連続系アルゴリズム
レポート課題1

経済学部金融学科3年
07-152042　松下 旦
連絡先：mail@myuuuuun.com

平成28年1月17日

1 月 17 日: 問題 1 の Taylor 展開が不正確だったので式を訂正（結果に影響はなし）

レポート内で使用したプログラムは GitHub にアップロードしています。 `https://github.com/myuuuuun/various/tree/master/ContinuousAlgorithm/HW1/`

## 問題 1

実数 $x$ に対して $y = cos(x)$ を浮動小数点数で計算した時に、$y$ に含まれる誤差を見積もれ。

※ $x$ を浮動小数点で表現する時の誤差と $cos(x)$ を浮動小数点数で表現する時の誤差だけを考える。

$x$ に $\delta$ 程度の誤差が含まれている時、$y = cos(x)$ に伝播する誤差を見積もる。2 乗の項まで 0 周りで Taylor 展開すると、

$$\begin{aligned}|cos(x+\delta) - cos(x)| &= \left|\left\{cos(\delta) - sin(\delta)(x+\delta) - cos(\delta)\frac{(x+\delta)^2}{2} + O(x^3)\right\} - \left\{1 - \frac{x^2}{2} + O(x^3)\right\}\right| \\ &\approx |x\delta| + \frac{\delta^2}{2}\end{aligned}$$

となる。ただし $\delta \simeq 0$ を想定しているので、$sin(\delta) \approx 0,\ cos(\delta) \approx 1$ となることを用いた。

浮動小数点形式の表現誤差（相対誤差）を $\epsilon$ とおくと、ある浮動小数点数 $r$ に含まれる絶対誤差は $\epsilon|r|$ 程度になるので、$y$ に含まれる誤差は結局、

$$\begin{aligned}\Big|cos(x+\epsilon|x|) + \epsilon\Big|cos(x+\epsilon|x|)\Big| - cos(x)\Big| &= \Big|cos(x+\epsilon|x|) - cos(x)\Big| + \epsilon\Big|cos(x+\epsilon|x|)\Big| \\ &\simeq \epsilon x^2 + \frac{\epsilon^2 x^2}{2} + \epsilon\Big|cos(x+\epsilon|x|)\Big| \\ &\simeq \epsilon x^2 + \epsilon\Big|cos(x+\epsilon|x|)\Big| \\ &\leq \epsilon x^2 + \epsilon\end{aligned}$$

と見積もることが出来る。ただし、$\epsilon^2$ が十分に小さいことを用いた。$\epsilon x^2$ が $\epsilon$ に比べて小さければ、誤差は $\epsilon$ 程度であると考えられる。

$|x|$ が大きい時、$cos(x)$ の精度が下がるのはなぜか。

倍精度の場合、$\epsilon = 2^{-52} \simeq 10^{-16}$ 程度であるので、$|x|$ がの整数部が 6 桁程でも計算結果に大きく影響することが予想できる※。この誤差を解消するためには

$$x \quad \leftarrow \quad x \ \ mod\ 2\pi$$

として（$\leftarrow$ は代入）、先に $2\pi$ の定数倍を引いてから計算すれば良い。
※ ただし、大きな $|x|$ に対して $cos(x)$ の Taylor 展開を行うこと自体がそもそも難しい。

$|x|$ が 0 に近い時、$1 - cos(x)$ の相対精度が下がるのはなぜか。

$|x| \simeq 0$ の時、$cos(x) \simeq 1$ となるので、$1 - cos(x)$ を計算する際に桁落ちが生じる。これを防ぐためには、

$$1 - cos(x) = 2sin^2\left(\frac{x}{2}\right)$$

として、右辺を計算すれば良い。

例: $x = 0.02$ として $2sin^2(\frac{x}{2})$ と $1 - cos(x)$ をそれぞれ単精度で計算すると、

- 前者: $1.99993323 \times 10^{-4}$
- 後者: $1.999\underline{73583} \times 10^{-4}$

となり、後者の精度は 4 桁しかない。

ソースコード: hw1-1.cpp

```
#include <iostream>
#include <limits>
#include <cmath>
using namespace std;

int main(void){
  float x = 0.02;
  cout.precision(numeric_limits<float>::max_digits10);
  cout << 1 - cos(x) << endl;
  cout << 2 * pow(sin(x / 2), 2) << endl;
  return 0;
}
```

## 問題 2

半径 $r$ の 2 円が等速直線運動をしている時、衝突時刻を求めるプログラムを書け。

半径 $r$, 2 円の初期位置（x 座標, y 座標）, 2 円の移動速度（x 方向, y 方向）を与え、衝突時間を求めるプログラムが以下。

ソースコード: hw1-2.cpp

```
#include <iostream>
#include <array>
#include <cmath>
#include <limits>
#define PI 3.1415926535
```

```
using namespace std;

array<double, 4> f_to_d(array<float, 4> f_array);
template <class T> array<T, 2> quadratic(T a, T b, T c);
template <class T>
  T compute_clash_time(array<T, 4> circle1, array<T, 4> circle2, T radius);

int main(){
  float radius = 1.0;
  float distance = 10.0;
  float rad, cos_rad, sin_rad, clash_time_f;
  double clash_time_d;
  array<float, 4> circle1, circle2;

  cout.precision(numeric_limits<float>::max_digits10);
  for(int deg=0; deg<=180; deg++){
    rad = deg * PI / 180;
    cos_rad = cos(rad / 2);
    sin_rad = sin(rad / 2);

    circle1[0] = 10 * sin_rad + 1;
    circle1[1] = 10 * cos_rad;
    circle1[2] = -10 * sin_rad;
    circle1[3] = -10 * cos_rad;

    circle2[0] = -1 * circle1[0];
    circle2[1] = -1 * circle1[1];
    circle2[2] = -1 * circle1[2];
    circle2[3] = -1 * circle1[3];

    array<double, 4> circle1_d, circle2_d;
    circle1_d = f_to_d(circle1);
    circle2_d = f_to_d(circle2);

    clash_time_f = compute_clash_time(circle1, circle2, radius);
    clash_time_d = compute_clash_time(circle1_d, circle2_d, (double)radius);
    cout << deg << ", " << abs(clash_time_d - clash_time_f) << endl;
  }
  return 0;
}


// ax^2 + bx + c = 0 の形の二次方程式を解く
template <class T>
array<T, 2> quadratic(T a, T b, T c){
  T x1, x2;
  T d;
  array<T, 2> rst;

  // 判別式D < 0 ならnan を返す
  d = sqrt(pow(b, 2) - 4 * a * c);

  //cout << b * b << ", " << 4 * a * c << endl;

  if(isnan(d)){
    rst[0] = numeric_limits<T>::quiet_NaN();
    rst[1] = numeric_limits<T>::quiet_NaN();
    return rst;
```

```
  }

  // 桁落ち防止
  if(b < 0){
    x1 = (-1 * b + d) / (2 * a);
  }
  else{
    x1 = (-1 * b - d) / (2 * a);
  }

  // 解と係数の関係
  x2 = c / (a * x1);
  rst[0] = x1;
  rst[1] = x2;

  return rst;
}


// 2 円の半径，初期位置，速度（等速）から衝突時刻（最初の1 回）を計算
// circle1, 2: [ 初期位置x 座標，初期位置y 座標，x 方向速度，y 方向速度 ]
template <class T>
T compute_clash_time(array<T, 4> circle1, array<T, 4> circle2, T radius){
  T x, y, vx, vy;
  T clash_time;

  x = circle1[0] - circle2[0];
  y = circle1[1] - circle2[1];
  vx = circle1[2] - circle2[2];
  vy = circle1[3] - circle2[3];

  if(x*x + y*y <= 4*radius*radius){
    clash_time = 0;
  }
  else{
    array<T, 2> time_list =
        quadratic(vx*vx+vy*vy, 2*(x*vx+y*vy), x*x+y*y-4*radius*radius);

    if( isnan(time_list[0]) || (time_list[0] < 0 && time_list[1] < 0) ){
      clash_time = numeric_limits<T>::quiet_NaN();
    }
    else{
      if(time_list[0] < 0 || time_list[1] < time_list[0]){
        clash_time = time_list[1];
      }
      else{
        clash_time = time_list[0];
      }
    }
  }

  return clash_time;
}

// use in main()
array<double, 4> f_to_d(array<float, 4> f_array){
  array<double, 4> d_array;
  for(int i=0; i<4; i++){
```

```
    d_array[i] = f_array[i];
  }
  return d_array;
}
```

円がどのように衝突する時に相対精度が落ちやすいか？

衝突時刻 $t$ に関する 2 次方程式を解く過程で、$\sqrt{b^2 - 4ac}$ が桁落ちを起こし、かつ $-b$ がルートの中身に比べて相対的に小さい場合、ルートの中身の桁落ちが結果にまで影響することが予想される。中身を変形すると、

$$
\begin{aligned}
&b^2 \simeq 4ac \\
&\quad (xv_x + yv_y)^2 \simeq (v_x^2 + v_y^2)(x^2 + y^2 - 4r^2) \\
&\quad (xv_y - yv_x)^2 \simeq 4r^2(v_x^2 + v_y^2)
\end{aligned}
$$

の時、精度が落ちやすい。ただし、これが具体的にどのようなケースに相当するのかは未検討。

例) 上のコードでは、

- 円 1: x 座標 $10sin(1°) + 1$, y 座標 $10cos(1°)$, x 方向速度 $-10sin(1°)$, y 方向速度 $-10cos(1°)$
- 円 2: x 座標 $-10sin(1°) - 1$, y 座標 $-10cos(1°)$, x 方向速度 $10sin(1°)$, y 方向速度 $10cos(1°)$

とすると、相対精度が 2～3 桁程落ちた（衝突時刻 1.0, 衝突地点 0, 0)。