計算物理学II

第6回
数値計算プログラミング入門（2）：
条件分岐, ループ, 関数

秋山 進一郎

2025年11月14日

授業日の確認

・全10回

- 第1回：10月3日（金）
- 第2回：10月10日（金）
- 第3回：10月17日（金）
- 第4回：10月24日（金）★
- 第5回：10月31日（金）
- **第6回：11月14日（金）★**
- 第7回：11月21日（金）
- 第8回：12月5日（金） ★
- 第9回：12月12日（金）
- 第10回：12月19日（金）★

・★の付いた授業にてレポート課題を配布予定

（前回と）今日の授業の目標

- Pythonを使ってみる
  - 変数と型，四則演算
  - 条件分岐，ループ
  - 関数，スコープ
- 数学と数値計算の違いについて理解を深める

毎回の授業の流れ

“compphy2”に移動する

↓

“git fetch”と“git merge”で演習資料を入手

↓

“lecture*”に移動し,
“lecture_material_*.pdf”を開いて演習開始

本日の演習内容

- 第5回の演習内容の続きに取り組む
  - lecture_material_5.pdfを最後まで終える
- 数値計算で現れる誤差について(本演習資料)
- 上の演習が終わった人は, 第２回レポート課題に取り組む
  - レポート問題はlecture6の中に格納されている
  - レポートの提出〆切は11/28

- 第1回レポート課題時のアンケート結果はこちら

数値計算する時に頭に入れておいて欲しいこと

- 皆さんが普段学んでいる「数学」と「数値計算」には大きな違いがあります
  - 以降の演習は, 数学における「実数」と数値計算における「浮動小数点数」に注目し, この違いを理解してもらうことを目的としています
  - 数値計算の厄介な（煩わしい）側面とも言えます
- 数値計算をする上での“常識”として理解しておいてください

数値計算における「誤差」

- 大きく分けて次の二種類の「誤差」がある
  - **打ち切り誤差 (truncation errors)**
    - 計算機が有限回の計算しかできないことに起因する誤差

    (例) $y = \sum_{n=0}^{n_{\max}} x^n/n! \Longrightarrow n_{\max}$に依存した$\mathrm{e}^x$の近似値
  - **丸め誤差 (rounding errors)**
    - 計算機が有限桁の数値しか扱えないことに起因する誤差

    (例) 数学的には, $\left(\sqrt{2}\right)^2 - 2 = 0$. 数値計算だと $\left(\sqrt{2}\right)^2 - 2 \neq 0$

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> (math.sqrt(2))**2 - 2
4.440892098500626e-16
```

「4.44…e-16」は
「$4.44\cdots\times10^{-16}$」の意味

計算機上では「実数」を完全に再現することはできない

- 第5回の演習で出てきた「浮動小数点数」とは, 以下のような形で表される数のこと
  - $\pm$仮数$\times 10^{\text{指数}}$
    - ここで, 計算機が扱えるのは, 有限桁の数に限られる
    - そのため, 任意の実数を計算機上で完全に再現することはできない
- ちなみに, Pythonの「整数」は任意精度のため, 「整数」は完全に再現できる

計算機上では「実数」を完全に再現することはできない

- Pythonでよく用いられる「倍精度浮動小数点数」の表現範囲はおおよそ以下の通り
  - $\pm 4.9 \times 10^{-324} \leftrightarrow \pm 1.8 \times 10^{308}$
- 1.8×10^{308}より大きな数を扱おうとすると「オーバーフロー」する
- 絶対値が4.9×10^{-324}より小さい数を扱おうとすると「アンダーフロー」する
- 倍精度浮動小数点数の場合, その精度は$1/2^{52}$である
  - $1/2^{52} \approx 2.2 \times 10^{-16}$のため, 15~16桁の精度になる
    - ゆえに, 例えば, 計算機上で4.9×10^{-324}という数を表現することは可能だが, 「324桁の数字として表現可能」という意味ではないことに注意

オーバーフローさせてみよう

- lecture6の中に“ex_errors”というディレクトリを作り, “demo_overflow.py”という名前で以下のようなスクリプトを書いてみよう
  - 渡されたlarge_numberをmaxiter回だけ2倍する関数
  - 4行目は, 「resulting_number = resulting_number * 2」の短縮

```
1   def demo_overflow(large_number, maxiter):
2       resulting_number = large_number
3       for i in range(maxiter):
4           resulting_number *= 2
5           print(i, resulting_number)
6
7   demo_overflow(2.0**1021,3)
```

- スクリプトが書けたら, ターミナルから実行してみよう

オーバーフローさせてみよう

- 以下のような結果が得られれば成功

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3 demo_overflow.py
0 4.49423283715579e+307
1 8.98846567431158e+307
2 inf
```

- $2^{1024} \approx 1.7976 \times 10^{308}$で, 倍精度浮動小数点数の表現可能範囲を超え, 「inf」となる
- i=1の結果をコピーし, python3を対話モードで立ち上げ, 以下の計算を実行してみよう
  - 1.99…(9を15個)倍してみると…

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 8.98846567431158e+307 * 1.999999999999999
```

オーバーフローさせてみよう

- 以下のような結果が得られる
  - ぎりぎりオーバフローしない

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 8.98846567431158e+307 * 1.999999999999999
1.7976931348623l5e+308
```

- ちなみに, 1.99…(9を16個)倍するとオーバーフローする

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 8.98846567431158e+307 * 1.999999999999999
1.7976931348623l5e+308
>>>
>>> 8.98846567431158e+307 * 1.9999999999999999
inf
```

マシンイプシロン

- 浮動小数点数において，「1」と「1より大きな最小の数」の差のことをマシンイプシロン（ε_m）と呼ぶ
  - $\varepsilon_m = 1/2^{52} \approx 2.2 \times 10^{-16}$

マシンイプシロンを確認してみよう

・ex_errorsの中に“demo_machine_eps.py”という名前で以下のスクリプトを書こう
  ・渡されたsmall_numberをmaxiter回だけ1/2倍する関数
  ・4行目は，「resulting_number = resulting_number / 2」の短縮
  ・5行目で，1.0+resulting_numberとresulting_numberを出力させる

```
1  def demo_eps_machine(small_number, maxiter):
2      resulting_number = small_number
3      for i in range(maxiter):
4          resulting_number /= 2
5          print(i, 1.0+resulting_number, resulting_number)
6
7  demo_eps_machine(1/2**50, 3)
```

・スクリプトが書けたら，ターミナルから実行してみよう

マシンイプシロンを確認してみよう

・以下のような結果が得られれば成功

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3 demo_machine_eps.py
0 1.0000000000000004 4.440892098500626e-16
1 1.0000000000000002 2.220446049250313e-16
2 1.0 1.1102230246251565e-16
```

・i=2の「1.0+resulting_number」の結果は1.0になる

・i=2において, resulting_numberの値が $\varepsilon_m \approx 2.2 \times 10^{-16}$ より小さくなったから

マシンイプシロンについてもう少し

- Python3を対話モードで立ち上げ, 以下の4種類の足し算を実行しよう
  - この結果に不可解な点はないだろうか？

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1.0 + 2.3e-16
1.0000000000000002
>>> 1.0 + 1.6e-16
1.0000000000000002
>>> 1.0 + 1.12e-16
1.0000000000000002
>>> 1.0 + 1.1e-16
1.0
```

マシンイプシロンについてもう少し

・ε_mよりも小さい数xを足した場合，1+xの結果は1ではなく，1+ε_mとなることがある

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1.0 + 2.3e-16
1.0000000000000002
>>> 1.0 + 1.6e-16
1.0000000000000002
>>> 1.0 + 1.12e-16
1.0000000000000002
>>> 1.0 + 1.1e-16
1.0
```

丸め誤差が顕著に現れる例1

・ほとんど等しい数値同士の引き算(要注意)

(例) 数学的には,

$$\mathbf{1.2345678}12345678912 - \mathbf{1.2345678}00000000000 = 0.00000000012345678912$$

数値計算だと,

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1.23456789012345678912 - 1.23456789000000000000
1.234568003383174e-09
```

数値計算の結果は, 数学的な結果の6桁目までしか再現していない

・なぜ？？

丸め誤差が顕著に現れる例1

- 答えは「丸め誤差」
  - それぞれの数値が計算機上でどのように保持されているかを確認してみよう
  - 下の計算結果から, 最初の数値が有限桁で近似されたことに起因したズレだったことが分かるだろう

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1.2345678912345678912 - 1.2345678900000000000
1.234568003383174e-09
>>>
>>> 1.2345678912345678912
1.234567891234568
>>> 1.2345678900000000000
1.23456789
>>>
>>> 1.234567891234568 - 1.23456789
1.234568003383174e-09
```

丸め誤差が顕著に現れる例2

- 浮動小数点数同士の計算では, 演算順序が結果に影響することがある
  - つまり, 結合律が成り立たないことがある
  - 以下の計算をやってみよう

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> (0.7 + 0.1) + 0.3
1.0999999999999999
>>> 0.7 + (0.1 + 0.3)
1.1
```

丸め誤差が顕著に現れる例2

- 浮動小数点数同士の計算では, 演算順序が結果に影響することがある
  - 次の計算もやってみよう

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 1.0e20
>>> b = -1.0e20
>>> c = 1.0
>>> (a + b) + c
1.0
>>> a + (b + c)
0.0
```

  - a + (b + c)では「ほとんど等しい数値同士の引き算」が発生している
- 丸め誤差の影響をなるべく排除するような数値計算を設計するように意識しましょう
- 数値計算には丸め誤差が伴うことを必ず頭に入れておきましょう

Kahan summation algorithm

- 浮動小数点数同士の足し算で, 丸め誤差を補正していくアルゴリズム
  - ex_errors中に, “kahansum.py”を作り, 以下をコーディングしましょう

home > akiyama > compphys2 > lecture6 > ex_errors > kahansum.py

```
1   def naivesum(xs):
2       total = 0.0
3       for x in xs:
4           total += x
5       return total
6
7   def kahansum(xs):
8       total = 0.0
9       error = 0.0
10      for x in xs:
11          temp = total
12          y = x + error
13          total = temp + y
14          error = (temp - total) + y
15      return total
16
17  xs = [0.7, 0.1, 0.3]
18  print('Naive summation =', naivesum(xs))
19  print('Kahan summation =', kahansum(xs))
20
```

total = total + xの短縮

「xs=[0.7, 0.1, 0.3]」は
「リスト」と呼ばれる書き方
（次回以降の演習で詳しく学びます）

Kahan summation algorithm

- 実行してみましょう

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3 kahansum.py
Naive summation = 1.0999999999999999
Kahan summation = 1.1
```

- 二つの数の足し算ではKahan sumとNaïve sumに違いは出ないが, 三つ以上の数の足し算ではKahan sumによる精度改善が見られる場合がある
- kahansum.pyの17行目を編集し, 以下の二つの場合の足し算もやってみましょう

```
17    xs = [123456789 + 0.01*i for i in range(10)]
```

```
17    xs = [1.0e20, 1.0, -1.0e20]
```

Kahan summation algorithm

・それぞれの足し算の結果は以下のようになります

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3 kahansum.py
Naive summation = 1234567890.4499996
Kahan summation = 1234567890.45
```

```
akiyama@:~/compphys2/lecture6/ex_errors$ python3 kahansum.py
Naive summation = 0.0
Kahan summation = 0.0
```

・二番目の結果では精度改善が見られない

・$\sum_i |x_i| \gg |\sum_i x_i|$の場合はKahan sumによる精度改善は保証されない

計算精度 vs 計算時間

- なお, naïve sumの方がKahan sumよりも演算回数が少なくて済むため, より大規模な足し算の場合には, Kahan sumによる恩恵が受けられないこともある
  - 数値計算では, 「計算精度 vs 計算時間」の間にトレードオフの関係がある
  - Kahan sumに限らない, 数値計算全般に言えること
- 達成したい計算精度と現実的な計算時間(人生は有限時間, 電気代も有料)の間でバランスを取りながら, 使用するアルゴリズムを選択していくことが重要

練習問題：関数の数値を計算する1

- 次の関数の値をx=1000, 10000, 100000, 1000000で計算しましょう
  - $f(x) = \frac{1}{\sqrt{x^2+1}-x}$
  - まずは, 素朴にf(x)を計算するスクリプトを書いて結果を見てみましょう
  - 次に, なるべく丸め誤差を回避できないか考えてみましょう
    - ヒント:「ほとんど等しい数値同士の引き算」を避けるよう事前に式変形できないか?
  - 素朴な計算結果と見比べ, 計算精度が改善されたか調べましょう

練習問題：関数の数値を計算する2

- 次の関数の値をx=9×10^{-16}で計算しましょう
  - $f(x) = \frac{e^x - 1}{x}$
  - 数学的な答えは, $f(x = 9\times10^{-16}) = 1.0000000000000000450$ ...
  - ヒント: 下のようなスクリプトを書き, 結果を見比べてみよう

home > akiyama > compphys2 > lecture6 > ex_errors > practice2.py

```python
import math

def f(x):
    return (math.exp(x)-1)/x

def g(x):
    return (math.exp(x)-1)/math.log(math.exp(x))

x = 9e-16
print(f(x), g(x))

```