# 埋め込み型ルンゲクッタ法による常微分方程式ソルバ
# ~マニュアル~

九州大学数理学研究院　秋山 正和

# はじめに

$$x'(t) = f(x(t), t)$$

という問題を解くとき，通常はオイラー法や多段のルンゲクッタ法を用いるだろう.

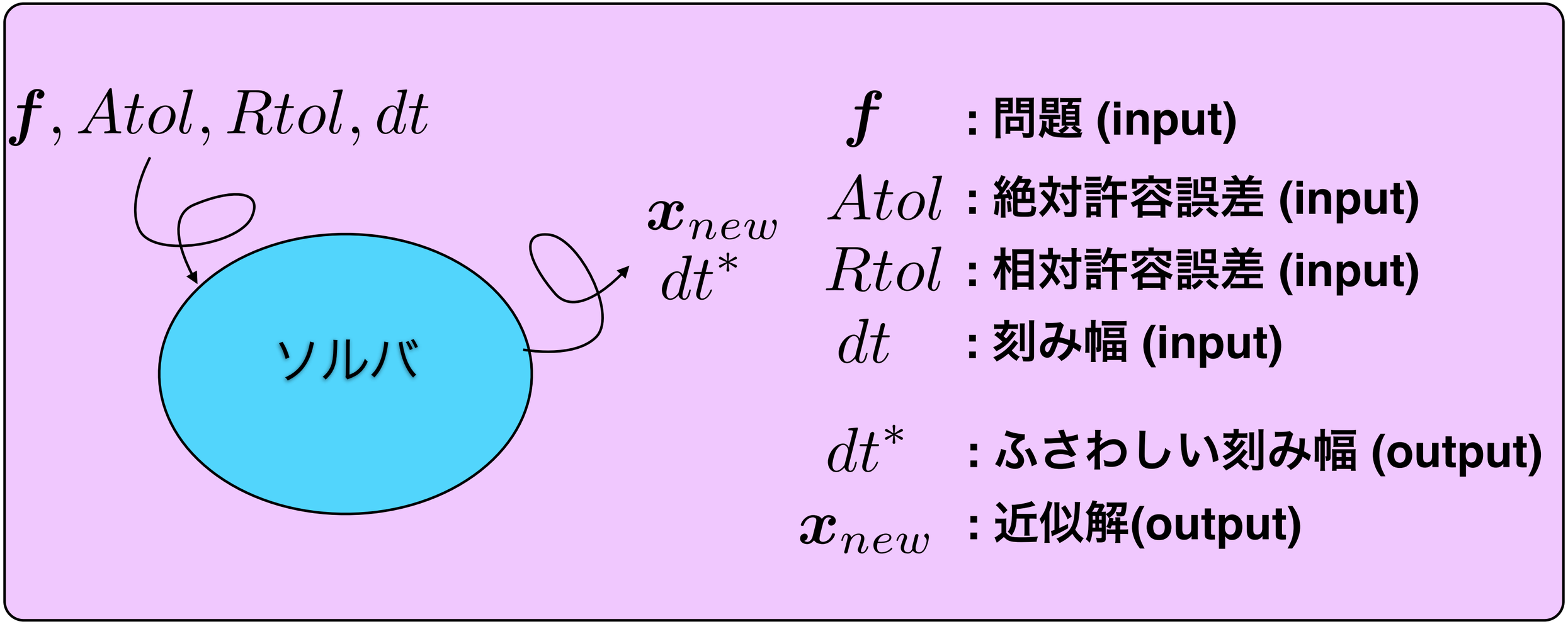
$$\boldsymbol{x}'(t) = \boldsymbol{f}(\boldsymbol{x}(t), t)$$

問題がベクトルになっても，やることは同じである.

大抵，多くの研究者は4段4次のルンゲクッタ法を使い，問題を解く.

この時，できるだけ dt を大きくとれるように計算したい．しかしながら，あまり大きくとると，真の解からの誤差が大きくなるので，葛藤が起こる．そこで．．．

発想を逆にして，ある程度まで誤差を許し，その範囲の中で取ることができる最大の dt を計算に使えないかを考える.




イメージ図

本ソルバはそれを可能とする「陽的埋め込み型ルンゲ・クッタ公式」を採用している．埋め込み型ルンゲ・クッタ公式の詳細はフォルダ（埋め込み型ルンゲクッタ法に関する文献/）を参照されたし．特に“Mathematicaの説明.pdf”はわかりやすい．常微分方程式の数値解法 I 及び II（E.ハイラー / S.P.ネルセット/ G.ヴァンナー）を読めば，完全に理解できるだろう．

この方法の良いところは, 誤差の推定と近似解との計算が一度にでき,（それが“埋め込み”という言葉の由縁）計算も超高速であることである．

重要な事は一般的なオイラー法やルンゲ・クッタ法と同様に，誤差に関する次数が存在することである．本ソルバでは2次から9次までの様々な次数のソルバを実装しており，解きたい問題によって次数を選択できる．

* 本ソルバはFSALスキームなMethodを多く用いているが，計算速度向上のためと，使い勝手の向上のために，あえてFSALスキームは捨てている．
FSALスキームバージョンがほしければ，各自作ること．

また，手っ取り早く使いたいなら，11ページから読んでもかまわない．

# ソルバの説明

**本ソルバは以下のような10種類のMethodを実装している.**

| | |
|---|---|
| Wolfram 2次(1)法 | Method Number 2 |
| Wolfram 3次(2)法 | Method Number 3 |
| Wolfram 4次(3)法 | Method Number 4 |
| Bogacki & Shampine 5(4)次法 | Method Number 5 |
| Verner 6(5)次法 | Method Number 6 |
| Verner 7(6)次法 | Method Number 7 |
| Verner 8(7)次法 | Method Number 8 |
| Verner 9(8)次法 | Method Number 9 |
| Bogacki & Shampine 3(2)次法 | Method Number 10 |
| Dormand & Prince 5(4)次法 | Method Number 11 |

**例えば**

**Bogacki & Shampine 5次(4)法とは Bogacki & ShampineがMethodの開発者であり，精度は5次であることを示す．(4)は誤差推定に使われる次数で普段は気にしなくて良い．Method Numberは9までなら次数と一致している.**

**10種類のソルバはC言語で実装されている.**

**各ソルバは共通の引数を持つ．例えばMethod Number 5を使う場合はＣでは次のように呼び出す．　(*より詳細には後述のサンプルプログラムを見ること．)**

```
Bogacki_Shampine54_n_dim(n,x,&time,&dt,Atol,Rtol,Kansu_f)
```

**Method Number 6を使うなら同じ引数で**

```
Verner65_n_dim(n,x,&time,&dt,Atol,Rtol,Kansu_f)
```

**とすればよい．つまり本ソルバは**

**「精度的に満足がいかないとき，次数を上げること(下げることも)が容易」であるように設計されている.**

# Method Number 5を使う場合

Bogacki_Shampine54_n_dim(n,x,&time,&dt,Atol,Rtol,Kansu_f)

**とするか**

Multiple_n_dim(5,n,x,&time,&dt,Atol,Rtol,Kansu_f)

**としてもよい．または一般に**

********_n_dim(n,x,&time,&dt,Atol,Rtol,Kansu_f)

**は**

Multiple_n_dim(p,n,x,&time,&dt,Atol,Rtol,Kansu_f)

**と同じである．**

| | |
|---|---|
| Wolfram 2次(1)法 | p=2 |
| Wolfram 3次(2)法 | p=3 |
| Wolfram 4次(3)法 | p=4 |
| Bogacki & Shampine 5(4)次法 | p=5 |
| Verner 6(5)次法 | p=6 |
| Verner 7(6)次法 | p=7 |
| Verner 8(7)次法 | p=8 |
| Verner 9(8)次法 | p=9 |
| Bogacki & Shampine 3(2)次法 | p=10 |
| Dormand & Prince 5(4)次法 | p=11 |

**Multiple_n_dimはMethodを様々に切り替えて使う場合に有用である．**

***後述のサンプルを見れば簡単に理解できる．**

# Sample1

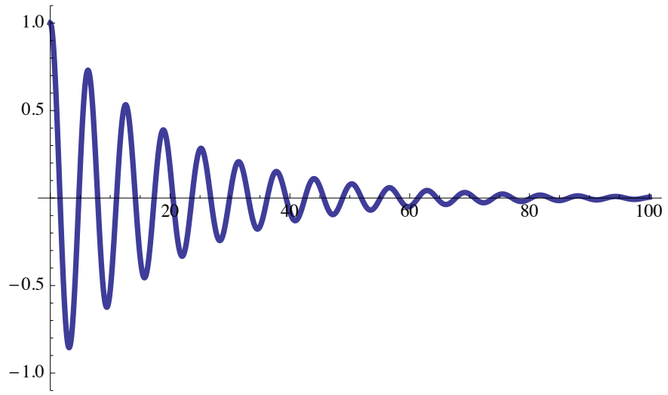
## バネ・ダンパ系の問題

**解くべき方程式は以下のような形になる．(変数の意味などは各自で考えて)**

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2 x = 0$$

$$x(0) = x_0 \quad x'(0) = p_0$$

**解のグラフはこんな感じになる．**




$0 < \gamma < \omega_0$ **のとき，厳密解は**

$$x(t) = e^{-\gamma t}\left(x_0 \cos\omega t + \frac{p_0 + \gamma x_0}{\omega}\sin\omega t\right)$$

$$p(t) = e^{-\gamma t}\left(p_0 \cos\omega t - \frac{p_0\gamma + \omega_0^2 x_0}{\omega}\sin\omega t\right)$$

**ただし**

$$p(t) := x'(t)$$

$$\omega := \sqrt{\omega_0^2 - \gamma^2}$$

# Sample1

## 問題は一階の微分方程式系で以下のようにもかける

$$\frac{dp}{dt} = -2\gamma p - \omega_0^2 x$$

$$\frac{dx}{dt} = p$$

## Sample1ではこれをMethod Number 5で解く.

$ cc main.c
$ ./a.out　　で実行できる.

main.c

main.c › main()

```c
#include<stdio.h>

#include "Embedded_Explicit_Runge_Kutta_Multiple.c"

#define     omega_0     (1.0)
#define     gamma       (0.05)
#define     omega       (sqrt(omega_0 * omega_0 - gamma * gamma))
```

← main.cでソルバをインクルードする.

← パラメタの設定

# Sample1

```
9  void
10 kansu_f(int n, double q[2], double q_dot[2], double time)
11 {
12     q_dot[0] = -2 * gamma * q[0] - omega_0 * omega_0 * q[1];
13     q_dot[1] = q[0];
14 }
15
16 int
17 main(void)
18 {
19     unsigned long long i_time = 0;
20
21     double          u[2];
22     double          p0 = 0, x0 = 1;
23     double          p = p0, x = x0;
24     double          genmitu_p, genmitu_x;
25     double          err_p = 0.0, err_x = 0.0;
26     double          max_err_p = 0.0, max_err_x = 0.0;
27
28     double          Atol = 1.0e-8;
29     double          Rtol = 1.0e-8;
30     double          t = 0;
31     double          dt = 0.0001;
32
33     u[0] = p;
34     u[1] = x;
35
```

← 解きたい問題をここに書く.

← 時間カウンター

← ソルバに渡す変数.

← 初期値

← 本当の変数に初期値を代入

← 厳密解用の変数

← 厳密解と数値解の誤差を代入する変数

← 上記の値の最大値

← 絶対許容誤差 1.0e-2~1.0e-15ぐらいが妥当

← 相対許容誤差 1.0e-2~1.0e-15ぐらいが妥当

← 時間

← 適当なdtをいれる．ソルバが自動的にdtを大きくとるだろう

← ソルバに渡す変数に本当の変数を代入

# Sample1

```
36  for (i_time = 1;; i_time++)
37  {
38      Bogacki_Shampine54_n_dim(2, u, &t, &dt, Atol, Rtol, kansu_f);
39
40      p = u[0];
41      x = u[1];
42
43      genmitu_p = exp(-gamma * t) * (p0 * cos(omega * t) - (p0 * gamma + x0 * omega_0 * omega_0) / omega * sin(omega * t));
44      genmitu_x = exp(-gamma * t) * (x0 * cos(omega * t) + (p0 + gamma * x0) / omega * sin(omega * t));
45
46      err_p = fabs(p - genmitu_p);
47      err_x = fabs(x - genmitu_x);
48
49      if (err_p > max_err_p)
50          max_err_p = err_p;
51      if (err_x > max_err_x)
52          max_err_x = err_x;
53
54      printf("\nt = %5.5f\n", t);
55      printf("err_p = %15.15g\n", err_p);
56      printf("err_x = %15.15g\n", err_x);
57
58      if (t > 100.0)
59      {
60          break;
61      }
62  }
```

← 時間ループ

← ソルバを使う．uは上書きされ，dtは（大抵大きな値に）変更される．t += dtされる．

← ソルバの変数を本当の変数に代入

↑ 厳密解を計算

← 近似解と厳密解の差を計算

← 近似解と厳密解の差の最大値を計算

← 時間と誤差を出力

← 時間がある程度までいったら，終了．

# Sample1

```
63      printf("\n");
64      printf("max_err_p = %15.15g\n", max_err_p);
65      printf("max_err_x = %15.15g\n", max_err_x);
66      return 0;
67  }
68
```

← 最大の誤差を表示.

# 結果

```
masakazu — Terminal — bash — 49×47
Embedded_Explicit_Runge_Kutta_Multiple.c
main.c
Sample1;cc main.c
Sample1;./a.out

t = 0.00010
err_p = 1.35525271560688e-20
err_x = 1.11022302462516e-16

t = 0.00070
err_p = 2.16840434497101e-19
err_x = 1.11022302462516e-16

t = 0.00430
err_p = 8.67361737988404e-19
err_x =                   0

t = 0.02590
err_p = 9.0205620750794e-17
err_x = 3.33066907387547e-16

t = 0.15550
err_p = 4.11726208682239e-12
err_x = 1.82904802414896e-11
```

← 当初のdtで計算を開始

← 誤差がAtol Rtolの範囲よりも小さいため，ソルバが自動的にdtを大きく変更している．その分誤差は大きくなる.

# Sample1

```
t = 98.94002
err_p = 2.17933397968564e-08
err_x = 8.97587745362499e-09

t = 99.40603
err_p = 2.31347025972037e-08
err_x = 1.54462364925466e-09

t = 99.87222
err_p = 1.95560012636667e-08
err_x = 1.18360740288675e-08

t = 100.33884
err_p = 1.17739422291624e-08
err_x = 1.96918029608112e-08

max_err_p = 2.31347025972037e-08
max_err_x = 2.24361031558873e-08
Sample1;
```

← 最終的にdt=0.5付近まで大きくとっている.

← しかしながら誤差は許容範囲内に収まっている

← 最大誤差も許容範囲内に収まっている

# Atol,Rtol,解きたい問題を様々に変えることによって，dtの選定は大きく変わる.

Atol=Rtol=1.0e-15にすると．．．

```
t = 99.99663
err_p = 1.82145964977565e-16
err_x = 1.4736475928423e-15

t = 100.01550
err_p = 2.24646690138997e-16
err_x = 1.4883927423881e-15

max_err_p = 1.34163513632046e-14
max_err_x = 9.58955137519979e-15
Sample1;
```

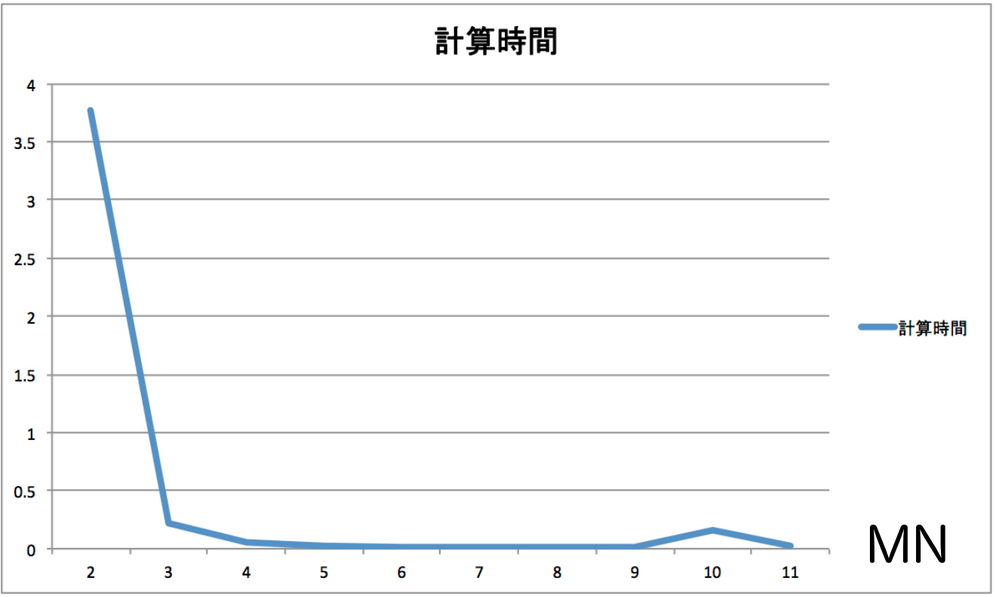
最大誤差も許容範囲内には収まるが,
dt~0.02となってしまう．しかしながら当初のdt=0.0001よりは大きいばかりか，精度も保証されるため，とっても有用.

# Sample2

# 問題や設定は同じまま，Methodを様々に変えてみる．

```
35
36    for (i_time = 1;; i_time++)
37    {
38        Multiple_n_dim(2,2, u, &t, &dt, Atol, Rtol, kansu_f);
39
40        p = u[0];
41        x = u[1];
```

↑　Multiple_n_dimを使うと便利．ここではMethod Number 2を選択




Method Number(MN)と計算時間の関係

MNが2~9までをみると，次数の増加に伴って計算時間が減っているように見るが，実際にはMN=8が最も計算時間が短い結果となっている．MN 10,11もいい線をいっている．

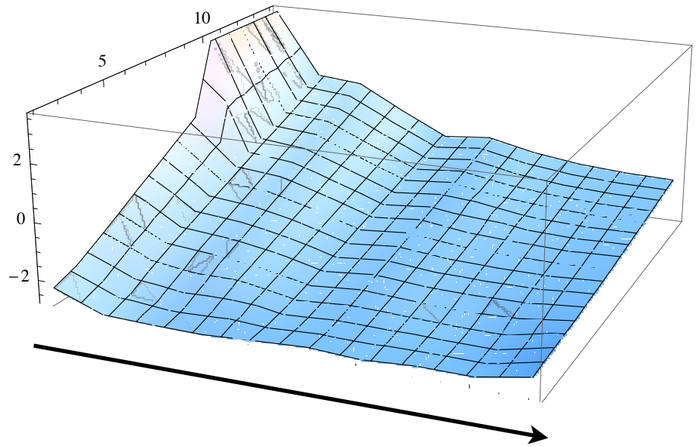
次数の高い公式は計算精度も高いが，そのかわり関数評価の回数も増える．（MN=2では関数評価は3回だが，MN=9は16回！となる．）よってMNが9までは計算精度の向上と伴って，計算コストもかかるので，単に次数の高い公式を選択するだけでなく，各自が解いている問題によってはMethodを変えた方が（場合によっては，低次に）トータルの計算時間は小さくなる場合がある．

作者の経験ではMethod Number 5~8,11は良い成果を上げている．

# 性能評価

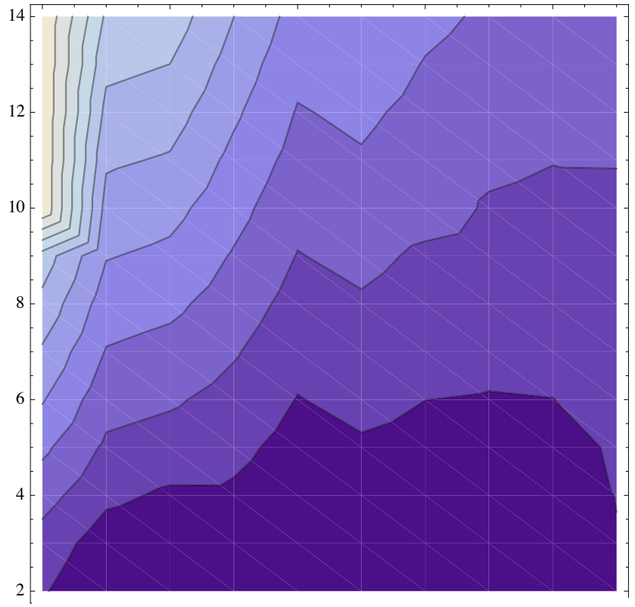
# 性能評価

作者はAtol,Rtol，次数を様々に変えて，どのMethodが最も効率が良いかを検討した.




次数

Atolの指数部



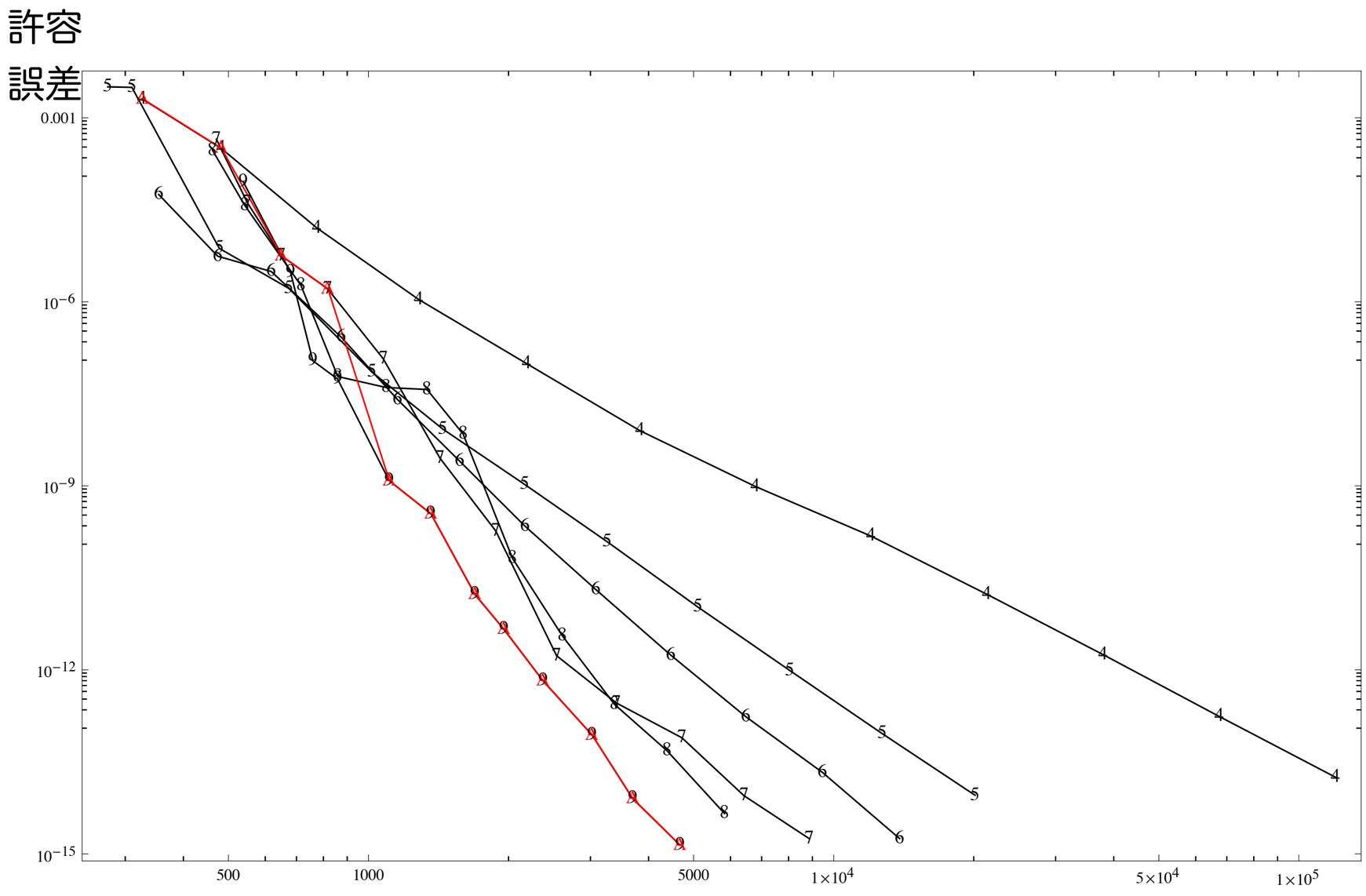

次数

上図は次数(MNではなく)と計算時間(のLog)の関係で，右はその等高線である.
良い結果ほど，色が濃く見えるべきである.

許容誤差を1/10にすると計算時間も増えるが，（当たり前）
どのMethodが最小の計算時間で計算できるかは図のようにはっきりしない.
Method Number 5~8,11を選んでおくのが妥当であろう.

# 性能評価

汎用計算ソフトMathematicaは常微分方程式のソルバとして，本ソルバと同様なMethodを実装している．そこで，他のソフトでの埋め込み型ルンゲ・クッタ法の性能評価を参考にしてみる．




Mathematicaにて同様なMethodを用いた場合(ブラッセレーターモデルを数値計算している)，左図のような結果を出している．

これを見る限り，許容誤差が緩い時は， 次数6のMethodが,許容誤差が厳しい時は次数9のMethodが(MN=9)が優秀であるといえる．

ただし，問題によってこれらのカーブは異なるので，どのMethodが最も良いかは，はっきりしない．
やはり，各自が探るのが一番早い．

基本的にここまで読めば，使いこなす事はできるはず．以降ではよりソルバを使いこなす方法を説明する．

# Sample3

## ここでは，ソルバの性能をさらに引き出す方法について説明する.

本ソルバは利用者の負担を減らすために，計算上必要なワーキング配列を自動的にとっている.
しかしながら，そのためにソルバの速度を少なくとも20%以上犠牲にしている．そこで，以下では前もって，ワーキング配列を取ることによって性能向上させることができるやり方である．Sample1のmain.cに以下の変更を加える.

```
double          u[2],*work_u;
double          p0 = 0, x0 = 1;
double          p = p0, x = x0;
double          genmitu_p, genmitu_x;
double          err_p = 0.0, err_x = 0.0;
double          max_err_p = 0.0, max_err_x = 0.0;

double          Atol = 1.0e-8;
double          Rtol = 1.0e-8;
double          t = 0;
double          dt = 0.0001;

u[0] = p;
u[1] = x;

work_u = Bogacki_Shampine54_n_dim_Initial_Setting(2);

for (i_time = 1;; i_time++)
{
    Bogacki_Shampine54_n_dim_core(2, u, &t, &dt,work_u, Atol, Rtol, kansu_f);
```

← ワーキング配列のポインタ

ワーキング配列の領域をとる

“_core”を付加し，Atolの前にwork_uを追加するだけ.

本ケースでは，50%速度が向上した.

# Sample4
# Sample3と同値な書き方.

```
35
36      work_u = Multiple_n_dim_Initial_Setting(5,2);
37
38      for (i_time = 1;; i_time++)
39      {
40          Multiple_n_dim_core(5,2, u, &t, &dt,work_u, Atol, Rtol, kansu_f);
41
```

“Multiple_”に続き，上記の様にプログラムを書けば，自動的にワーキング配列が設定される.
プログラムの終了時にはfreeをすること.

# Sample5

本ソルバは決められた許容誤差の範囲内で，ソルバが適応的にdtを選択することによって，数値計算の大幅な高速化を可能にしている．しかしながら，問題によってはdtを定数扱いにしたいこともある．そのような時は単に以下のようにプログラムを変更すれば良い．Sample5は(Sample4を変更して)以下の様になる．

```
for (i_time = 1;; i_time++)
{
    Multiple_n_dim_core(5,2, u, &t, &dt,work_u, Atol, Rtol, kansu_f);
    dt = 0.0001;

    p = u[0];
    x = u[1];
```

つまり，ふさわしいdtを破棄し，ソルバの直後でdt=0.0001などとすれば良いのである．当然ながら，この様なやり方ではAtol,Rtolで定めた許容誤差は守られていない可能性があることに注意すべきである．

# おわりに

以上までで，ソルバの説明を終わる．

問題にもよるが，Sample4を変更して，各自の解きたい問題を解くのが高速でかつ取っつきやすいだろう．

秋山正和謹製のmybasicはVersion8.0以降では，本ソルバを内包しているので，そちらを入手し，インストールすれば，本ソルバを使う事ができる．「秋山正和　mybasic　とかでググれ」

本ソルバは誰でも再配布して良い．ただし，作者にメールしてくれれば，実態把握ができてうれしい.また，バグの報告もしてほしい．