情報工学実験１

「微分方程式の数値解法」

担当教員：　宮里 智樹
学籍番号：　０９５７０７Ｂ
氏　　名：　大 城 佳 明
提出期限：　2010 年 6 月 29 日
提 出 日：　2010 年 6 月 29 日

目 次

1 問題:1

実際にオイラー法を適用しようとすると，幾つかの問題のために 精度が悪く使われることはほとんどない．オイラー法の問題点を説明せよ．

1. 「刻み時間」の単位時間を短くすることで近似値を得るが，オ イラー法はこの近似値を補正しないため実用的な近似値を得るためにはかなり刻み時間を短くしなくてはならない
2. 1 階常微分方程式の数値解法としては誤差が蓄積されるため、精度が悪く、元の微分方程式によってはいかなる τ をとっても元の方程式の解に収束しないこともある
3. 極端に τ を小さく取ると丸め誤差が累積してきて逆に精度が落ちることもあり得る。
4. よって現実的なアルゴリズムとは言えなくなる．
5. オイラー法は実際にはほとんど用いられない．
6. オイラー法を応用し，さらなる近似値を得る他のアルゴリズムでは単位時間での計算の中で誤差を修正する処理が加わっている．
7. なので、学習目的以外であまり使われない。

2 問題:2

微分方程式 (11) の解を導出して確かめよ．

$$\frac{d}{dt}x = -x \quad 、 \quad x(0)=1 \text{ を解くと、}$$

$$\begin{aligned}
\frac{dx}{dt} &= -x \\
dx &= -xdt \\
-\frac{1}{x}dx &= dt \\
\int -\frac{1}{x}dx &= \int dt \\
-log|x| &= t + C \\
x &= e^{-t+C} \\
x &= C_1 e^{-t}
\end{aligned}$$

$x(0)=1$ より、

$$\begin{aligned}
x(0) &= C_1 e^0 = 1 \\
C_1 &= 1 \\
x &= e^{-t}
\end{aligned}$$

3 問題:3

ボールの軌道のグラフを gnuplot で出力せよ．なお，時間刻み τ は $0 < x \leq 2$ の間で 5 個選ぶこと．補助的に Octave を用いてよもい.

ソース（baseball.cpp）

```
// baseball.cpp: オイラー法を用いて野球ボールの軌道を計算するプログラム
#include "NumMeth.h"

int main() {

  //* ボールの初期位置及び初期速度を設定する.
  double y1, speed, theta,top;
  double r1[2+1], v1[2+1], r[2+1], v[2+1], accel[2+1];
  cout << "高さの初期値（メートル）: "; cin >> y1;
  r1[1] = 0; r1[2] = y1;    // 初期位置ベクトル
  cout << "初期速度（m/s）: "; cin >> speed;
  cout << "初期角度（度）: "; cin >> theta;
  const double pi = 3.141592654;
  v1[1] = speed*cos(theta*pi/180);   // 初期速度（x）
  v1[2] = speed*sin(theta*pi/180);   // 初期速度（y）
  r[1] = r1[1]; r[2] = r1[2];   // 初期位置および初期速度を設定
  v[1] = v1[1]; v[2] = v1[2];

  //* 物理パラメータを設定（質量，Cd 値など）
  double Cd = 0.35;        // 空気抵抗（無次元）
  double area = 4.3e-3;    // 投射物の横断面積（m^2）
  double grav = 9.81;      // 重力加速度（m/s^2）
    double mass = 0.145;    // 投射物の質量（kg）
  double airFlag, rho;
  cout << "空気抵抗（あり:1，なし:0）: "; cin >> airFlag;
  if( airFlag == 0 )
    rho = 0;   // 空気抵抗なし
  else
    rho = 1.2;   // 空気の密度（kg/m^3）
  double air_const = -0.5*Cd*rho*area/mass;   // 空気抵抗定数

  //* ボールが地面に着くまで，あるいは最大の刻み数になるまでループ
  double tau;
  cout << "時間刻みτ（秒）: "; cin >> tau;
  int iStep, maxStep = 1000;   // 最大の刻み数
  double *xplot, *yplot, *xNoAir, *yNoAir;
  xplot = new double [maxStep + 1];
  yplot = new double [maxStep + 1];
  xNoAir = new double [maxStep + 1];
  yNoAir = new double [maxStep + 1];
```

```
for( iStep=1; iStep<=maxStep; iStep++ ) {

  //* プロット用に位置 (計算値および理論値) を記録する
  xplot[iStep] = r[1];   // プロット用に軌道を記録
  yplot[iStep] = r[2];
  double t = ( iStep-1 )*tau;   // 現在時刻
  xNoAir[iStep] = r1[1] + v1[1]*t; // 位置 (x)
  yNoAir[iStep] = r1[2] + v1[2]*t - 0.5*grav*t*t; // 位置 (y)

    //* ボールの加速度を計算する
    double normV = sqrt( v[1]*v[1] + v[2]*v[2] );
  accel[1] = air_const*normV*v[1];   // 空気抵抗
  accel[2] = air_const*normV*v[2];   // 空気抵抗
  accel[2] -= grav;                  // 重力

  //* オイラー法を用いて，新しい位置および速度を計算する

  r[1] += v[1]*tau;
  r[2] += v[2]*tau;

  v[1] += accel[1]*tau;
  v[2] += accel[2]*tau;

 //* 最高到達高さを求める

  if(top<=yplot[iStep]) top=yplot[iStep];

  //* ボールが地面に着いたら (y < 0) ループを抜ける

    if( yplot[iStep] < 0 ) break;

}

//* 最大到達高さと滞空時間を表示する
cout << "最大到達高さは" << top << "メートル" << endl;
cout << "滞空時間は" << ( iStep-1 )*tau << "秒" << endl;

            //* プロットする変数を出力する
            //   xplot, yplot xNoAir, yNoAir
            ofstream xplotOut("xplot.txt"), yplotOut("yplot.txt"),
            xNoAirOut("xNoAir.txt"), yNoAirOut("yNoAir.txt");

          int i;
          for( i=1; i<=iStep; i++ ) {
            xplotOut << xplot[i] << endl;
```

```
            yplotOut << yplot[i] << endl;
          }
          for( i=1; i<=iStep; i++ ) {
            xNoAirOut << xNoAir[i] << endl;
            yNoAirOut << yNoAir[i] << endl;
          }

          delete [] xplot, yplot, xNoAir, yNoAir;  // メモリを開放

}
```

3.1 図 (gnuplot)

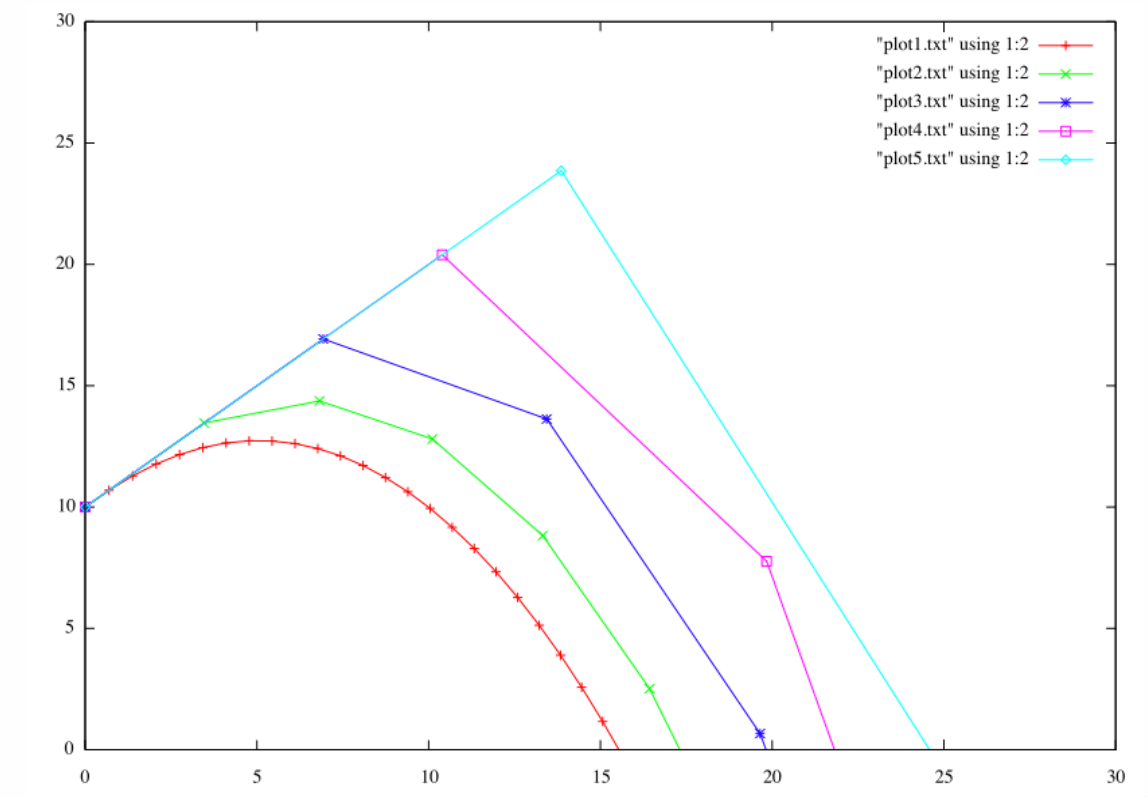

図 (1)：baseball.cpp

3.2 説明

1. 高さの初期値 (メートル)：20
2. 初期速度 (m/s)：30

3. 初期角度 (度)：30

4. 空気抵抗 (あり:1, なし:0)：1

5. 時間刻み τ(秒) は以下のように設定した

linename	τ
plot1	0.1
plot2	0.5
plot3	1.0
plot4	1.5
plot5	2.0

表 (1)：時間刻み τ(秒) の設定

4 問題:4

時間刻み τ の設定によって，計算結果が大きく異なること が確認できるが，その理由を考察せよ．

1. 計算結果を表にまとめた

linename	τ	最高到達高さ (m)	滞空時間 (秒)
plot1	0.1	12.7292	2.4
plot2	0.5	12.2386	3
plot3	1.0	16.9296	4
plot4	1.5	20.3945	4.5
plot5	2.0	23.8593	4

表 (2)：計算結果

2. 計算結果からわかるように、高さと時間が異なる
3. 図 (1) の plot1 と plot5 を比べて考察する
   (a) plot1 は約 1〜5m まで上昇し続けている
   (b) plot5 は約 1〜15m まで上昇し続けている
   (c) plot5 は初めの傾きのまま次の刻み時間になるまで同じ傾きである。
   (d) 実際は下降していても次の刻み時間がくるまでずっと上昇続けている
   (e) このように τ の値が大きくなればなるほど実際とは違う結果が生じる
   (f) したがって、最高到達高さが大きく変わる
4. 時間刻み τ の値によって、その時点での傾き (速度) を求めるのでどうしても誤差ができてしまう。

5　問題:5

オイラー法よりも高精度な数値計算アルゴリズムについて調べよ．余力のある人は，アルゴリズムを実装しその結果をオイラー法と比較してみよ．

5.1　ルンゲクッタ法

数値微分の世界で広く用いられている、常微分方程式の数値解法の一つである。おおざっぱにいって、関数を 4 回計算して、4 次近似の値を得ている。計算量と誤差の兼ね合いがよいのが用いられる理由の一つである。

$$\begin{aligned} k_1 &= hf(t_i, x_j) \\ k_2 &= hf(t_i + \frac{h}{2}, x_j + \frac{k_1}{2}) \\ k_3 &= hf(t_i + \frac{h}{2}, x_j + \frac{k_2}{2}) \\ k_4 &= hf(t_i + h, x_j + k^3) \\ x_{j+1} &= x_j + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

ここで添え字は何ステップ目かを、h は刻み幅を示す。k の値を求める際に h をかけないで求めることもでき、そのときは f の引数として与える時等に h をかけることになる。計算量、式の書きやすさ、計算誤差に対する厳密性等によって、必要な書き方は多少異なってくるだろう。

5.2　RKF45 法

RKF45 法、(Runge-Kutta-Fehlberg-4,5 Method) は、Fehlberg という人が 6 段 5 次の Runge-Kutta 公式をつくったものだが、係数を工夫することによって 4 次の値も求められるようにしている。これを 4 次公式が 5 次公式に埋め込まれている、と表現するらしく、埋め込み型 Runge-Kutta 法と呼ばれる。5 次の近似値から 4 次の近似値の誤差を見積もり、誤差があまりに多ければ打ち切ったり、刻み幅を調節することができる。

$$\begin{aligned} k_1 &= f(t_i, x_j) \\ k_2 &= f(t_i + \frac{1}{4}h, x_j + \frac{1}{4}hk_1) \\ k_3 &= f(t_i + \frac{3}{8}h, x_j + \frac{1}{32}h(3k_1 + 9k_2)) \\ k_4 &= f(t_i + \frac{12}{13}h, x_j + \frac{1}{2197}h(1932k_1 - 7200k_2 + 7296k_3)) \\ k_5 &= 6f(t_i + h, x_j + h(\frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4)) \\ k_6 &= f(t_i + \frac{1}{2}h, x_j + h(-\frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5)) \\ x_{j+1} &= x_j + h(\frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6) \\ x^*_{j+1} &= x_j + h(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5) \end{aligned}$$

係数をうつし間違えないように注意することが大切である。この点、4 次のルンゲクッタ公式は書きやすく分かりやすいことも見て取れるだろう。x が 5 次、x*が 4 次の近似値となる、上記では h をかけないで k を求めている。

この式も、括弧の位置、割り算をいつ行うか、割り算を掛け算にしたほうが計算ははやい…といった理由で多少の変形は考えられる。誤差の見積もりの式は以下のようになる。

$$\delta_{j+1} = x_{j+1} - x^*_{j+1} = h * (\frac{1}{360}k_1 - \frac{128}{4275}k_3 - \frac{2197}{75240}k_4 + \frac{1}{50}k_5 + \frac{2}{55}k_6))$$

この値が (h 等に比して) あまりに大きくなるときは、誤差が大きい、つまり計算になんらかの支障があることを示しているので、h の値を小さくすることにより、微小領域での変化も小さくなるよう計算を進めていくことができる。次の値として、4,5 次のどちらの値を使う方がいいのかは、筆者は把握していない。誤差評価のため 4 次の値を利用しただけで、5 次で計算を進める、というのか、5 次の値を誤差評価に用い、4 次で進めていく、というのか、よく分からないのである。5 次の誤差評価のためには 6 次を求めるべきだ、という論法も確かに成立する。

5.3 オイラー法とルンゲクッタ法

5.3.1 オイラー法

ソース（oira.cpp）

```
include <stdio.h>
double f1(double t,double x,double v);
double f2(double t,double x,double v);
int main()
{
  double x,v,t,dt,tmax;
  double k0[2];

  FILE *output;
  output=fopen("output.data","w");

  /*初期値*/
  x=1.0;
  v=0.0;
  dt=0.01;
  tmax=100;

  for(t=0.0;t<=tmax;t+=dt) {
    k0[0]=dt*f1(t,x,v);
    k0[1]=dt*f2(t,x,v);
    x=x+k0[0];
    v=v+k0[1];

    fprintf(output,"%f %f %f\n",t,x,v);
  }

  fclose(output);

  return 0;
}
```

```
double f1(double t,double x,double v)
{
  return v;
}

double f2(double t,double x,double v)
{
  return (-x);
}
```

5.3.2 ルンゲクッタ法

ソース（runge.cpp）

```
#include <stdio.h>
double f1(double t,double x,double v);
double f2(double t,double x,double v);
int main()
{
  double t,x,v,dt,tmax;
  double k1[2],k2[2],k3[2],k4[2];

  x=1.0;                    //位置の初期値
  v=0.0;                    //速度の初期値
  dt=0.01;                  //刻み幅
  tmax=500.0;               //繰り返し最大回数

  FILE *output;
  output=fopen("output.data","w");

  for(t=0;t<tmax;t+=dt) {
    k1[0]=dt*f1(t,x,v);
    k1[1]=dt*f2(t,x,v);
    k2[0]=dt*f1(t+dt/2.0,x+k1[0]/2.0,v+k1[1]/2.0);
    k2[1]=dt*f2(t+dt/2.0,x+k1[0]/2.0,v+k1[1]/2.0);
    k3[0]=dt*f1(t+dt/2.0,x+k2[0]/2.0,v+k2[1]/2.0);
    k3[1]=dt*f2(t+dt/2.0,x+k2[0]/2.0,v+k2[1]/2.0);
    k4[0]=dt*f1(t+dt,x+k3[0],v+k3[1]);
    k4[1]=dt*f2(t+dt,x+k3[0],v+k3[1]);

    x=x+(k1[0]+2.0*k2[0]+2.0*k3[0]+k4[0])/6.0;
    v=v+(k1[1]+2.0*k2[1]+2.0*k3[1]+k4[1])/6.0;

    fprintf(output,"%f %f %f\n",t,x,v);
```

```
  }
  fclose(output);
double f1(double t,double x,double v)
{
  return v;
}

double f2(double t,double x,double v)
{
  return (-x);
}

  return 0;
}
```

5.3.3 実行結果

1. オイラー法

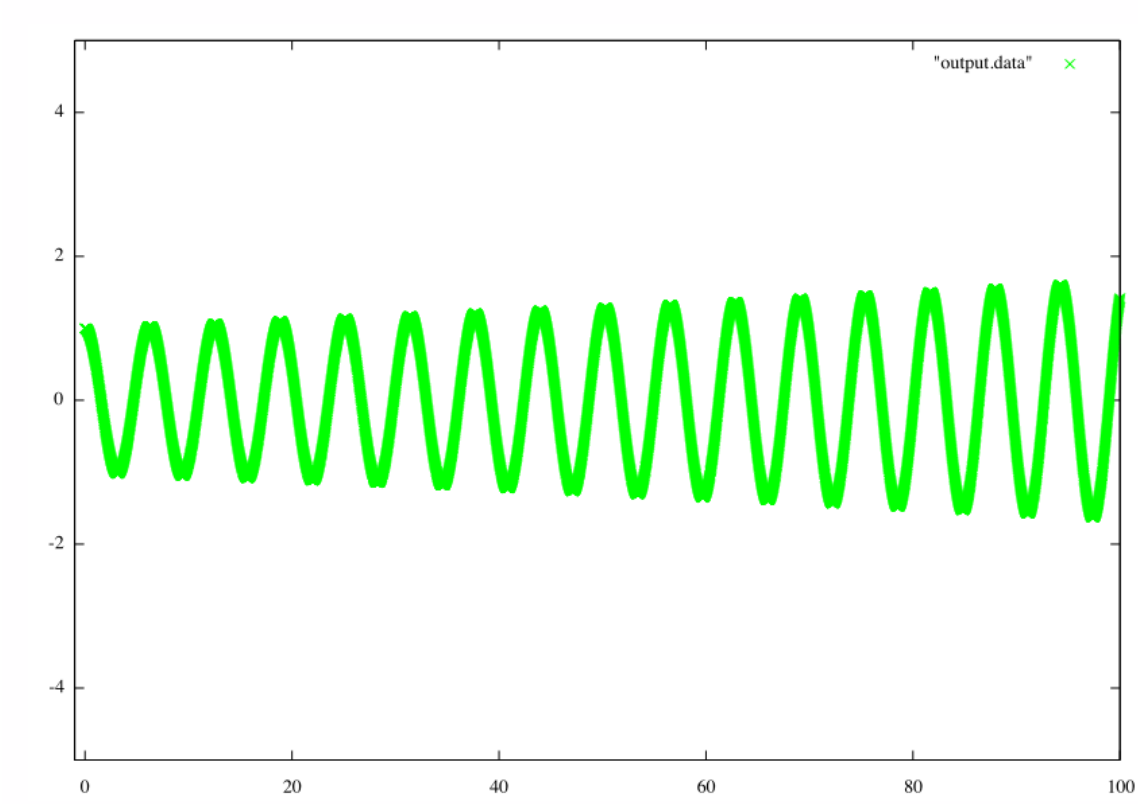

図 (2)：オイラー法による gnuplot

2. ルンゲクッタ法

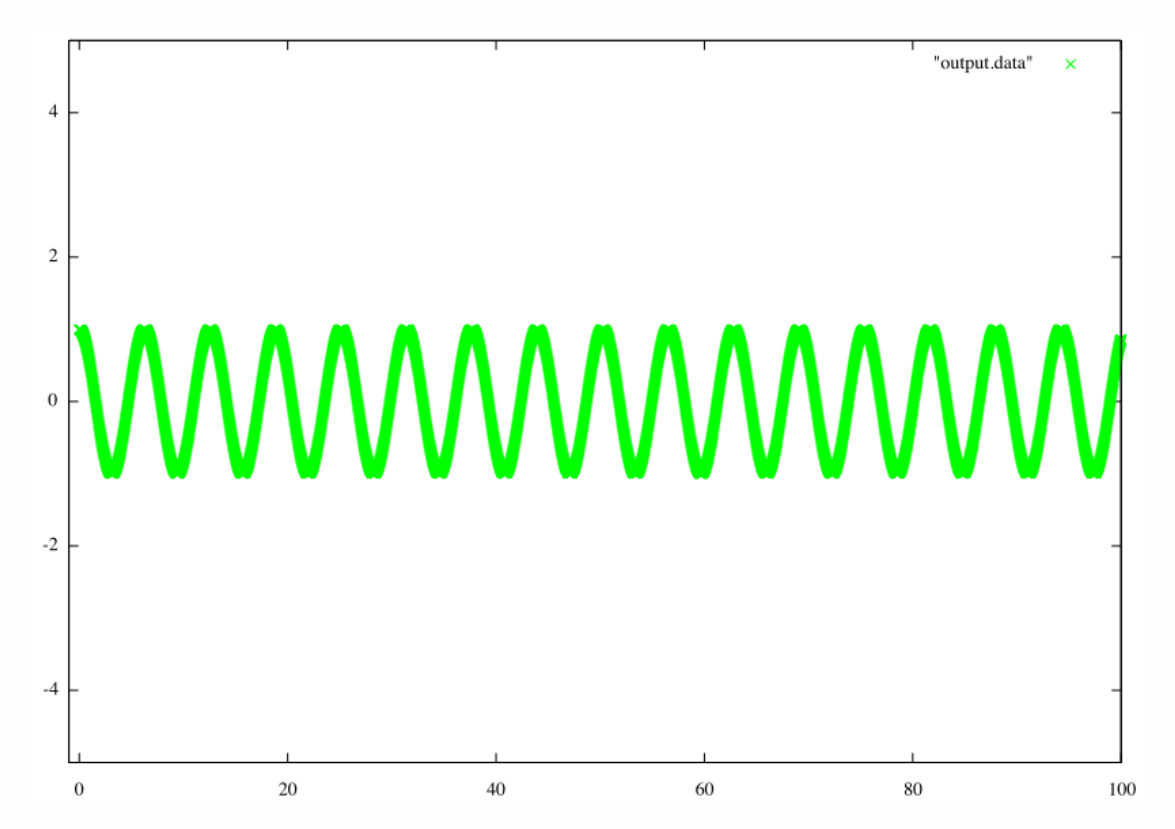

図 (3)：ルンゲクッタ法による gnuplot

5.3.4 考察

1. 図 (2) と図 (3) を比べる
2. 図 (2) のオイラー法は右に行くにつれて線の幅が広くなっている
3. 図 (3) のルンゲクッタ法はほとんど変わっていない
4. ルンゲクッタ法の方が細かいのでほぼ一定に見える

6　参考文献

データファイルの数値のプロット（その 3）
http://t16web.lanl.gov/Kawano/gnuplot/datafile3.html

ルンゲクッタ法
http://www.tzik.mydns.jp/ap2007/wiki/index.php?ルンゲタック法