf (2021/2/7〜).

[15] 池田幸太, 末松 J. 信彦：数理解析であばく化学振動反応の本質, https://m-katsurada.sakura.ne.jp/labo/library/bz/ikeda_suematsu_abstract.pdf, 第 34 回 発展方程式若手セミナー報告集, pp. 73–97 (2012).

[16] 小林雄太：BZ 反応, 2004 年度桂田研 卒業研究レポート. https://m-katsurada.sakura.ne.jp/labo/report/open/2004-kobayashi.pdf. なお https://m-katsurada.sakura.ne.jp/labo/sotsugyou-report/node70.html も見ると良い。 (2005/2/28).

[17] 桂田祐史：単振り子の話, https://m-katsurada.sakura.ne.jp/labo/text/furiko/ (2007).

[18] 坂井秀隆：常微分方程式, 大学数学の入門, 東京大学出版会 (2015), 内容が非常に豊富. やや粗いけれど優れたテキスト. 丸善 eBook では https://elib.maruzen.co.jp/elib/html/BookDetail/Id/3000028058 でアクセス可能である.

[19] 浅田秀樹：三体問題 — 天才たちを悩ませた 400 年の未解決問題, ブルーバックス B-2167, 講談社 (2021/3/18).

[20] ロバート・アデア著, 中村和幸訳：ベースボールの物理学, 紀伊国屋書店 (1996/10/26), 原著は Robert K. Adair, The Physics of Baseball: Second Edition, Harper Perennial (1994). 2015 年に第 3 版が出版された.

[21] 岡本久：非線形力学 第 I 部「流体の運動と力学系」, 岩波書店 (1995), 岩波講座応用数学の分冊。今は入手しにくいかも。 読みたいゼミ生は相談して下さい。

[22] 今井功：流体力学 前編, 裳華房 (1973), 流体力学の基本的文献。後編は書かれなかった。

[23] 中木達幸：渦の相互作用問題の数値解析, 数理科学, 1998 年 3 月号, No. 417 (1998).

[24] 稲垣亜希子, 栗田智昭, 田中賢史：渦糸の力学系, 2002 年度桂田研卒業研究レポート. https://m-katsurada.sakura.ne.jp/labo/report/pdf/2002-inagaki-kurita-tanaka.pdf (2003).

[25] 片倉孝規：ボウリングの数理モデル, http://nalab.mind.meiji.ac.jp/2015/report/2016-katakura.pdf (2017/2/10).

[26] 進藤裕之, 佐藤建太：1 から始める Julia プログラミング, コロナ社 (2020/4/17), https://elib.maruzen.co.jp/elib/html/BookDetail/Id/3000091727.

[27] 桂田祐史：GLSC の紹介, https://m-katsurada.sakura.ne.jp/labo/howto/intro-glsc/ (2009〜).