Sources of error

誤差

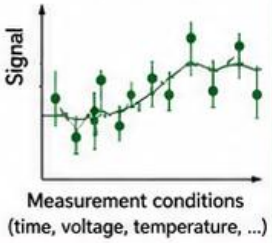
ERRORS IN EXPERIMENTAL SCIENCE: CLASSIFICATION AND ANALYSIS FLOW

Errors enter at each stage and affect the final results

ANALYSIS FLOW

1 MEASUREMENT

Obtain data by experiment



2 MODELING

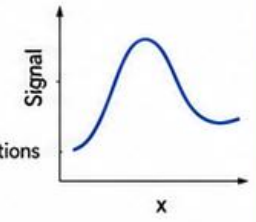
Choose theoretical / analytical model

Physical model & assumptions

$$y = f(x;\theta)$$

- Theoretical approximation
- Assumptions & boundary conditions
- Input parameters

Model prediction



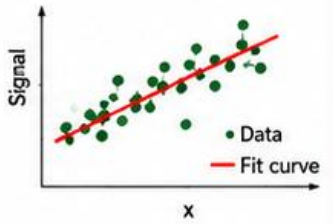
3 COMPUTATION / REGRESSION

Analyze by numerical methods or regression

Numerical computation

- Discretization
- Iterative solver
- Convergence check
- Data handling

Fit result



4 INTERPRETATION

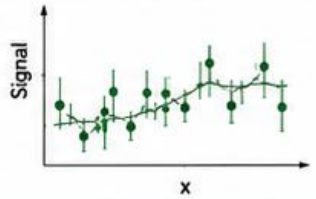
Interpret results physically

- Physical quantities
- Estimated parameters
- Confidence intervals
- Physical insight

MAIN ERROR CATEGORIES

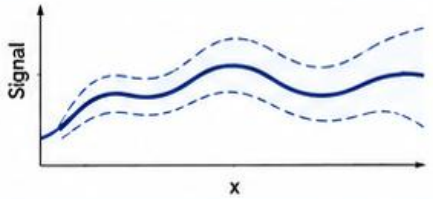
1. MEASUREMENT ERRORS

Random errors (scatter)	Systematic errors (bias)
• Noise • Fluctuations • Resolution limit	• Calibration error • Offset • Drift / shift
Averaging reduces them	Do not vanish even after averaging



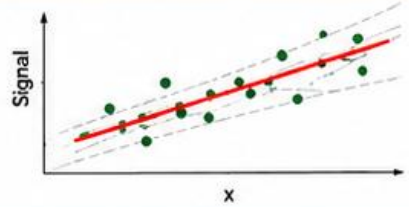
2. MODELING EERRORS

Physical model errors (model idealization)	Model structure errors (model form)	Input parameter uncertainty (parameters)
• Approximate theory • Neglected effects (e.g., correlation, finite temperature)	• Wrong functional form • Oversimplified model (e.g., linear instead of nonlinear)	• Uncertain material constants • Thickness • Temperature
Remain even with exact calculation	Model form causes systematic deviation	Uncertainty propagates to outputs



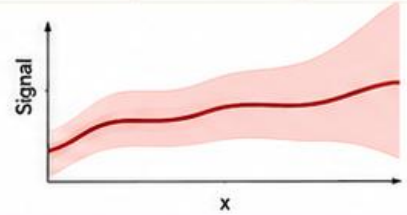
3. NUMERICAL / COMPUTATIONAL ERRORS

Truncation errors (discretization)	Round-off errors (finite precision)	Convergence errors (incomplete)	Other (computational limitations)
• Grid / mesh size • Time step • k-points • Basis set size	• Finite precision • Cancellation • Loss of significance	• Unconverged SCF • Insufficient iterations • Inadequate sampling	• Ill-conditioned problems • Algorithmic instability
Decrease with finer discretization	Limited by machine precision	Arise when convergence is insufficient	Depend on algorithms and problem setup



4. STATISTICAL ESTIMATION ERRORS

Parameter uncertainty (estimation)	Prediction uncertainty (extrapolation)	Identifiability issues (correlation)
• Standard error • Confidence interval	• Prediction interval • Extrapolation outside data range	• Parameter correlation • Multiple solutions
Depends on data scatter and model	Uncertainty grows outside measured range	Different parameters give similar results



5. INTERPRETATION ERRORS

Misinterpretation / Overinterpretation (applicability)

- Confusing correlation with causation
- Ignoring model limitations
- Extrapolating beyond valid range

May lead to wrong conclusions even if analysis is correct

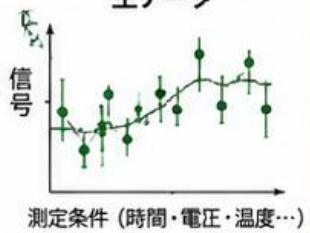
KEY MESSAGE: High-precision computation or good fitting does NOT guarantee a correct result. → Identify the dominant error sources to obtain reliable scientific conclusions.

実験科学における誤差の分類と解析フロー ― 誤差は各段階で入り、最終結果に影響する ―

解析フロー

❶ 測定 (Measurement)
実験・観測でデータを得る

生データ



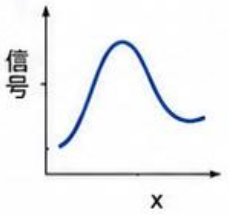
❷ モデル化 (Modeling)
理論・解析モデルを決める

物理モデル・仮定

$$y = f(x;\theta)$$

- 理論近似
- 仮定・境界条件
- 入力パラメータ

モデルの予測

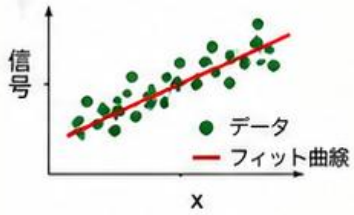


❸ 数値計算・回帰 (Computation / Regression)
数値計算や最小二乗法などで分析

数値計算

- 離散化
- 反復計算
- 収束判定

フィット結果



❹ 解釈 (Interpretation)
結果を物理的に解釈する

物理量・結論

推定パラメータ
信頼区間
物理的洞察

主な誤差種類

1. 測定誤差

ランダム誤差（ばらつき）	系統誤差（偏り）

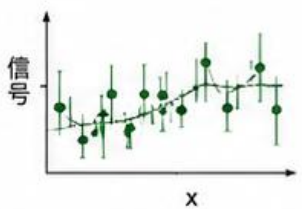
誤差の原因（代表例）

- ノイズ
- 温度ゆらぎ
- 分解能限界
- 校正ずれ
- オフセット
- 位置ずれ

誤差の特徴

平均化で小さくなる	平均化しても消えない

誤差の伝播イメージ



2. モデル化誤差

物理モデル誤差（近似の限界）	モデル構造誤差（モデル形の不適切）	入力パラメータ誤差

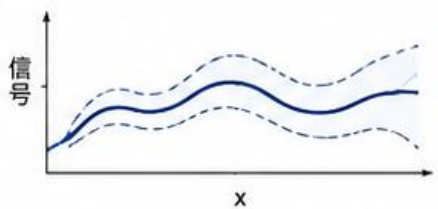
誤差の原因（代表例）

- 近似理論
- 汎関数
- 有限温度効果の無視
- 線形近似
- 単一指数近似
- 単一キャリア近似
- 誘電率
- 厚さ
- 温度

誤差の特徴

高精度計算でも残る	モデルの形が原因	入力不確かさが伝播

誤差の伝播イメージ



3. 数値計算誤差

打切り誤差（離散化など）	丸め誤差（有限精度）	収束誤差（未収束）

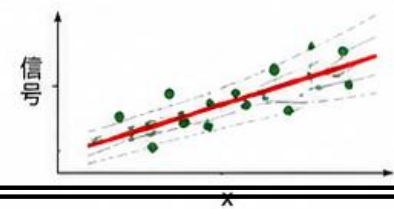
誤差の原因（代表例）

- 格子間隔
- 積分刻み
- k点不足
- 丸め誤差
- 桁落ち
- SCF未収束
- 最適化不足
- サンプリング不足

誤差の特徴

条件改善で小さくできる	桁数に依存して発生	十分な収束で低減可能

誤差の伝播イメージ



4. 統計推定誤差

パラメータ誤差（推定の不確かさ）	予測誤差（外挿など）	識別不能性（パラメータ相関）

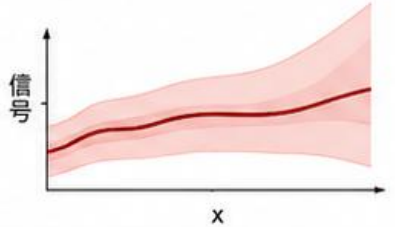
誤差の原因（代表例）

- 標準誤差
- 信頼区間
- 信頼帯
- 外挿誤差
- パラメータ相関
- 多重解

誤差の特徴

データ量に依存	外挿で急増することがある	一意に決められない

誤差の伝播イメージ



5. 解釈誤差

解釈・適用範囲の誤り（過剰解釈・外挿）

誤差の原因（代表例）

- 相関を因果と誤認
- 局所解を正解と誤認
- 適用範囲外の外挿

誤差の特徴

正しくても誤る可能性がある

誤差の伝播イメージ

⚠ **重要なポイント：** 高精度な計算やフィット ＝ 正しい結果 ではない → どの段階の誤差が支配的かを見極めることが、信頼できる結論につながる

Checking Floating-Point Issues in Python

(Pythonで浮動小数点の問題を確認する)

Let’s run Python in **interactive mode** and see what happens.

(Pythonを実行 (インタラクティブモード)して確認してみましょう)

terminal

```
% python
>>> 0.1
0.1                    python displays a shortened representation by default

>>> f"{0.1:.17f}"    Show 0.1 with 17 digits after the decimal point
'0.10000000000000001’

>>> 0.1 + 0.1 + 0.1 == 0.3    How does Python judge this comparison?
False
```

Floating-point representation error (浮動小数点型の表現による誤差)

Representation of floating point in computer:

$-1.0111101_2 \times 2^{-015_{10}}$ **(in binary)**

Errors arise from converting Base 10 to Base 2.

- **Some values do not have errors between Base 10 and Base 2 if fraction equals to 2^n**

  1.0 $= (1.0)_2 \times 2^0$

  0.5 $= (1.0)_2 \times 2^{-1}$

  0.125 $= (1.0)_2 \times 2^{-3}$

  0.0390625 $= 1.25 \times 2^{-5} = (1.01)_2 \times 2^{-5}$

  1.75 $= 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = (1.11)_2$

  0.65625 $= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} = (0.10101)_2$

  100.0 $= 1.5625 \times 64 = (1+2^{-1}+2^{-4}) \times 2^6 = (1.1001)_2 \times 2^4$

- **Other values have errors**

  even if it is represented by a simple figure in Base 10:

  $0.1 = (1.1001100110011001\cdots)_2 \times 2^{-4}$

Python program: sum.py

Program: sum.py
Usage: python sum.py h N
    Summing small value h for many times N

Example command: python sum.py 0.1 10

OUTPUT:

```
 0:                                0.0 + 0.1 => 0.10000000000000001
 1:                                0.1 + 0.1 => 0.20000000000000001
 2:                                0.2 + 0.1 => 0.30000000000000004
 3:  0.30000000000000004 + 0.1 => 0.40000000000000002
 4:                                0.4 + 0.1 => 0.50000000000000000
 5:                                0.5 + 0.1 => 0.59999999999999998
 6:                                0.6 + 0.1 => 0.69999999999999996
 7:                                0.7 + 0.1 => 0.79999999999999993
 8:    0.7999999999999999 + 0.1 => 0.89999999999999991
 9:    0.8999999999999999 + 0.1 => 0.99999999999999989
```

Round-off error (桁落ち誤差): effect of size of FP type

Summing small value *h* for many times *N*

Calc by summation

```
x = 0.0;
for i in range(N)
    x = x + h
```

Error is accumulated by each summation

Calc by multiplication

```
x0 = 0.0;
for i in range (N)
    x = x0 + i * h
```

Typically multiplication is slower than summation, but **the total error originates from only one multiplication operation**

Result of sum_error.py (compare different precision FP types): *h* = 0.01, *N* = 101

Program: sum_error.py

float16: half precision, 16 bit　　float32: single precision, 32 bit　　float64: half precision, 64 bit

Exact:	float16 (error)	float32 (error)	float64 (error)
0.0100:	0.01000213623046875 (−2.14e−06)	0.009999999776482582 (+2.24e−10)	0.010000000000000000 (+0.00e+00)
0.1100:	0.11004638671875 (−4.64e−05)	0.10999998450279236 (+1.55e−08)	0.10999999999999987 (+1.39e−17)
0.2100:	0.21008300781250 (−8.30e−05)	0.21000002324581146 (−2.32e−08)	0.21000000000000048 (−5.55e−17)
0.3100:	0.31005859375000 (−5.86e−05)	0.30999997258186340 (+2.74e−08)	0.31000000000000109 (−1.11e−16)
0.4100:	0.41015625000000 (−1.56e−04)	0.40999987721443176 (+1.23e−07)	0.41000000000000198 (−1.67e−16)
0.5100:	0.50976562500000 (+2.34e−04)	0.50999981164932251 (+1.88e−07)	0.51000000000000231 (−2.22e−16)
…			
0.8100:	0.80273437500000 (+7.27e−03)	0.80999952554702759 (+4.74e−07)	0.81000000000000497 (−4.44e−16)
0.9100:	0.90039062500000 (+9.61e−03)	0.90999943017959595 (+5.70e−07)	0.91000000000000586 (−5.55e−16)
1.0100:	0.99804687500000 (+1.20e−02)	1.00999939441680908 (+6.06e−07)	1.01000000000000675 (−6.66e−16)

Program (roundoff error): sum_error.py

Usage: python sum_error.py *h n iPrintStep*

Summing up *h* for n times with different precision interger types. Output every *iPrintStep* steps.

python sum_error.py 0.1 100 20

exact:	sum16 (error)	sum32 (error)	sum64 (error)
0.1000:	0.099975585937500000 (+2.44e-05)	0.100000001490116119 (-1.49e-09)	0.100000000000000006 (+0.00e+00)
2.1000:	2.095703125000000000 (+4.30e-03)	2.100000143051147461 (-1.43e-07)	2.100000000000000533 (-4.44e-16)
4.1000:	4.089843750000000000 (+1.02e-02)	4.099998474121093750 (+1.53e-06)	4.100000000000001421 (-8.88e-16)
6.1000:	6.121093750000000000 (-2.11e-02)	6.099996566772460938 (+3.43e-06)	6.099999999999994316 (+6.22e-15)
8.1000:	8.148437500000000000 (-4.84e-02)	8.099994659423828125 (+5.34e-06)	8.099999999999987210 (+1.24e-14)

python sum_error.py 0.125 100 20

exact:	sum16 (error)	sum32 (error)	sum64 (error)
0.1250:	0.125000000000000000 (+0.00e+00)	0.125000000000000000 (+0.00e+00)	0.125000000000000000 (+0.00e+00)
2.6250:	2.625000000000000000 (+0.00e+00)	2.625000000000000000 (+0.00e+00)	2.625000000000000000 (+0.00e+00)
5.1250:	5.125000000000000000 (+0.00e+00)	5.125000000000000000 (+0.00e+00)	5.125000000000000000 (+0.00e+00)
7.6250:	7.625000000000000000 (+0.00e+00)	7.625000000000000000 (+0.00e+00)	7.625000000000000000 (+0.00e+00)
10.125:	10.125000000000000000 (+0.00e+00)	10.125000000000000000 (+0.00e+00)	10.125000000000000000 (+0.00e+00)

python sum_error.py 0.0390625 100 20

exact:	sum16 (error)	sum32 (error)	sum64 (error)
0.0391:	0.039062500000000000 (+0.00e+00)	0.039062500000000000 (+0.00e+00)	0.039062500000000000 (+0.00e+00)
0.8203:	0.820312500000000000 (+0.00e+00)	0.820312500000000000 (+0.00e+00)	0.820312500000000000 (+0.00e+00)
1.6016:	1.601562500000000000 (+0.00e+00)	1.601562500000000000 (+0.00e+00)	1.601562500000000000 (+0.00e+00)
2.3828:	2.382812500000000000 (+0.00e+00)	2.382812500000000000 (+0.00e+00)	2.382812500000000000 (+0.00e+00)
3.1641:	3.164062500000000000 (+0.00e+00)	3.164062500000000000 (+0.00e+00)	3.164062500000000000 (+0.00e+00)

Caution for conditional branch (条件分岐における注意)

Calculation of integers does not produce errors.

=> Conditional judgement only using integers works properly

整数変数のみでの条件判断は誤差が出ないので問題ない

```
if i * 10 == 30:
    print(“i == 30”)    # executed if i == 30
```

Calculation of floating points can produce roundoff error.

DO NOT USE: Strict conditional judgement often does not work properly

実数計算では丸め誤差が発生するので、厳格な条件判断は使ってはいけない

```
if x * 10.0 == 30.0:
    print(“x == 3.0”)   # expected to execute if x == 3.0, but may be not
```

Program: bad_if.py

Usage: python bad_if.py h n answer
  Check the condition h * n == answer (eps = 1.0e-10)

python bad_if.py 0.1 1 0.1 **Confirm $\sum_{i=1}^{1} 0.1 == 0.1$**
Summing up 0.1 for 1 times: v = 0.1
v == 0.1?: True
|v – 0.1| < 1e-10?: True

python bad_if.py 0.1 2 0.2 **Confirm $\sum_{i=1}^{2} 0.1 == 0.2$**
Summing up 0.1 for 2 times: v = 0.2
v == 0.2?: True
|v – 0.2| < 1e-10?: True

python bad_if.py 0.1 3 0.3 **Confirm $\sum_{i=1}^{3} 0.1 == 0.3$: Failed**
Summing up 0.1 for 3 times: v = 0.30000000000000004
v == 0.3?: **False** **$\sum_{i=1}^{3} 0.1 == 0.3$ では判断を間違える**
|v – 0.3| < 1e-10?: **True** **$\left|\sum_{i=1}^{3} 0.1 - 0.3\right| <$ eps (eps = 10^{-10}) では正しく判断できる**

How to use conditional branch, if
(条件分岐の判断)

Bad (悪い例):

```
if x * 10.0 == 30.0:
```

DO NOT use the strict comparison '==' for floating values
(浮動小数点の比較には、厳密な比較 == は使わない)

Good (良い例):

```
if abs(x * 10.0 - 30.0) < eps
    eps = 1.0e-6    # epsilon:
```

A value satisfactory smaller than minimum expected value
(想定される誤差よりも十分大きいが、なるべく小さい値)

FP-to-int conversions are system-dependent: The Case in Python

Do not rely only on intuition: results may differ

- **int(x) performs truncation toward zero**

  -1.5 -1.0 -0.5 0 0.5 1.0 1.5

  | truncate toward zero=>| | <= truncate toward zero|

  - Matches intuition for positive values
  - Deviates from intuition for negative values
  - Small FP errors can cause values to be “pulled toward zero”
- **round(x) performs mathematical rounding**
  - Rounds to the nearest integer for both positive and negative values **(round(0.5) depends on system)**
  - **int(round(x)) corresponds to “ordinary rounding to an integer”**
- **Floating-point errors affect rounding behavior**
  - For example, -1.0 may become -0.9999999999999993
  - As a result, int() may return 0 even when the value is close to -1
- **The modulo operator % of PYTHON always returns a non-negative result**
  - -0.2 % 1.0 = 0.8
  - This differs from the mathematical definition of remainder
  - It is important to remember this as Python’s specification
- **Rounding of negative values often behaves counter-intuitively**
  - Especially when FP errors are involved, the behavior can appear confusing

FPとintegerの変換は処理系依存: pythonの場合

常識と一致しないことがある

- int(x) は truncate（0 方向）である

  -1.5　　-1.0　-0.5　0　　0.5　1.0　1.5

  | 0方向に切り詰め =>|　　| <= 0方向に切り詰め|

  - 正の値では直感と一致
  - 負の値では直感とズレる
  - 誤差があると 0 に吸い寄せられる
- round(x) は数学的丸め
  - 正負どちらでも「最も近い整数」へ
  - **int(round(x)) が「普通の丸め込み」**
- 浮動小数点誤差が丸めに影響する
  - -1.0 が -0.9999999999999993 になる
  - その結果、int() が 0 を返すことがある
- 剰余 % は常に非負
  - -0.2 % 1.0 = 0.8
  - これは数学的な剰余とは違う
  - Python の仕様として覚える必要がある
- 正負の丸め込みは直感とズレるので注意
  - 特に負の値は「わけわからない」挙動になりやすい

Program: bad_int.py

```
> python bad_int.py
Summing up 0.01 for 100 times: v = 1.0000000000000007
int(v)          = 1
int(v + eps)  = 1
int(round(v))= 1
 Used eps=-1e-10

Summing up -0.01 for 100 times: v = -1.0000000000000007
int(v)          = -1
int(v + eps)  = 0
int(round(v))= -1
 Used eps=1e-10

Summing up -0.01 for 100 times: v = -1.0000000000000007
int(v)          = -1
int(v + eps)  = -1
int(round(v))= -1
 Used eps=-1e-10
```

All cases return 1

Return -1 and 0

All cases return -1

You may expect:

int(1.0000001) = 1
int(0.0000001) = 0
int(-0.9999999)=-1
int(-1.0000001)=-1

Case for floating point to integer conversion (浮動小数点 => 整数変換):

Bad (悪い例):

```
n = int(v)
```

Good (良い例):

```
n = round(v)
```

Summary: int_conversion.py

input x	int(x)	round(x)	int(round(x))	floor(x)
0.00001	0	0	0	0
-0.00001	0	0	0	-1
0.4999	0	0	0	0
0.5001	0	1	1	1
1.4999	1	1	1	1
-1.5001	-1	-2	-2	-2
-0.5001	0	-1	-1	-1
-0.4999	0	0	0	-1
-1.4999	-1	-1	-1	-2

Wrap fractional coordinate to [0, 1)

input x	int(x)	floor(x)	x - int(x)	x - floor(x)	x % 1.0
2.2	2	2	0.2	0.2	0.2
1.2	1	1	0.2	0.2	0.2
0.2	0	0	0.2	0.2	0.2
-0.2	0	-1	-0.2	0.8	0.8
-1.2	-1	-2	-0.2	0.8	0.8
-2.2	-2	-3	-0.2	0.8	0.8

Wrap fractional coordinate to [0, 1)

Python's best:

```
def wrap01(v):
    return v % 1.0
```

For some programming languages like python:
For fractional coordinates in a unit cell :
Use v % 1.0 for safely reducing v to the range [0, 1).
単位格子の部分座標などでよくある範囲還元の注意:
v % 1.0 を使うと安全に [0, 1) 範囲に還元できる

In case floor() is available:

```
def wrap01(v):
    return v - floor(v)
```

Other cases (only int() is available):

```
def wrap01(v):
    v_wrapped = v - int(v)
    if v_wrapped < 0.0:
        return v_wrapped + 1.0
    return v_wrapped
```

see wrap_fractional_coord.py

Loops using floating-point values(浮動小数点型によるループ)

Bad (悪い例):

```
import numpy as np
for v in np.arange(start, end, step):
    …
```

np.arange(start, end, step):
  Iterate from start while v < end,

Caution: The end value is expected not to be included, but it may be included due to floating-point error
  (endの値は実行されないことが想定されているが、FPの誤差のため、endが実行される可能性がある)

Better method #1: range(N)

```
# Choose a sufficiently small eps value, e.g., 1.0e-6, to compute the loop count N w/o affected by floating-point errors.
  (eps として十分小さい値 (例えば 1.0e-6) をとることでFP誤差に影響されないループ回数 N を計算する)
N = int((end – start) / step + eps)
for i in range(N):
    v = start + i * step
    …
```

Better method #2: np.linspace(start, end, N)

```
for v in np.linspace(start, end, N):    # always execute the end value (常にendを含む)
```

Typical cases for FP calculations with care

Evaluate possible errors every time for FP floating point calculations

- Error originates from the limited length of FP type: underflow, overflow
  Representation range of 64bit FP (IEEE 754 standard)
  Exponent: 11 bit -1024 ~ +1023
  Fraction : 23 bit 4,503,599,627,370,495: 16 digits
- **Most decimal fractions cannot be represented exactly in binary floating-point format**
- **Most of FP values in computer include errors**
  1.0/3.0 = 0.3333…333 (16 digits) Error ~ 10^{-16} should be included

Conditional branch:

Bad: if x * 10.0 == 30.0: **No guarantee to get the correct judge 'true'** even if x = 3.0

Good: eps = 1.0e-30 # epsilon: A value satisfactory smaller than expected values bur larger than the precision of the variable type.
if **abs(x * 10.0 - 30.0) < eps**: **Gives the correct judge within the error of eps**

FP => integer conversion:

How to calculate the number of division in the range xmin – xmax at xstep step

Bad: n = int((xmax – xmin) / xstep): The value in int() can include error.
Even if the correct value is n = 3.0,
you will get n = 2 if int() becomes 2.99999… due to error,.

Good:

eps = 1.0e-6
n = int((xmax – xmin) / xstep + eps)
Even if (xmax – xmin) / xstep becomes smaller than the expected integer due to erro,
you can receive the correct value as long as the error is smaller than eps.

数値演算プログラムの一般的な注意

浮動小数点型の演算では、常に誤差を意識すること

・変数長の制限による誤差: underflow, overflow

IEEE 754の標準で、64bit浮動小数点の範囲は

指数部: 11 bit -1024 ~ +1023

仮数部: 23 bit 4,503,599,627,370,495: 16桁

・浮動小数点では、多くの実数を “正確に” 表現できない

・有限の桁数の浮動小数点の表現は、ほぼすべての場合に誤差を含む

1.0/3.0 ＝ 0.3333…333 (小数点以下16桁)　　10^{-16} 程度の誤差が発生する

条件分岐の判断:

悪い例: if x * 10.0 == 30.0:　x = 3.0 であっても、**true と判断される保証はない**

良い例: eps = 1.0e-6　# epsilon: 想定される誤差よりも十分大きいが、変数型が判別できるなるべく小さい値を設定する。

if **abs(x * 10.0 - 30.0) < eps**:　**誤差 eps 以内で必ず実行される**

浮動小数点 => 整数変換: xmin ~ xmax の範囲を xstep 毎の幅で分割したときの分点の数

悪い例:

```
n = int( (xmax – xmin) / xstep):
```

(xmax – xmin) / xstepが誤差により 2.99999… となった場合、

本来はint() = 3 となって欲しいのに、2 となってしまう

良い例:

```
eps = 1.0e-6
n = int( (xmax – xmin) / xstep + eps):
```

(xmax – xmin) / xstepが誤差により期待する整数値より小さくなっても、

誤差がepsより小さければ、本来期待している整数値が得られる

Precision and errors in computer

Data bit width (データ長): Determine the upper limit of precision

=> Roundoff (rounding) error (丸め誤差)

Other error sources

- **Overflow (積み残し誤差, 桁あふれ):**

  e.g. by summation between large integers (有効桁数を超える整数の和・積)

  ⇔ **underflow**

  (overflow and underflow can be detected by CPU / software
  but may deteriorate calculation speed)

- **Roundoff error (丸め誤差): By subtracting very similar values**

  ex: for 4 digits calculation:

  $5\sqrt{41} - 32 \sim 5*6.403 - 32.00 = 32.015 - 32.00 = 32.02 - 32.00 = 0.02$

  The given values have 4 significant digits
  but the result has only 1 significant digits

  Avoid subtraction between similar large values

- **Loss of trailing digits (情報落ち):**

  by summing / subtracting between largely-different values

  ex: 1000 + 1.456 = 1001 (The initial significant value of .456 is lost)

Catastrophic cancellation (桁落ち･情報埋没)

Program: python information_buried.py

***e.g.*, calculate exp(-40) by $\exp(x) = \sum_{n=0} x^n/n!$**

Summing up large values with opposite signs results in significant errors (正負が交番する大きな数の和を取るために誤差が大きくなる)

Better to add positive values only

$\mathbf{A} = \sum_{n=0}^{N}(-x)^n/n!$ (if $x < 0$)

, and take $\frac{1}{A} = \exp(-40)$

Exact value 4.24835425529159×10^{-18}

N :	$\sum_{n=0}^{N} x^n/n!$	$1.0/\sum_{n=0}^{N}(-x)^n/n!$
0 :	1	1
1 :	-39	0.024390244
2 :	761	0.0011890606
18 :	7.3620174e+12	5.290335e-14
19 :	-1.5234693e+13	2.4096905e-14
20 :	2.9958728e+13	1.153502e-14
21 :	-5.6123978e+13	5.7878667e-15
22 :	1.0039003e+14	3.0368438e-15
23 :	-1.7180825e+14	1.6625449e-15
	Sum up large +/- values	
79 :	-1.3651644e+09	**4.2483543e-18**
115:	5.8811462	4.2483543e-18
116:	5.8811665	4.2483543e-18

Well converged,

but **18 digits of error!!**

Errors in calculation process

- Overflow (summing large values)
- Underflow (huge numbers of summing up small values)
- Round-off error
- **Truncation error (打切り誤差)**

  Summing terms that slowly approach zero:

  - **Taylor expansion (テーラー展開)**
  - **Summation of Coulomb energy (Coulombエネルギーの和)**

  Need to terminate the summation if calculation time has limitation or the result reaches the required precision

  (計算時間と必要な精度に応じて、どこかで計算を打ち切る)

- **Convergence error (収束誤差)**

  A required precision, often denoted by EPS, is used as a stopping criterion for iterative calculations.

- **Errors originating from physical model (物理モデルの誤差)**