情報数学C
Mathematics for Informatics C

第1回 ガイダンス, 数値計算の基礎
(数の表現, 数値誤差, 桁落ち)

情報メディア創成学類
藤澤誠

講義の概要

- 開講日: 秋AB 水曜 3,4時限
- 教室 : 7A106
- 担当 : 藤澤 誠 (fujis@slis.tsukuba.ac.jp , 春日7D401)
  コンピュータグラフィックス・物理シミュレーション
- 概要 :
  数学のコンピュータサイエンスへの応用として,これまで習得してきた微分積分,線形代数を**離散的に計算する**ための手法を講義する.多くの問題が数学によりモデル化されている中で,**それを如何にしてコンピュータを用いて計算するのか**,という点を中心として応用例や実際のアルゴリズム等も示しながら解説する.

講義の注意点

- 2018年度までに情報メディア創成学類で開設された「情報数学III」(GC21301)の単位を修得した者の履修は認めない.

  講義内容が大きく変わっているので, 単位は取得済みだけど講義は聴きたいという場合は参加してもOK
  (その場合単位を出せないのでレポート提出は自由)

- 出席はmanabaの出席カード(respon)を用いる.
  毎回講義の最初に出席番号を示す.

学習・教育目標

1. **数値計算**のための概念とその基礎を知り, **説明できるようになる.**
2. 数学的にモデル化された問題を**実際にコンピュータで計算できる**ようになる.

講義計画

	日程	講義内容
1	10月2日	ガイダンス&数値計算の基礎 (数の表現,数値誤差,桁落ち)
2	10月9日	線形連立方程式の直接解法 (ガウスの消去法,ピボット選択付きガウス消去法,LU分解,コレスキー分解)
3	10月23日	線形連立方程式の反復解法 (ヤコビ法,ガウス・ザイデル法,SOR法,共役勾配法)
4	10月30日	非線形方程式の求根問題 (2分法,ニュートン法,初期値と収束性,DKA法)
5	11月6日	最適化問題 (黄金分割探索,最急降下法,準ニュートン法)
6	11月13日	補間法と回帰分析 (ラグランジュ補間,スプライン補間,最小2乗法)
7	11月20日	数値積分 (区分求積法,台形公式,シンプソン公式)
8	12月4日	常微分方程式の数値解法 (前進/後退差分法,中心差分法,ルンゲ・クッタ法,完全陰解法)
9	12月11日	偏微分方程式の数値解法 (放物型方程式,楕円型方程式,双曲型方程式)
10	12月18日	行列の固有値計算 (べき乗法,ハウスホルダー変換,QR法)

10月16日(水)は月曜授業日, 11月27日(水)は推薦入試による臨時休講のため授業なし.

参考書

戸川隼人ほか : **よくわかる数値計算 アルゴリズムと誤差解析の実際**, 日刊工業新聞社 (¥2,300+税)
- コードは載っていないけど導出含めてアルゴリズムの詳しい解説が載っている (コードがないので評価が低いけど個人的にはおすすめ)

皆本晃弥 : **C言語による数値計算入門 解法・アルゴリズム・プログラム**, サイエンス社 (¥2,400+税)

W.H.Pressほか : **Numerical Recipes in C 日本語版**, 技術評論社 (¥4,757+税)
- C言語によるコードが載っている本. 後者はこの分野ではとても有名な本だけど分厚いので辞書的に使うことをおすすめ

参考資料は, 必ずしも購入する必要はない
授業で必要な資料等は配布します

成績評価

- 到達目標の確認のための問題を

中間および期末レポートとして課す.
これらを提出して満点の**60%以上**をとること
を単位取得の条件とする.
また,A+～Cの評点はレポートの点数に基
づいて評価する.

responを用いた出席確認

manabaと連携している**respon***で出席(視聴)確認を行う

・https://atmnb.tsukuba.ac.jpにアクセスすると右下の
画面が出るので「**9桁の番号を入力**」のところに
各回の講義で示された番号を入力

・その回の講義で学んだことを2～3文以内にまとめて記述

9桁の番号を入力して「Go」をクリック

「確認」→「提出」のクリックを忘れないように!

*responについてはmanabaの学生用マニュアルp20参照

講義の注意事項

- わからない時はその場で質問するように.
- 専門用語は2回目以降は説明なしに使うことがあるので,復習しておくこと.
- 数式の意味を理解するだけでなく,それを実際にプログラムすることを考えながら講義を受けること.
- 授業用資料はWebページに載せる.
https://fujis.github.io/numerical/
- 授業資料とは別に講義中に説明するプログラムコード(C++)はGitHubに置いてある.
https://github.com/fujis/numerical/
⇒ 理解に必要な最低限のC++の知識は今日の講義の中で教えます.

今回の講義でやること

- 数値計算とは?
- 今後の講義の進め方
(コード例の取得方法)
- コード例を理解するためのC++の基本
- 数の表現,数値誤差,桁落ち

数値計算とは?

数学って何の役に立つんだろう?

答え) 実際に色々な分野で使われている.

例えばCG分野では:
行列による座標変換(線形代数),レベルセット**関数**による形状表現,**媒介変数**表示による形状表現,**多項式**と**微分**を使った曲線,**三角関数**による反射モデル,ナビエ・ストークス**方程式**による水の表現,サンプリング(**確率**)論に基づく描画高速化,**積分**方程式を用いたボリュームレンダリング,などなど

数値計算とは?

中学,高校や大学で習った方程式とかって
実際に使えるの?

例) 2次方程式,連立方程式,微分/積分,数列

答え) 問題をモデル化するのには使えるけど,
実際の問題に当てはめると複雑になり,
人間が解くことは**難しい**

数値計算とは?

人間が解くのが難しい問題をどうやって解くのか?

⇒ **コンピュータ**を使おう!

コンピュータが**できること**とは?

- **四則演算**(+,—,×,÷)
- **反復処理** (これが一番得意!)

⇒ 難しく複雑な問題を**単純な四則演算の繰り返し**で近似計算

これが数値計算!

数学の問題を解いてみよう

方程式の求根問題:$f(x) = 0$ を満たすxの値を求める問題

例) $f(x) = 2x^2 - 9x + 14 - \frac{9}{x} + \frac{2}{x^2} = 0 \quad (x > 0)$

(2017年度筑波大学前期日程 数学問題より)

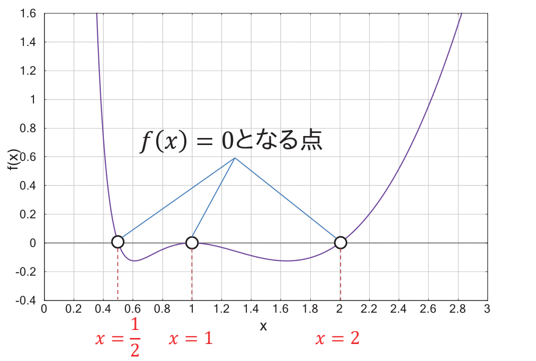
数学で解くと:
$\left(x + \frac{1}{x}\right)^2 = x^2 + \frac{1}{x^2} + 2$ を使って, $\left(x + \frac{1}{x}\right)$について の2次方程式にして解く

解) $x = \frac{1}{2}, 2, 1$

数学の問題を解いてみよう

$f(x) = 2x^2 - 9x + 14 - \frac{9}{x} + \frac{2}{x^2} = 0 \quad (x > 0)$

⇒ グラフにするとどうなるか見てみよう



数学の問題を解いてみよう

グラフにすると**直感的に答えが分かる**

⇒ どんな問題も**グラフ**にしてしまえばいいのでは?

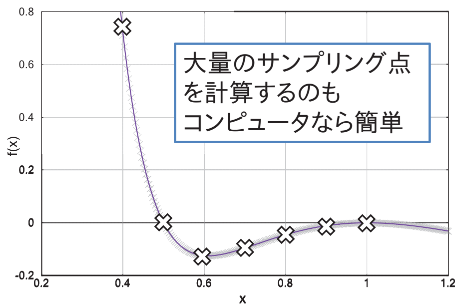
そもそもグラフってどうやって描いていたのか?

⇒ いくつかの点(**サンプリング点**)で実際に$f(x)$を**計算**してみてその点を通るような線を描く

数学の問題を解いてみよう

$$f(x) = 2x^2 - 9x + 14 - \frac{9}{x} + \frac{2}{x^2} = 0$$

$f(0.4) = 0.72$
$f(0.5) = 0.0$
$f(0.6) = -0.12444$
$f(0.7) = -0.0955102$
$f(0.8) = -0.045$
$f(0.9) = -0.0108642$
$f(1.0) = 0.0$
⋮



グラフの$0.2 < x < 1.2$部分だけ拡大した図

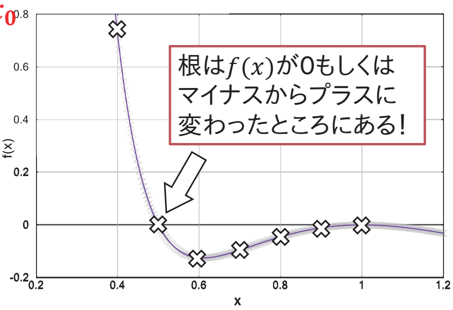
数学の問題を解いてみよう

$$f(x) = 2x^2 - 9x + 14 - \frac{9}{x} + \frac{2}{x^2} = 0$$

初期値: $x(0), x_0$

$f(\boxed{0.4}) = 0.72$
$f(0.5) = 0.0$
$f(0.6) = -0.12444$
$f(0.7) = -0.0955102$
$f(0.8) = -0.045$
$f(0.9) = -0.0108642$
$f(1.0) = 0.0$
⋮

xの増分: Δx



グラフの$0.2 < x < 1.2$部分だけ拡大した図

数学の問題を解いてみよう

方程式の求根問題：$f(x) = 0$ を満たすxの値を求める問題

例) $f(x) = 2x^2 - 9x + 14 - \frac{9}{x} + \frac{2}{x^2} = 0 \quad (x > 0)$

(2017年度筑波大学前期日程 数学問題より)

数値計算で解くと:

1. 初期値 x_0を決めて, $x = x_0$とする
2. $f(x)$を計算 ⇦ **四則演算**
3. もし$f(x) = 0$ もしくは $f(x_0)$と$f(x)$の符号が異なった場合はxが解, 同じならば$x_0 = x$とし, xをΔxだけ増やして2に戻る

繰り返し演算(反復処理)

数学の問題を解いてみよう

数値計算法の特徴･問題

- **アルゴリズム化**さえできてしまえば
  どんな**複雑な式**でも同じように解ける
- **パラメータ設定**によっては解けないことあり
  例) 先ほどの問題で初期値が3とかだったら?
- **計算効率**は?
  ⇒ コンピュータのリソースも無限ではない
  (解の正確さはΔxに依存する)
  ⇒ もっと**効率の良い数値計算法**がある
  (来週以降の授業ではこれの説明を行っていく)

今回の講義でやること

- 数値計算とは?
- **今後の講義の進め方 (コード例の取得方法)**
- コード例を理解するためのC++の基本
- 数の表現,数値誤差,桁落ち

次回以降の授業の進め方

1. その回の講義で対象となる数式(解きたい数式)を提示して説明 講義計画
2. 解くためのアルゴリズムの説明
3. 実際のコード例を使った説明

コード例(&授業資料)はgithubに置いてある

https://github.com/fujis/numerical

一部C++の機能を使っているのでこの後に説明

レポートは実際に問題をプログラムを使って解いてもらうというものになる
例) 円周率πを数値計算法を使って求めよ

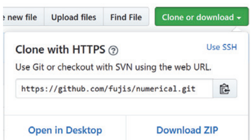
コード例について

Githubからコードを得る方法

https://github.com/fujis/numerical

(publicリポジトリですがプッシュはできません)

方法1) ウェブブラウザで上記URLにアクセスして, Clone or download →Download ZIP



方法2) git環境が手元のPCにあるならば

```
git clone https://github.com/fujis/numerical.git
```


今回の講義でやること

- 数値計算とは?
- 今後の講義の進め方 (コード例の取得方法)
- **コード例を理解するための C++の基本**
- 数の表現,数値誤差,桁落ち

C++の基本

- **C++**
  - C言語を**拡張**した言語
  - C言語の完全な下位互換性あり
    ⇒ **迷ったらC言語で書けば問題ない**
  - オブジェクト指向(class)の追加がメイン

コード例を読み取るための最低限の機能だけを教えます.

⇒ オブジェクト指向まではやりません

(コード例ではclassは使っていないので)

Hello World!(C言語版)

List.1

```c
#include <stdio.h>
int main(void)
{
  /* Hello の画面出力World */
  printf("Hello␣World!\n");
  return 0;
}
```


Hello World!(C++版)

List.2

```cpp
#include <iostream>
//名前空間の設定
using namespace std;
int main(void)
{
  // Hello の画面出力World
  cout << "Hello␣World!" << endl;
  return 0;
}
```


標準入出力

画面表示, 数値入力のC言語のとの対応

標準出力 : cout	⟺	printf
標準入力 : cin	⟺	scanf

C++でprintf,scanfも使えます

使い方

```cpp
float a;
int b;
cin >> a;
cin >> b;
cout << "a␣=␣" << a << endl;
cout << "b␣=␣" << b << "␣=␣0x" << hex << b << endl;
```

cinからデータを変数に入れて, 変数の値をcoutに出力するイメージ
(>>, <<の本来の使い方はシフト演算)

標準入出力

cin,coutの特徴

- stdio.hの代わりにiostreamをインクルード
- 変数の型を気にしなくて良い
- cinの場合はエラー処理も可能(次ページ参照)

前ページの出力結果

```
10
20
a = 10
b = 20 = 0x14
```

hexを通すと16進表示になる

[endlの役割] endlはprintfにおける¥nと同じ改行を示す.
ただし, cout は "¥n"としても改行できる

cinのエラー処理

cinは内部にエラーフラグ変数を持つ

```cpp
int x;
cin >> x;
while(!cin){
  cin.clear();
  cin.ignore(INT_MAX, '\n');
  cout << "再入力␣:␣";
  cin >> x;
}
```

エラーフラグのチェック
↓
エラーフラグのリセット
↓
入力バッファのクリア
↓
再入力

ファイル入出力1

fstream (ifstream, ofstream)

- ファイル出力：ofstream ⟺ fprintf
- ファイル入力：ifstream ⟺ fscanf
- ファイル入出力：fstream ⟺ FILE

ファイル入出力の手順

ファイルを開く
↓ 開けたら
データをファイルに入力 or 出力
↓
ファイルを閉じる

ファイル入出力2

```
ofstream fo;
fo.open("output.txt");
if(fo){
  fo << "データの出力" << endl;
  fo.close();
}
else{
  //エラー処理
}
```


変数の宣言

C++では変数を必ずしも関数の先頭で宣言する必要はない
⇒ 変数のスコープ(有効範囲)に注意！

List.3

```
#include <iostream>
using namespace std;
int main(void)
{
  //変数の宣言
  for(int i = 0; i < 10; ++i){
    double a = (double)i*i;
    cout << a << endl;
  }
  return 0;
}
```

double a の有効範囲

int iの有効範囲

iとaはfor文の後では使えないことに注意

関数のデフォルト引数

関数の引数にデフォルト値を設定できる

List.4

```
#include <iostream>
using namespace std;
//デフォルト引数
void def_test(int d = 0)
{
  cout << "d = " << d << endl;
}
int main(void)
{
  def_test();
  def_test(1);
  return 0;
}
```

引数に何も指定しなかったらデフォルト値d = 0が使われる

関数のオーバロード

引数が異なる同じ名前の関数を定義できる
↓
先ほどのデフォルト引数を設定した場合は注意が必要
(右の例でfunc(int d=0)としていたらコンパイルエラー)

List.5

```
#include <iostream>
using namespace std;
//関数オーバロード
int func(void)
{
  int d;
  cout << "Input Integer Number : ";
  cin >> d;
  return d;
}
void func(int d)
{
  cout << d << endl;
}
int main(void)
{
  func(func());
  return 0;
}
```


引数の参照渡し

C言語では関数の引数は**コピー渡し**
⇒ 渡したデータを関数内でどんなに変更しても呼び出し元は変わらない

変数の値を更新するには引数を**ポインタ**にしなければならない

List.6

```
void swap(int *a, int *b)
{
  int c;
  c = *a;
  *a = *b;
  *b = c;
}
```

引数の参照渡し

C++でも関数の引数は**コピー渡し**

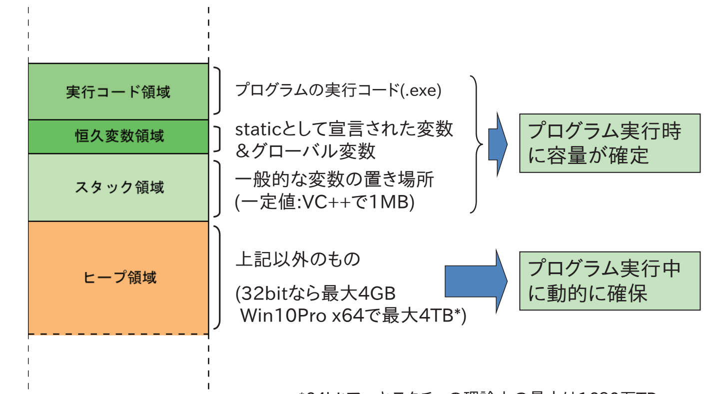
ただし, 引数をポインタにしなくても良い方法が用意されている
⇒ **参照渡し** : 引数に”&”をつけるだけ

List.7

```
void swap(int &a, int &b)
{
  int c;
  c = a;
  a = b;
  b = c;
}
```


メモリについて

メモリマップ



*64bitアーキテクチャの理論上の最大は1680万TB

スタック領域

スタック領域は何のためにあるのか?

レジスタの数を超えた変数が使用されたときに用いられる

スタックの利点 : 割り当てや開放の手間がないので**高速**

スタックの欠点 : スタックのサイズはあまり**大きくできない**

ヒープの利点 : **サイズ制限なし**(ただしメモリ容量による)

ヒープの欠点 : OSによるメモリ管理コードのため**低速**

メモリの動的確保

malloc,free (C言語)

```
void *malloc(size_t size)
void *calloc(size_t n, size_t size)

int *p;
p = (int*)malloc(10*sizeof(int));
--------pを使った処理 ---------
free(p);
```

new,delete (C++)

```
int *p;
p = new int[10];
--------pを使った処理 ---------
delete [] p;
```

確保した領域は
必ず開放する

(STLのvectorは自動開放)

メモリの動的確保

配列のサイズを**実行中に変更**するには?

```
int *p = new int[10];
// ----pを使った処理----
int *p0 = new int[10];// pの値の一時的な退避場所
for(int i = 0; i < 10; ++i) p0[i] = p[i]; // pの値を確保しておく
delete [] p;      // メモリ領域を一時的に削除
p = new int[20]; // 新しくメモリ領域を確保
for(int i = 0; i < 10; ++i) p[i] = p0[i]; // データを元に戻す
```

とても面倒

動的配列を扱えるもっと簡単な方法がないか?

⇒ STL : Standard Template Library

STLによる動的配列

STL(標準テンプレートライブラリ)による動的配列
- 標準とあるようにほとんどの処理系で**デフォルトで使える**
- 任意型の動的配列を作れ, **データを保持したまま自由にサイズを変更可能**
- 解放は自動(**deleteしなくて良い**)

使い方 :

1. vectorをインクルード
```
#include <vector>
```
2. vector<型名> で変数を宣言
```
vector<int> x;
vector<int> y(10, 0); // 配列サイズを10にして, 値0で初期化
```
3. (配列サイズをresize関数で変更)
4. 普通の配列として使用

STLによる動的配列

STL(標準テンプレートライブラリ)による動的配列

```
#include <vector>
using namespace std;
void main(void){
    vector<int> x;

    // 配列サイズを変更しながら1つずつデータを追加(push_back関数)
    for(int i = 0; i != 5; ++i) x.push_back(i);

    // resize関数でサイズ変更
    x.resize(10);

    // 現在のデータ数はsize関数で取得できる
    for(int i = 0; i != x.size(); ++i) cout << x[i] << endl;
 }
```


STLによる動的配列

他にもSTLにはdeque, list, stackなど様々なデータ構造が定義されている.

興味のある人は調べてみよう.

(この講義で使うのはvectorぐらいなのでそれ以外は解説しません)

今回の講義でやること

- 数値計算とは?
- 今後の講義の進め方
(コード例の取得方法)
- コード例を理解するためのC++の基本
- **数の表現,数値誤差,桁落ち**

数値の表し方と精度

数値計算では**数値をコンピュータ内**で表現する

・コンピュータ内は**2進数**

・数学で使うのは**10進数**が一般的

どのような**変換**がされてどのように**格納**されているかを知ることが重要

⇒ 後で説明する誤差の問題と関係する

数値の表し方と精度

整数型データ

- C言語の char, short, int, long など
  (charは文字型だけど内部的には8bit整数)
- 2の補数表現 (補足スライドあり)
- 扱える数値の範囲

  8bit : -128〜127
  16bit : -32,768〜32,767
  32bit : -2,147,483,648〜2,147,483,647

21億以上の数値を扱うことなんてなさそうだけど実際の数値計算ではありうる

数値の表し方と精度

整数型データ

21億を超える例

- フィボナッチ数列

  $F_{n+2} = F_n + F_{n+1}$ ただし,$F_0 = 0, F_1 = 1$
  $F_{47} = 2{,}971{,}215{,}073$でint型限界を超える
  ($n = 89$で64bit整数の限界(9.2×10^{18})を超える)

- ビッグデータ

  例) Instagramの総画像数
  登録者数約10億人 ⇒ 一人10枚ずつ画像をアップロードしただけで32bit整数の範囲を超える

数値の表し方と精度

整数型データの**利点**

整数値での演算誤差/変換**誤差は発生しない**
(範囲を超えない限り ⇒ 桁落ち)

整数型データの**欠点**

整数値の**範囲を超える**と誤差が発生する

実数全体を表せない

数値の表し方と精度

浮動小数点型データ

- C言語のfloatやdoubleなど
- 数値(実数)を$F \times b^E$の形で表す

例) 1.234×10^{56}など

F:**仮数**, E:**指数**, b:**基数**に分けて表現する

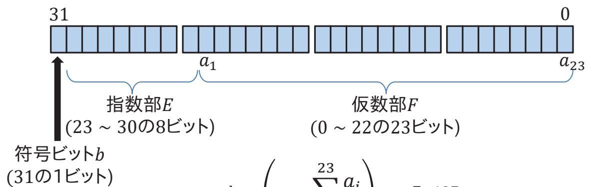
- 仮数, 指数にどれだけビットを割り当てるか?

仮数部を大きくすると小さい数値の**精度が上がる**,
指数部を大きくするとより**大きな数値を表せる**.

数値の表し方と精度

IEEE754規格*

- 単精度浮動小数点数(32bit)



$$(-1)^b \times \left(1 + \sum_{i=1}^{23} \frac{a_i}{2^i}\right) \times 2^{E-127}$$

仮数部を $1.a_1a_2a_3\cdots$ とすることで23bitで24bit分の数値を表現

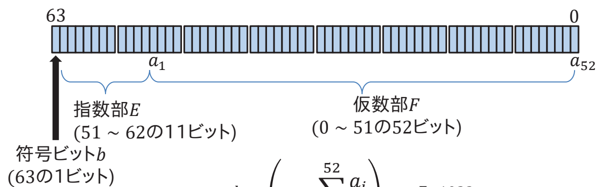
10進数での有効桁数は7~8桁 (24bitでの桁数は$\log_{10} 2^{24} \fallingdotseq 7.22$)
⇒ 例) 1.234567×10^8

* IEEE : 米国電気電子技術者協会

数値の表し方と精度

IEEE754規格

- 倍精度浮動小数点数(64bit)



$$(-1)^b \times \left(1 + \sum_{i=1}^{52} \frac{a_i}{2^i}\right) \times 2^{E-1023}$$

10進数での有効桁数は15~16桁 (53bitでの桁数は$\log_{10} 2^{53} \fallingdotseq 15.95$)

IEEE754規格では4倍精度(128bit)や半精度(16bit)もある

数値の表し方と精度

有効桁を超えるとどうなるのか検証してみよう

```
float x = 123456789;
float y = 123456700;
float z = x-y;
cout.precision(10); // 表示桁数を10桁にする
cout << "x = " << x << endl;
cout << "y = " << y << endl;
cout << "x-y = " << z << endl;
```

```
x = 123456792
y = 123456704
x-y = 88
```

こういうこともあるので整数型は必要

数値の表し方と精度

浮動小数点型データの扱える値の範囲(C/C++の場合)

32bit : 約$\pm 1.17 \times 10^{-38}$〜$\pm 3.4 \times 10^{38}$

64bit : 約$\pm 2.2 \times 10^{-308}$〜$\pm 1.8 \times 10^{308}$

の実数を扱える!

ただし, **丸め誤差/桁落ち誤差**は考慮する必要あり

[C言語で値の範囲を確認するには?]
int型ならINT_MIN,INT_MAX, float型ならFLT_MIN, FLT_MAX
(float.hをインクルードする必要あり) と定義されている
例) cout << FLT_MIN << " ~ " << FLT_MAX << endl;

誤差について

誤差とは？

- 数値計算は基本的に**近似値**を求める
- **真値**との**差** = **誤差**
- 数値計算の評価=誤差の評価

研究ではこの誤差を小さくするために数値計算手法を改良したり，
別の手法を使ったり...

数値計算で発生する誤差

- 丸め誤差，桁落ち誤差
- 打ち切り誤差

誤差について

丸め誤差，桁落ち誤差

2進数⇔10進数の**変換誤差**

なぜ変換で誤差が生じるのか？

小数点以下の数値を2進数に変換すると…

10進数での有限小数は必ずしも(ほとんどの場合)，
2進数で**有限小数にはならない**(無限小数or循環小数になる)

例) $0.1 = 0.000110011001100\ldots_2$

⇒ 2進数に変換したときに無限小数になると

有効桁数以下は<u>切り捨てられる</u> ⇒ **丸め誤差**

(丸められる)

誤差について

丸め誤差を実際に検証してみよう

```
float a = 1.1;
cout.precision(20); // 表示する桁数を20桁に設定
cout << "a = " << a << endl; // 1.1ぴったりになるはずだけど...
```

```
a = 1.1000000238418579102
```

```
float x1 = 1.1;
double x2 = 1.1;
cout << "x1 = " << x1 << "  (single precision)" << endl;
cout << "x2 = " << x2 << "  (double precision)" << endl;
```

```
x1 = 1.1000000238418579102  (single precision)
x2 = 1.1000000000000000888  (double precision)
```


誤差について

丸め誤差，桁落ち誤差

大きい数値と小さい数値の演算時の**桁落ち**

$$a = 1234567$$
$$b = 0.00123$$
$$c = 1234567$$

このとき，$(a+b)-c$ と $(a-c)+b$ の結果は？

誤差について

桁落ちを実際に検証してみよう

```
float a = 1234567;
float b = 0.00123;
float c = 1234567;
cout << "(a+b)-c=" << (a+b)-c << endl;
cout << "(a-c)+b=" << (a-c)+b << endl;
```

```
(a+b)-c=0
(a-c)+b=0.0012300000526010990143
```

$(a+b)$の段階で一旦float型に格納されるので，
$(a+b) = 1234567.00123$ が1234567に**丸められてしまう**

誤差について

打ち切り誤差

無限級数を数学のテクニックを使わずに
反復計算で正確に求める

⇒ **無限**という時点で全てを計算するのは**無理**

⇒ 有限項で計算を<u>打ち切る</u>

打ち切り誤差

例) ライプニッツの公式による円周率πの計算

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

誤差について

円周率πの計算コード例

```
double pi = 0;
int sgn = 1, n = 100;
for(int i = 0; i <= n; ++i){
    pi += sgn/(2.0*i+1.0);    // 数列の計算
    sgn *= -1;                // 符号の反転
}
double pi0 = 3.141592653589793; // 真値
cout << "error = " << 4*pi-pi0 << endl;
```

```
n=100  : 3.151493401070991407
error = 0.009900747481198290955 7
```

```
n=1000  : 3.142591654339544238 2
error = 0.000999000749751122185 67
```

```
n=10000  : 3.141692643590534572 7
error = 0.000099990000074145669 7142
```

反復回数nが大きくなるほど精度は良くなる(ただし，計算時間は長くなる)

誤差について

相対誤差と**絶対**誤差

何回反復すればいいのか？

誤差(真値との差)が**ある一定値**以下になるまで

ある一定値をどう評価するのか？

例) 0〜1で変化する値での誤差0.1 ⇒ 10%

0〜100で変化する値での誤差0.1 ⇒ 0.1%

真値a, 近似値xとしたときの誤差E

$E = |x - a|$ ：**絶対誤差**，　$E = \left|\frac{x-a}{a}\right|$ ：**相対誤差**

誤差について

そもそも**真値が分からない問題**を解こうとしているのでは？

i番目の反復におけるxと$i+1$番目のxを比べる

$E = |x_i - x_{i+1}|$

解に十分近づいたら変化しなくなることが前提

$E < \varepsilon$となったら反復終了

許容誤差**εの決め方** ⇒ 使っている型の有効桁数が参考になる(それ以下にしても意味がないともいえる)

誤差について

コード例(ライプニッツの公式による円周率計算)

```
double eps = 1e-5; // 小数点以下5桁まで求めたい
double pi = 0;
int sgn = 1;
for(int i = 0; i <= 1000000; ++i){
    double pi0 = pi;           // 前の反復の値を確保
    pi += sgn/(2.0*i+1.0)*4;  // 数列の計算
    sgn *= -1;                 // 符号の反転
    if(fabs(pi-pi0) <= eps) break; // 収束判定
}
cout << "pi = " << pi << endl;
```

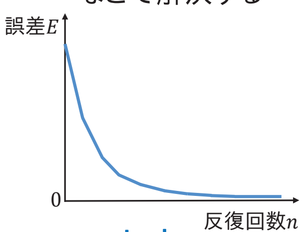
```
pi = 3.1415976535647618384
```

望んだ桁数までは正しい結果が得られている．
ただし，反復回数が十分でないならば得られない可能性も．

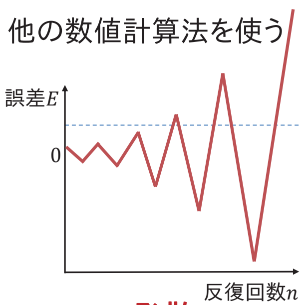
誤差について

パラメータ設定によっては反復しても**解に近づかないこと**もある

⇒ パラメータを変える，他の数値計算法を使うなどで解決する



解が**収束**するケース



解が**発散**するケース

今回の講義のまとめ

- 数値計算とは？
  ⇒複雑な問題を単純な計算の反復で解く
- 今後の講義の進め方
- コード例を理解するためのC++の基本
  ⇒github,C++の基本
- 数の表現,数値誤差,桁落ち
  ⇒誤差の種類，数値計算との関係

次週からは本格的に数値計算法について学んでいきます．

Appendix

(以降のページは補足資料です)

補数表現

正負 を反転させた数(n⇒-n)で表す.

- 「1の補数」表現
  0と1を反転させた値を負の数とする
- 「2の補数」表現
  000…0を境界として正負を分ける

1の補数表現 (3ビットの場合)

2進数	10進数	2進数	10進数
000	+0	111	−0
001	+1	110	−1
010	+2	101	−2
011	+3	100	−3

0と1を反転

0が2種類(+0と-0)あることに注意

1の補数表現(3ビット)での計算(桁あふれなし)

```
      100    (= −3)
+ )   010    (= +2)
-------------
      110    (= −1)
```

桁があふれないときは +0 と -0 の境界をまたがない (-3 → -2→-1) ので, 通常通りに計算できる

1の補数表現(3ビット)での計算(桁あふれあり)

```
         110    (= −1)
+ )      010    (=+2)
----------------
    (1) 000    (=+0)  桁あふれ
         ↓     補正(1 を足す)
        001    (=+1)
```

桁があふれるときは +0 と -0 の境界をまたぐ (-1 →-0→+0) ので, 解を1ずらす必要がある

2の補数表現 (3ビットの場合)

2進数	10進数	2進数	10進数
000	+0	000	−0
001	+1	111	−1
010	+2	110	−2
011	+3	101	−3

0と1を反転後, 1を加える
(or 1を引いた後反転)

0は1種類のみ

2の補数表現(3ビット)での計算

	010	(= +2)
+)	111	(= −1)
	(1) 001	(= +1)

桁あふれしても補正の必要がない.