レポート課題 No.1

川口廣伊智
学籍番号:051715223

2017/06/06

0　レポートについての注意

0.1　各課題の解答の構成について

まず課題の解釈をし、解答するためのプログラム (の一部) を記載した。次に得られた結果をグラフにして記載した。考察すべき内容があった場合は簡単な考察も付けた。すべての課題に考察がついているわけではない。

0.2　プログラムについて

各課題についてその計算を行うための本質的な部分のプログラムを抜粋して記載しているが、抜粋であるためプログラムの全体像が見えず理解しづらい可能性がある。分かりやすくするため配慮はしているが、その点は留意されたい。

1　基本課題 EX0-1

1.1　課題概要

fibonacci 数列を求める問題。60 番目まで求めればいいので $O(n)$ のアルゴリズムのプログラムによって計算した。以下は整数 n を受け取り n 番目のfibonacci 数列を double 型で返す関数のプログラムである。

ソースコード 1　fibonacci 数列を計算する関数

```
1  double fibonacci(int n){
2    double answer, temp_1, temp_2;
3    int i;
4    temp_1 = 1.0;
5    temp_2 = 0.0;
6    if(n == 0){
7      answer = 0;
8    }else if(n == 1){
9      answer = 1;
10   }else{
11     for(i = 1;i < n;i++){
12       answer = temp_1 + temp_2;
13       temp_2 = temp_1;
```

```
14        temp_1 = answer;
15      }
16    }
17    return answer;  /* output the nth of fibonacci series */
18  }
```

1.2 結果

fibonacci 数列の 20、30、40、50、60 番目を計算した結果を表にまとめた。

表 1　fibonacci 数列の値

n	a_n
20	6765
30	832040
40	102334155
50	12586269025
60	1548008755920

1.3 考察

fibonacci 数列を計算するアルゴリズムはいくつかあるが、ここでは配列または単純な繰り返しを用いたもの、再帰計算によるもの、行列計算を用いたものの 3 種類のアルゴリズムについて簡単に考察する。また以下のような考察により今回 fibonacci 数列を得るのに使用すべきアルゴリズムは単純な繰り返しを用いたものであったと思われる。単純ですぐプログラムが作れるうえ、$n = 60$ 程度なら時間もかからないからだ。

1.3.1 配列または単純な繰り返しを用いたアルゴリズム

まず配列を用いたアルゴリズムについて。これは最も単純なアルゴリズムで a_nを求めるために配列 $a[n]$ を用意する。$a[0] = 0$、$a[1] = 1$ を与えておき、$n = 2$ から順に $a[n] = a[n-1] + a[n-2]$ の計算を繰り返すことで $a[n]$、すなわち a_nを得ることが出来る。n に関するループが一つなので時間計算量は $O(n)$ である。一方、長さ n の配列を用いるので空間計算量が $O(n)$ と割を食う。空間計算量に関しては配列を用いずにソートをうまく取り入れることにより $O(1)$ と抑えることが出来る。

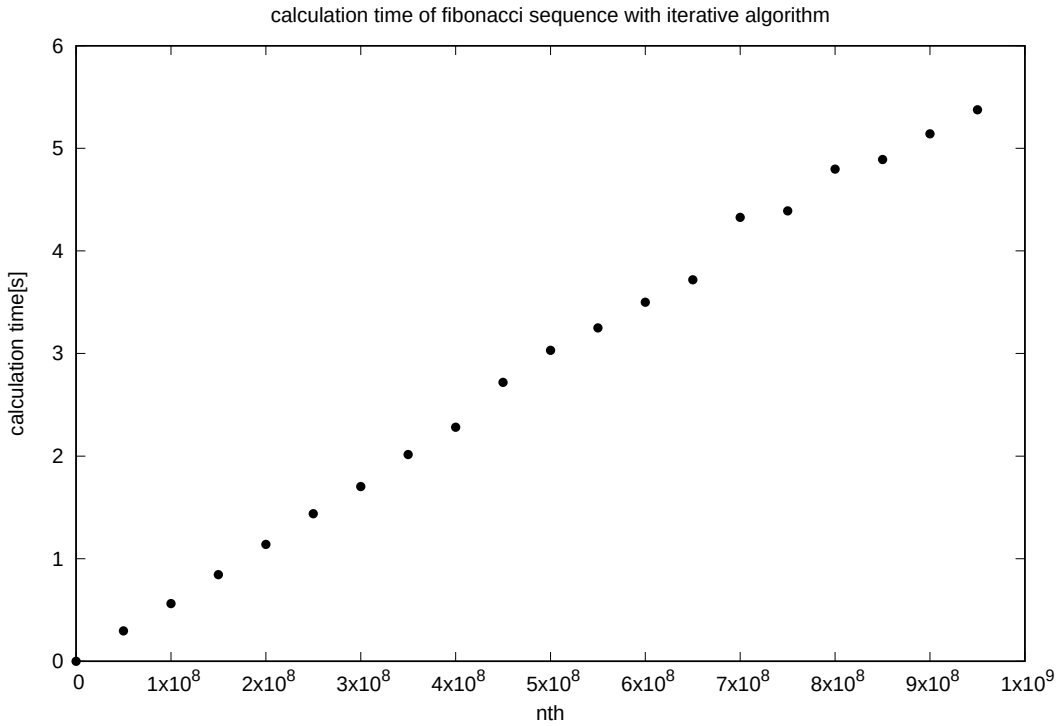

図 1　単純な繰り返しを用いた場合の n 番目の fibonacci 数列を計算するのにかかった時間のグラフ。横軸が n 番目、縦軸はその n 番目の数列を計算するのにかかった時間を表している。時間計算量が $O(n)$ であること分かる。

1.3.2　再帰計算を用いたアルゴリズム

次に再帰計算を用いたアルゴリズムについて。アルゴリズムは $n \geq 2$ について、n を引数として $a_{n-1}+a_{n-2}$ を返すような関数を用意し、この関数の中でまた関数が呼び出されるという再帰的な構造を持っている。この構造のため、同一の計算が複数回行われ、時間計算量は $O(\alpha^n)$ である。($\alpha > 1$、具体的には$\alpha = 1.618$ である。)n のべき乗のオーダーなので n が大きい計算を行うときにこのアルゴリズムは使用するべきではない。

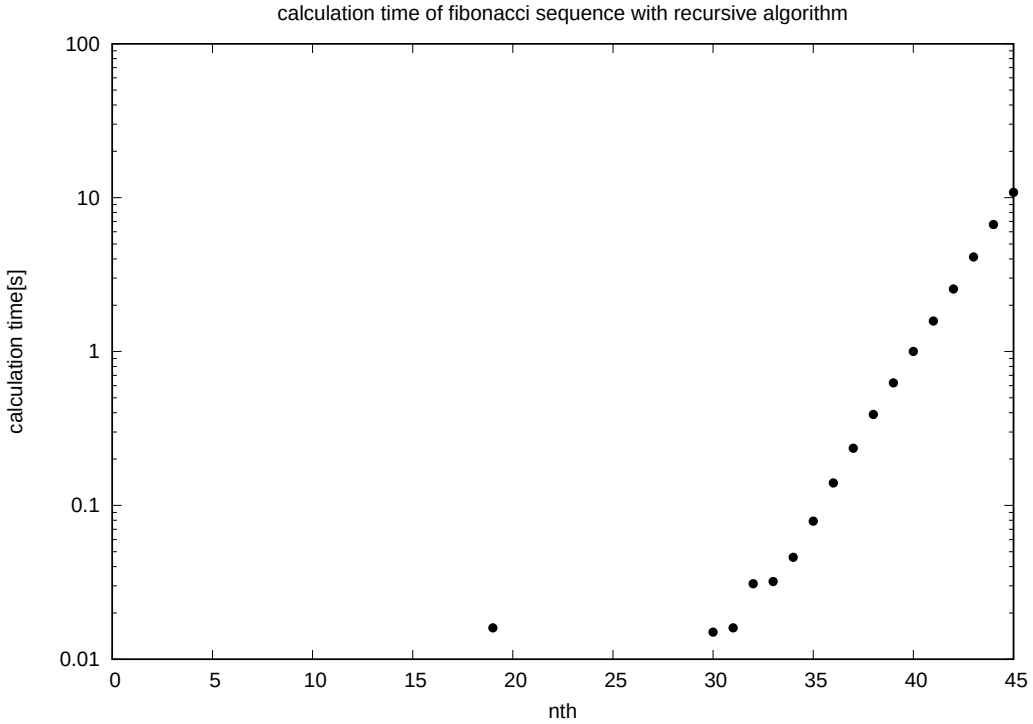

図 2　再帰計算を用いた場合の n 番目の fibonacci 数列を計算するのにかかった時間のグラフ。横軸が n 番目、縦軸はその n 番目の数列を計算するのにかかった時間を表していて対数表示されている。時間計算量が $O(\alpha^n)$、$(\alpha > 1)$ であることが分かる。

この結果から分かるように $n = 60$ を求めるのにさえおよそ一万秒かかることが分かる。そのため今回の課題でこのアルゴリズムは使うべきでない。

1.3.3　行列計算を用いたアルゴリズム

最後に行列計算を用いたアルゴリズムについて。まず、ベクトル $v_k = \begin{pmatrix} a_{k+1} \\ a_k \end{pmatrix}$ と行列 $Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ を用意する。これより直ちに $v_{k+1} = Qv_k$であることがわかる。すなわち $v_{n+1} = Q^n v_0$である。なので a_nを計算するためには Q^nを計算し $v_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ に作用させて得られたベクトルの第 2 成分を見ればよい。このアルゴリズムの本質は fibonacci 数列を計算するのに Q^n の計算に帰着できるところである。この計算では素直に Q を n 回かけることはなく Q の性質を利用して計算量を大幅に減らすことが出来る。結局、時間計算量は $O(\log n)$ である。計算コストの面では良いアルゴリズムだがソースコードから作るとなるとやや複雑で面倒だと個人的に思う。

2　基本課題 EX1-1

2.1　課題概要

与えられた関数の微分係数を 2 点差分法、3 点差分法のアルゴリズムを用いて計算し、得られた値と真値との誤差の振る舞いを調べる課題であった。刻み幅を小さくとればとるほど誤差が小さくなる結果が期待される。

次のプログラムは関数 (今回の場合 $\sin x$) と、ある値 (今回の場合 0.3π) と、刻み幅を受け取る。そして、その関数のその値での微分係数を与えられた刻み幅で 2 点差分により求め、出力する関数である。関数は引数と

して受け取ってない。

ソースコード 2　2 点差分法

```
1 double differentiation(double x, double h){
2
3         return (func(x + h) - func(x)) / h;
4 }
```

その 3 点差分の場合である。

ソースコード 3　3 点差分法

```
1 double differentiation(double x, double h){
2
3         return (func(x + h) - func(x - h)) / (2 * h);
4 }
```

2.2　結果

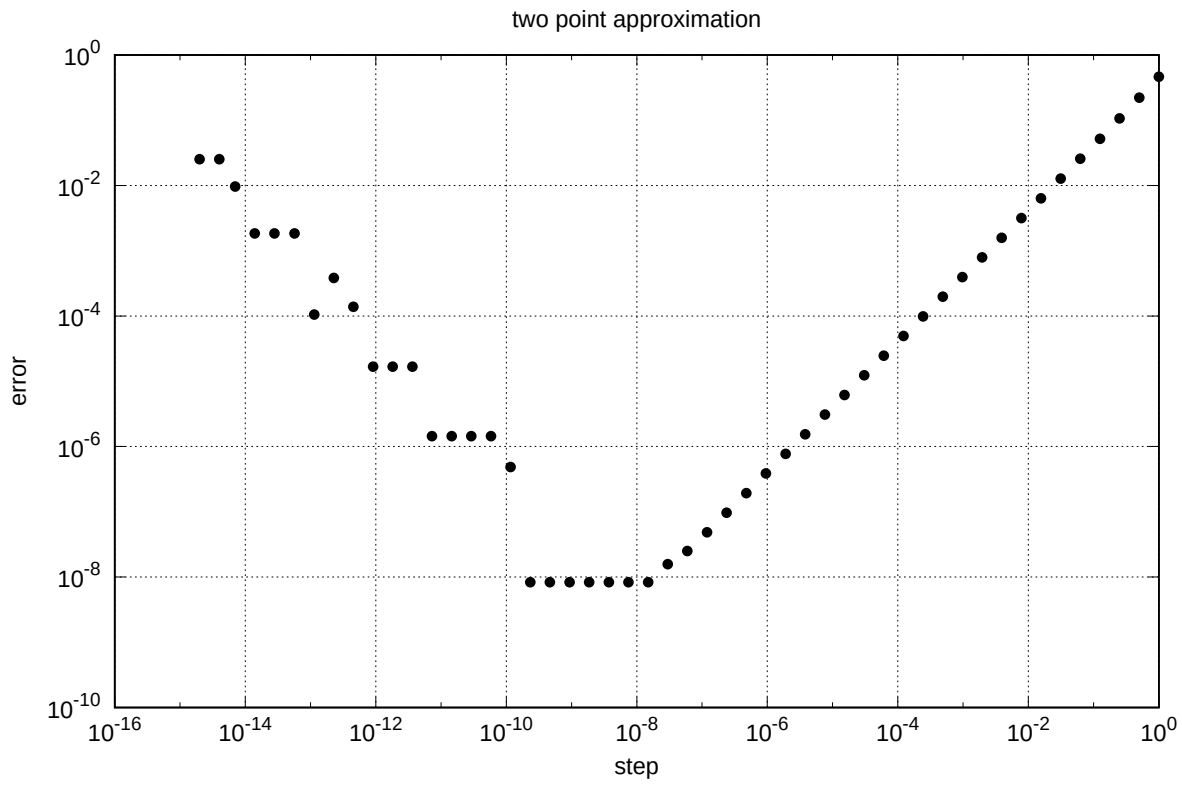

図 3　$h = 10^{-8}$ ぐらいまでは誤差が $O(h)$ であることが分かる。

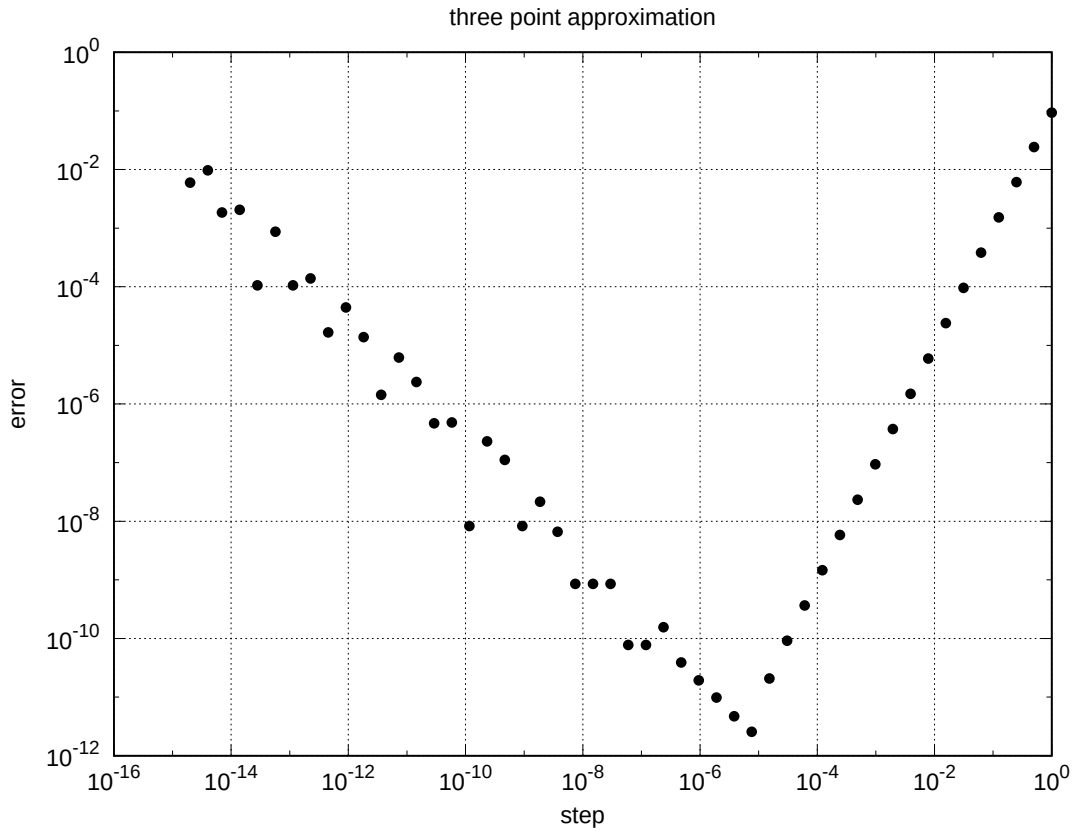

図 4　$h = 10^{-5}$ ぐらいまでは誤差が $O(h^2)$ であることが分かる。

2.3　考察

2.3.1　誤差の振る舞いについて

h を小さくしていったときに二点差分の計算結果では $h = 10^{-8}$、三点差分の計算結果では $h = 10^{-5}$ の前後で誤差の振る舞いが変化している。この結果について少し考察する。二点差分、三点差分のいずれも同じ根拠に基づいているので二点差分についてのみ考えれば十分である。まず、刻み幅が $h = 10^{-8}$ より大きい範囲での誤差の振る舞いについて。この範囲では打切り誤差が全体の誤差のリーディングオーダーになっており二点差分の場合、打切り誤差のオーダーは $O(h)$ であり結局これが全体としての誤差のオーダーとなる。一方、刻み幅が $h = 10^{-8}$ より小さい範囲での誤差の振る舞いについて。この範囲ではうって変わって桁落ち誤差が全体の誤差のリーディングオーダーとなる。桁落ち誤差とは今回の場合、h を小さくとればとるほど $f(x+h)$ と $f(x)$ の差が小さくなり、上位桁が次々一致していき、この差をとることで有効桁数が減ってしまうことによる誤差を指す。このため、刻み幅を小さくとればとるほど計算の精度が上がるかと思いきやそうでもないのである。二次方程式の解を求める際にも同様の桁落ち誤差が生じるがこのときにはうまく対処する方法が存在する。しかし今回のように微分係数を差分で近似する際にはうまい対処法があるのだろうか？

3　基本課題 EX1-2

3.1　課題概要

Newton 法により $\sqrt[3]{10}$ を計算する課題である。解答は初期値を $x = 3.0$ にして方程式 $x^3 - 10 = 0$ を解くことで $\sqrt[3]{10}$ を求めるようになっている。そのためのプログラムは次のようである。

ソースコード 4　ニュートン法

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define EPS pow(10.0,-15)
5  #define KMAX 100
6
7  double func(double x); //関数
8  double derf(double x); //その導関数
9
10 int main()
11 {
12   int k=1;
13   double x, d;
14   x = 3.0;
15   printf("%d␣%20.15lf\n", 0, x - 2.154434690031884); //初期の真値との差を出力
16   do{
17     d = -func(x) / derf(x);
18     x = x + d;
19     printf("%d␣%20.15lf\n", k, x - 2.154434690031884); //反復回数と真値との差
   をループ毎に出力
20     k++;
21   }while( fabs(d) > EPS && k < KMAX); //このように条件づけることでループが終わらなく
   なること防ぐ
22
23   if( k == KMAX ){
24     printf("The␣answer␣could␣not␣be␣found.\n" );
25   }
26   return 0;
27 }
28
29 double func( double x){
30   return ( pow(x, 3.0) - 10.0 );
31 }
32 double derf(double x){
33   return( 3.0 * pow(x, 2.0) );
34 }
```

3.2 結果

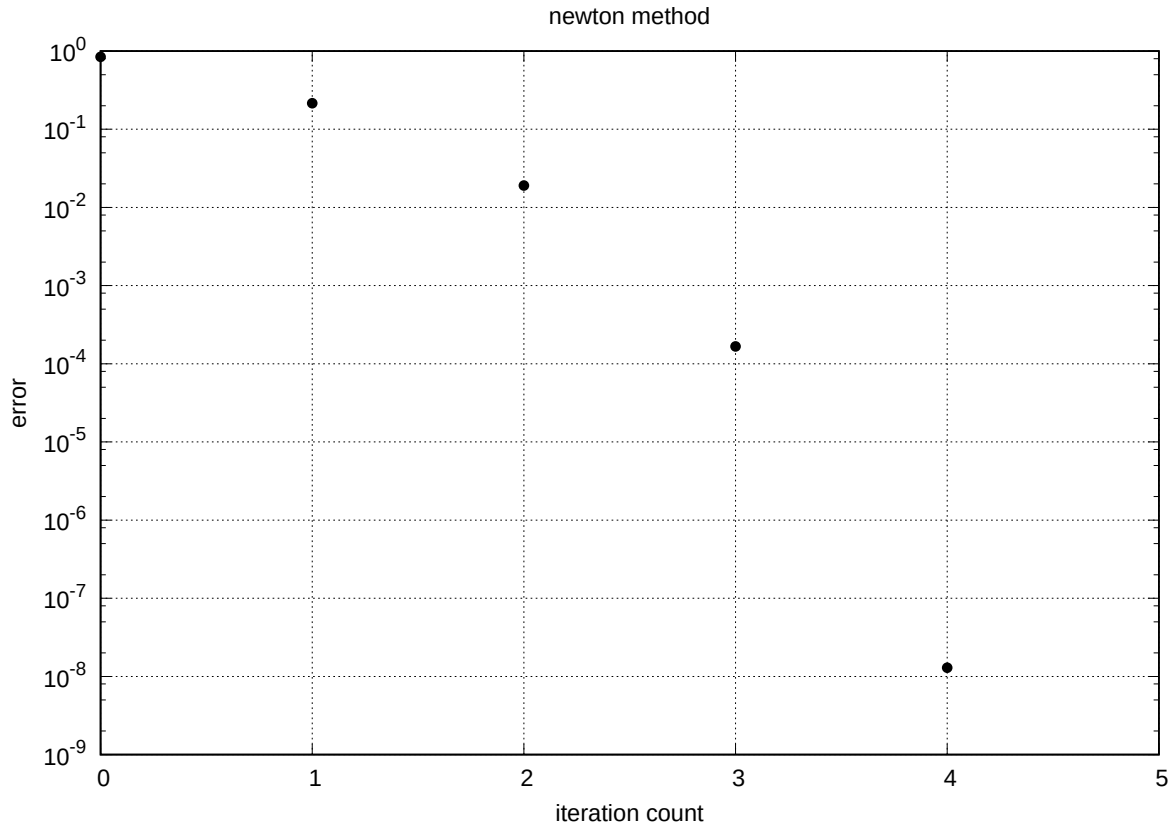

図 5　ニュートン法による $\sqrt[3]{10}$ の計算値と真値との差の反復回数による変化。反復を重ねるごとに誤差がだいたい 10^0、10^{-1}、10^{-2}、10^{-4}、10^{-8} となっており、二次収束であることが分かる。

4 基本課題 EX2-1

4.1 課題概要

Euler 法によって与えられたパラメータ、初期条件での減衰振動の解を計算する課題である。次のプログラムは加速度の関数 func()、時間 t、位置 x[1]、速度 x[2]、 刻み幅 h を受け取り Euler 法を実行し次のステップの位置と速度を計算する関数 euler と、実際の減衰振動の場合の位置微分と速度微分を計算する関数である。euler 関数で func としてこの damped などの関数を引数として受け取る。

ソースコード 5　Euler 法

```
1 void euler(void (*func)(), double t, double x[], double h)
2 {
3   double k1[N];
4   func(k1, t, x); //この関数を通してｋ１を計算する
5   x[0] = x[0] + h * k1[0]; //次のステップの位置の値を計算
6   x[1] = x[1] + h * k1[1]; //次のステップの速度の値を計算
7 }
8
```

```
9  void damped(double k[], double t, double x[])
10 {
11   double spring_const = 2.0;
12   double kappa = 0.2;
13
14   k[0] = x[1];
15   k[1] = -spring_const * x[0] - kappa * x[1];
16 }
```

また、精度の比較のため理論値を求めないわけにはいかない。が、うまく計算が出来ずグラフに参照線として描いてある理論線はおそらく正しいものではない。しかし、真の解ともそれほどずれてもいないなはずなので収束すべき対象として描くことにした。

4.2 結果

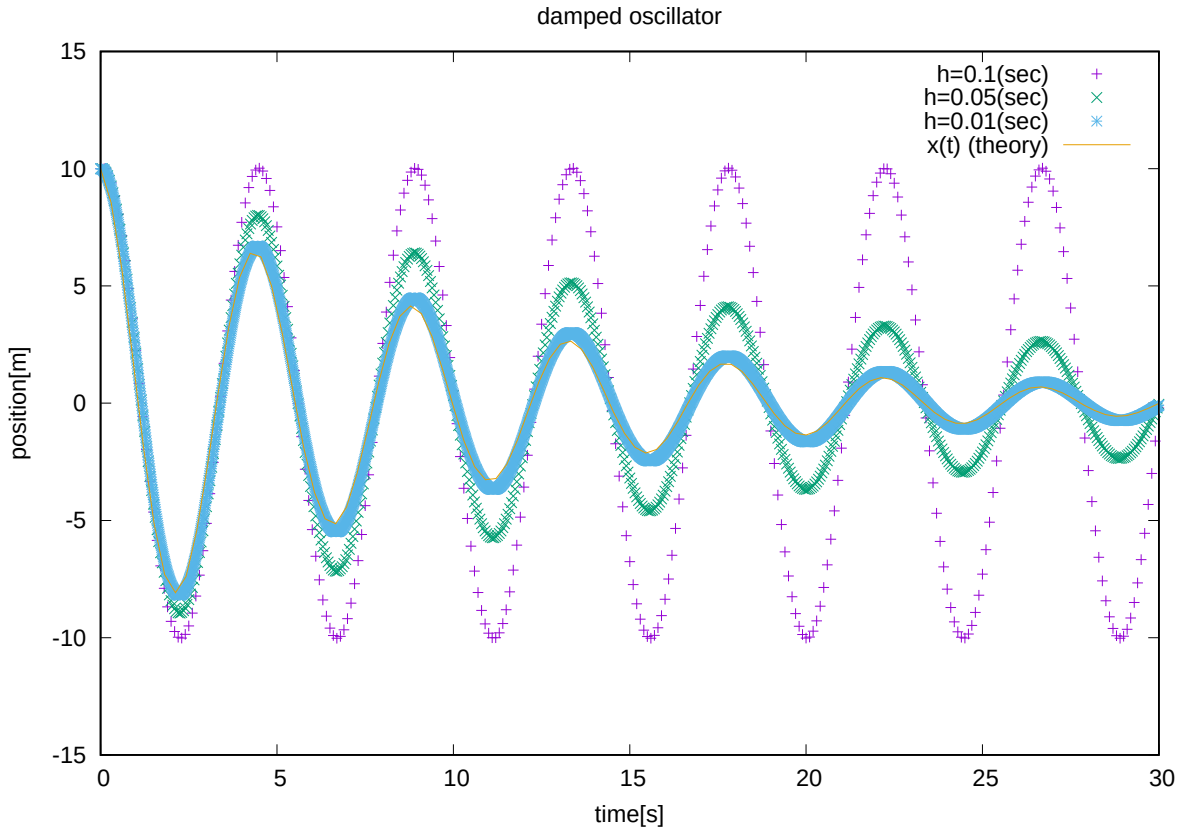

図 6　Euler 法によって計算した減衰振動の解。刻み幅を小さくとればとるほど理論値に近づいていく様子が分かる。

4.3 考察

4.3.1 誤差の振る舞いについて

Euler 法で計算した減衰振動の 30 秒後の位置の値の誤差が刻み幅の大きい領域で $O(h)$ で振舞わないことが分かった。これは刻み幅が大きいとテイラー展開が大雑把になり、しかも真の解が振動的であるため誤差のたまり方が悪い場合があり $O(h)$ に”見えなくなる”ためであろう。一方、刻み幅が小さい領域では誤差は $O(h)$ で振舞っていた。また別途、自由落下に対しても何秒後かの位置の値の精度を調べた。この時は誤差はぴった

り $O(h)$ で振舞った。他の単調増加する解が得られる場合でも私が調べた範囲では h が大きい領域でも $O(h)$ で振舞っていた。

結論としては、テイラー展開の打切り誤差は $O(h^2)$ だが振動する解を数値計算する場合は刻み幅が大きいと誤差の積もり方が悪く全体としての誤差が $O(h)$ に振舞っているように見えないということだ。

5 基本課題 EX2-2

5.1 課題概要

課題 2-1 で行った計算を中点法、4 次の Runge-Kutta 法でも同様に行う課題。次のプログラムは Euler 法、Euler 法のプログラムと同様の構造をしており、4 次の Runge-Kutta 法により次のステップの位置と速度を計算する関数である。

ソースコード 6　中点法

```
1  void mid_point(void (*func)(), double t, double x[], double h)
2  {
3    double k1[N], k2[N], k3[N], k4[N], f[N];
4    func(k1, t, x);
5    f[0]=x[0] + h * k1[0] / 2.0;
6    f[1]=x[1] + h * k1[1] / 2.0;
7    func(k2, t + h / 2.0, f);
8    x[0] = x[0] + h * k2[0];
9    x[1] = x[1] + h * k2[1];
10 }
```

次のプログラムは Euler 法、中点法のプログラムと同様の構造をしており、4 次の Runge-Kutta 法により次のステップの位置と速度を計算する関数である。

ソースコード 7　4 次の Runge-Kutta 法

```
1  void rk4(void (*func)(), double t, double x[], double h)
2  {
3    double k1[N],k2[N],k3[N],k4[N],f[N];
4    func(k1, t, x);
5    f[0]=x[0] + h * k1[0] / 2.0;
6    f[1]=x[1] + h * k1[1] / 2.0;
7    func(k2, t + h / 2.0, f);
8    f[0]=x[0] + h * k2[0] / 2.0;
9    f[1]=x[1] + h * k2[1] / 2.0;
10   func(k3,t + h / 2.0, f);
11   f[0]=x[0] + h * k3[0];
12   f[1]=x[1] + h * k3[1];
13   func(k4, t + h, f);
14   x[0] = x[0] + h / 6.0 * (k1[0] + 2.0 * k2[0] + 2.0 * k3[0] + k4[0]);
15   x[1] = x[1] + h / 6.0 * (k1[1] + 2.0 * k2[1] + 2.0 * k3[1] + k4[1]);
16 }
```

5.2　結果

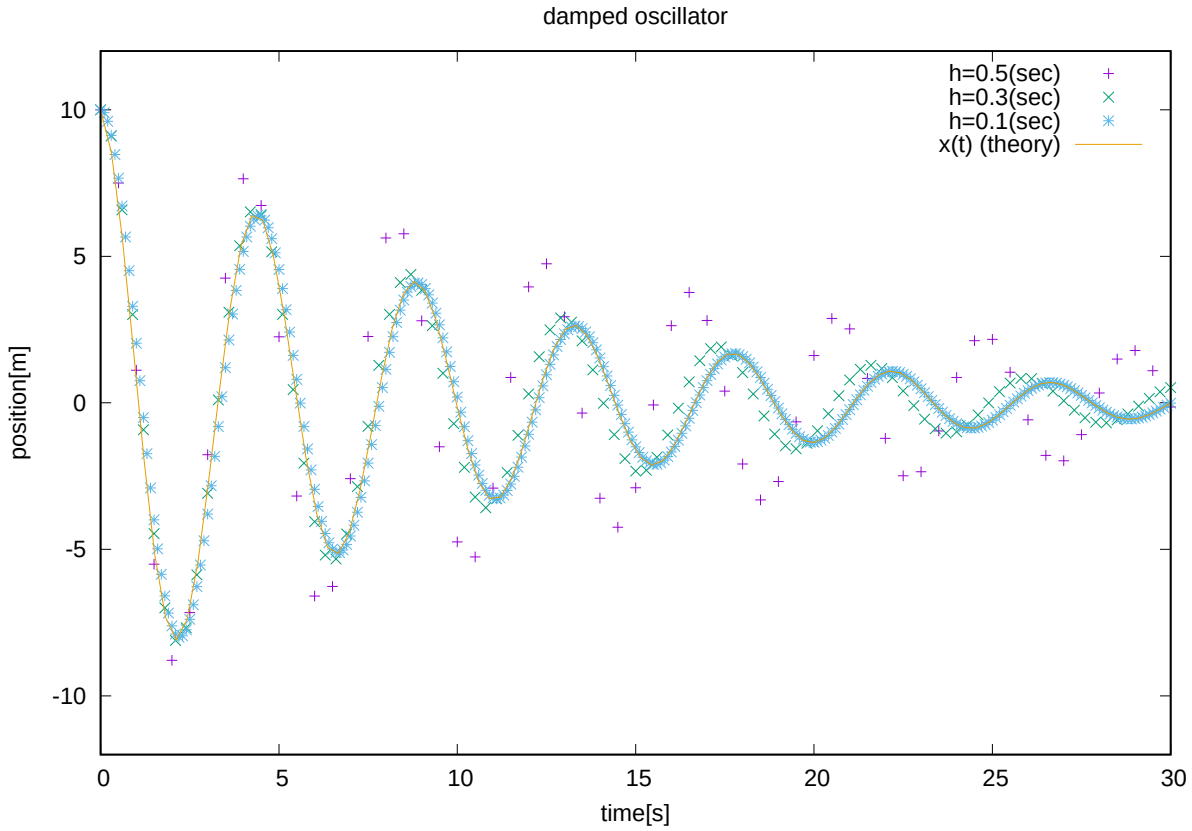

図 7　中点法によって計算された減衰振動の解。Euler 法に比べて早く理論解に収束していることが分かる。

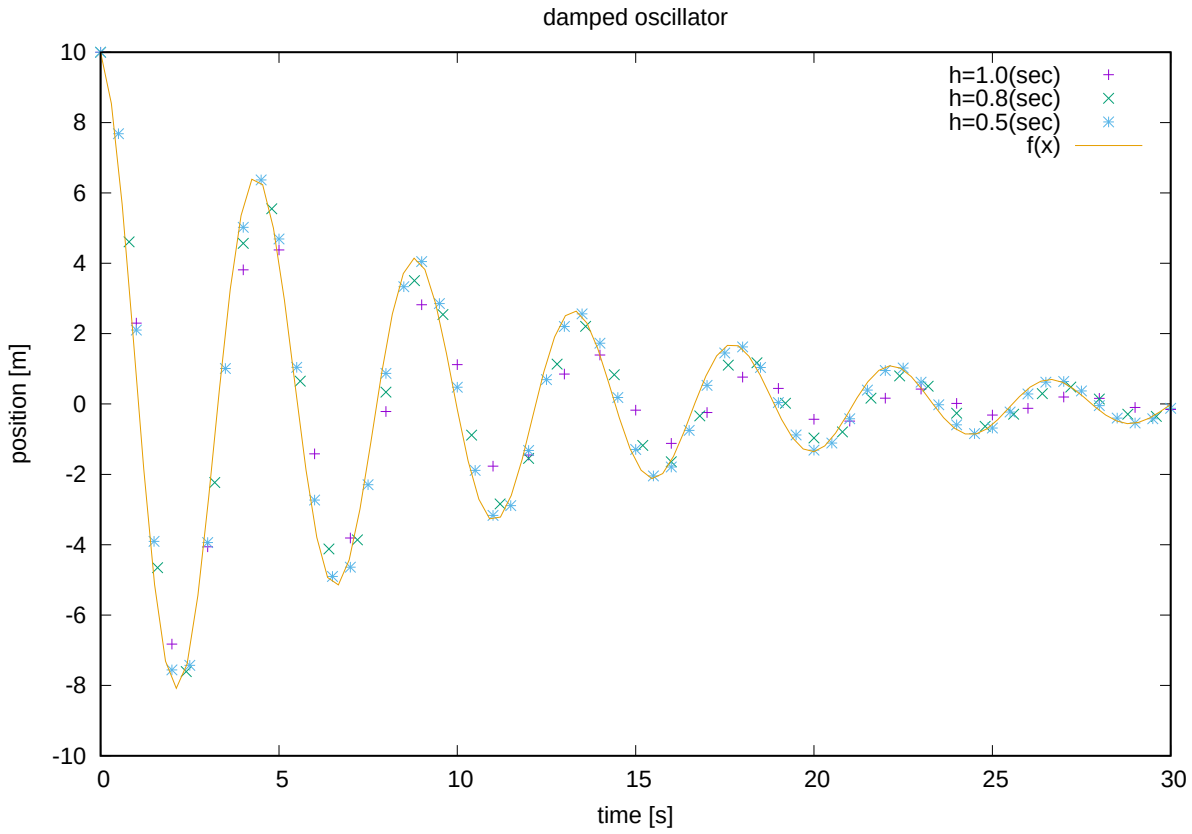

図 8　4 次の Runge-Kutta 法によって計算された減衰振動の解。Euler 法、中点法に比べてもより早く理論解に収束している。

6　基本課題 EX2-3

6.1　課題概要

摩擦がない場合の全エネルギーの時間変化を Euler 法、Runge-Kutta 法、Symplectic 法でエネルギーを計算してそれぞれ求めた。次のプログラムは課題 2-2 の中点法などで行ったのと同様に Symplectic 法により次のステップの位置と速度を計算する関数である。

ソースコード 8　4 次の Runge-Kutta 法

```
1 void symplectic(void (*func)(), double t, double x[], double h)
2 {
3   double k1[N];
4   func(k1, t, x);
5   x[0] = x[0] + h * x[1];
6   x[1] = x[1] - h * 2.0 * x[0];
7 }
```

6.2　結果

二つのプロットを載せているが一つは 30 秒までのエネルギーの時間変化を Euler 法、4 次の Runge-Kutta 法、Symplectic 法で計算したものを重ねてプロットしている。もう一つは 10^6 秒までのエネルギーの時間変

化を 4 次の Runge-Kutta 法、Symplectic 法で計算したものを重ねてプロットしている。

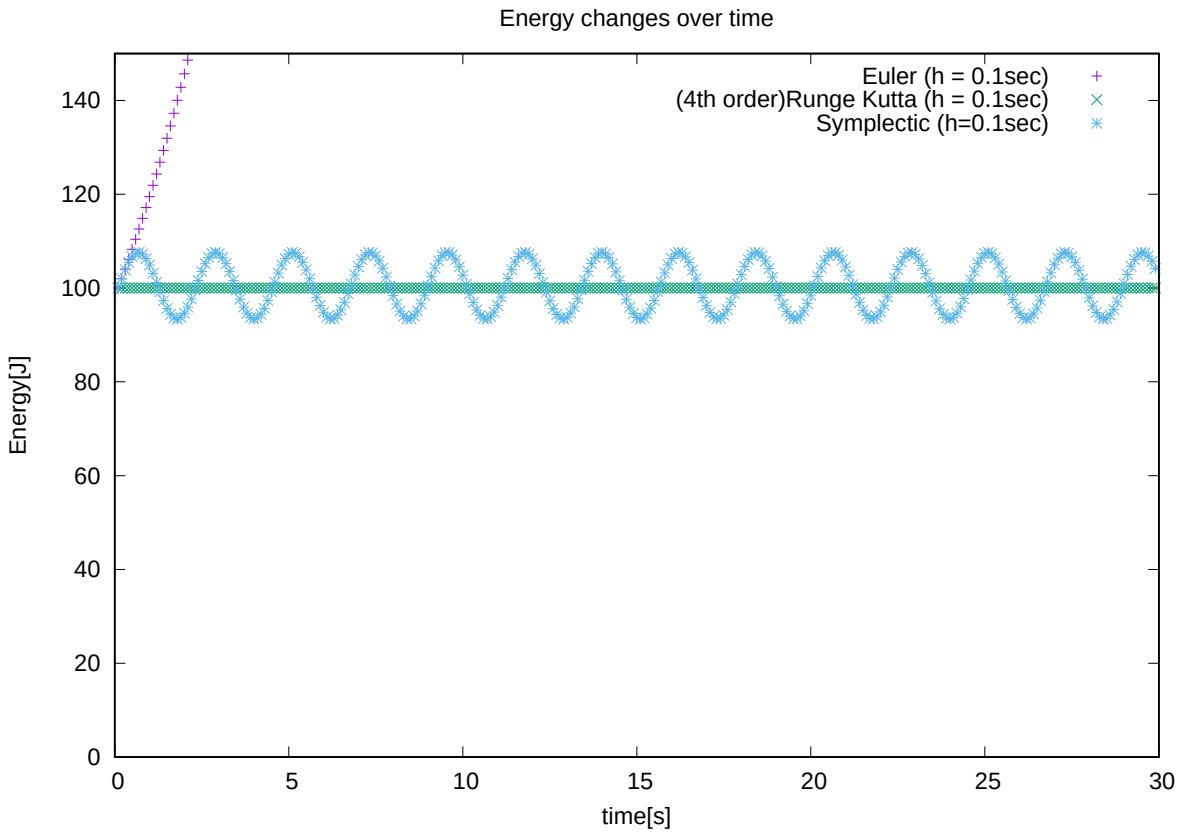

図 9　Euler 法、4 次の Runge-Kutta 法、Symplectic 法によって計算されたエネルギーの時間変化。時間の刻み幅は 0.1sec。Euler 法によって計算されたエネルギーは早い段階で理論値から遠ざかるが、Runge-Kutta 法によるものはこの時間範囲では理論値と一致しているように見える。Symplectic 法によるものは理論値の周りで振動している。

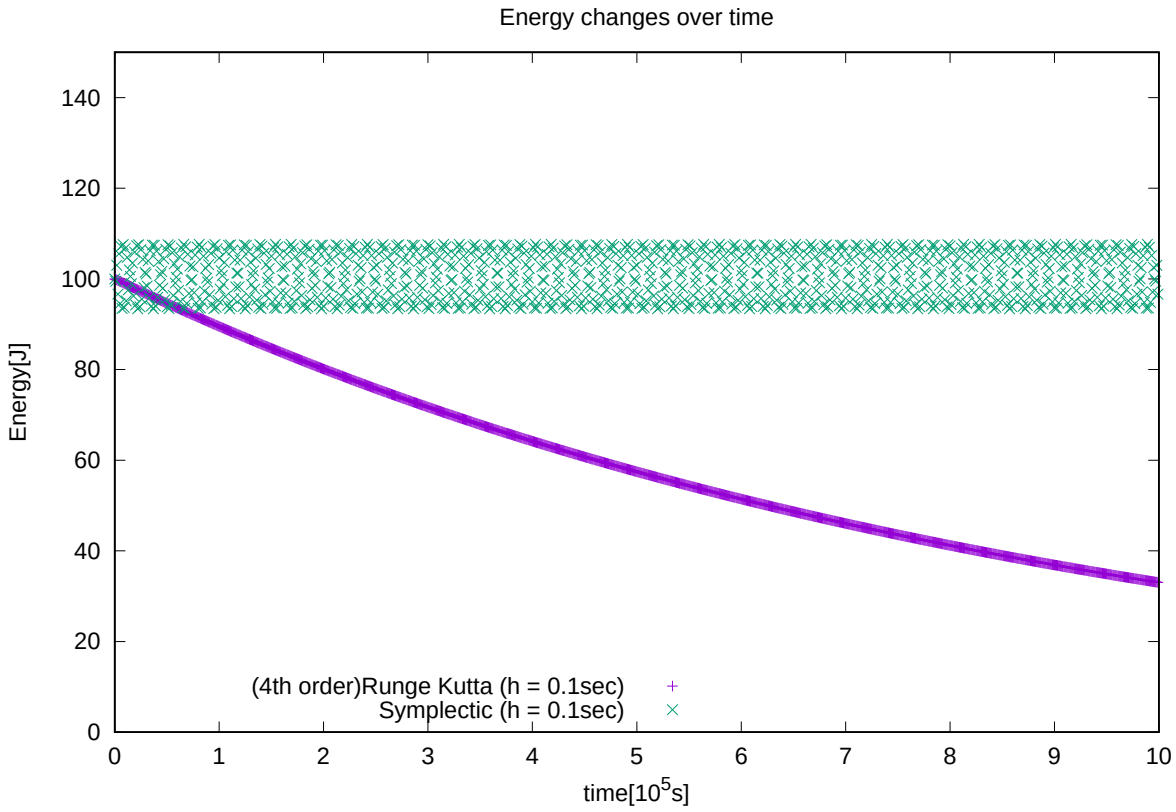

図 10　4 次の Runge-Kutta 法と Symplectic 数値積分法によって計算されたエネルギーの時間変化。時間の刻み幅は 0.1sec。Runge-Kutta 法によって計算されたエネルギーは長時間たつとやがて減少していく様子が見られる。一方、Symplectic 数値積分法によるものは時間がいくらたっても理論値の周りを振動しているだけである。

6.3　考察

今回用いた Euler 法、4 次の Runge-Kutta 法、Symplectic 数値積分法のいずれもエネルギーの振る舞いが異なっていて興味深く、かつ講義でも詳しく触れられなかったのでここで少し考察を試みる。ただし課題で与えられたパラメータを変えて (無次元化して) ハミルトニアンを

$$H = \frac{1}{2}p^2 + \frac{1}{2}q^2$$

と書くことにする。これより、運動方程式は

$$\begin{cases} \dot{p} = -q \\ \dot{q} = p \end{cases}$$

となる。

6.3.1　Euler 法におけるエネルギーの時間変化

Euler 法の場合、$n+1$ ステップ目の (p_{n+1}, q_{n+1})、n ステップ目の (p_n, q_n) の関係は行列で表すと

$$\begin{pmatrix} p_{n+1} \\ q_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -\Delta t \\ \Delta t & 1 \end{pmatrix} \begin{pmatrix} p_n \\ q_n \end{pmatrix}$$

となる。ただし Δt は 1 ステップの時間間隔を表す。この時の 1 ステップでの時間発展行列を

$$T = \begin{pmatrix} 1 & -\Delta t \\ \Delta t & 1 \end{pmatrix}$$

と表す。n ステップ目のエネルギーを E_nとすると

$$E_n = \frac{1}{2}(p_n^2 + q_n^2) = \frac{1}{2}\begin{pmatrix} p_n & q_n \end{pmatrix}\begin{pmatrix} p_n \\ q_n \end{pmatrix}$$

なので

$$\begin{aligned} E_{n+1} = \frac{1}{2}(p_{n+1}^2 + q_{n+1}^2 &= \frac{1}{2}\begin{pmatrix} p_n & q_n \end{pmatrix} T^t T \begin{pmatrix} p_n \\ q_n \end{pmatrix} \\ &= \frac{1}{2}\begin{pmatrix} p_n & q_n \end{pmatrix}\begin{pmatrix} 1+(\Delta t)^2 & 0 \\ 0 & 1+(\Delta t)^2 \end{pmatrix}\begin{pmatrix} p_n \\ q_n \end{pmatrix} \\ &= E_n(1+(\Delta t)^2) \end{aligned}$$

となることが分かる。これより

$$E_n = E_0(1+(\Delta t)^2)^n$$

なので Euler 法の場合はエネルギーが指数関数的に増加していくことが理解できる。

6.3.2 4 次の Runge-Kutta 法におけるエネルギーの時間変化

Euler 法と同様の考察から 4 次の Runge-Kutta 法の場合の時間発展行列は

$$T = \begin{pmatrix} 1-\frac{1}{2}(\Delta t)^2 + \frac{1}{24}(\Delta t)^4 & -\Delta t + \frac{1}{6}(\Delta t)^3 \\ \Delta t - \frac{1}{6}(\Delta t)^3 & 1-\frac{1}{2}(\Delta t)^2 + \frac{1}{24}(\Delta t)^4 \end{pmatrix}$$

と求めることが出来る。ゆえに

$$E_{n+1} = E_n(1 - \frac{1}{72}(\Delta t)^6 + \frac{1}{576}(\Delta t)^8)$$

となることが分かる。なので

$$E_n = E_0(1 - \frac{1}{72}(\Delta t)^6 + \frac{1}{576}(\Delta t)^8)^n$$

で、エネルギーが指数関数的に減少していくことが理解できる。

6.3.3 Symplectic 数値積分法におけるエネルギーの時間変化

これは講義資料にも少し記述があったので簡単に。この計算方法ではエネルギーの保存は成り立たないが

$$\frac{1}{2}p^2 + \frac{1}{2}q^2 + \frac{\Delta t}{2}pq$$

という量が保存する。つまり一般にエネルギーは $O(\Delta t)$ の範囲で保存し、特に調和振動子の場合はエネルギーが $O(\Delta t)$ の範囲で振動子の振動数の 2 倍で振動することが分かる。

参考文献

[1] 増原英彦 + 東京大学情報教育連絡会『情報科学入門 Ruby を使って学ぶ』(東京大学出版会, 2010)

[2] `http://www.cp.cmc.osaka-u.ac.jp/~kikuchi/texts/conservation.pdf`