Computational Physics

Numerical Accuracy

Prof. Paul Eugenio
Department of Physics
Florida State University
12 Feb 2019

http://hadron.physics.fsu.edu/~eugenio/comphy/

Announcements

- **Exercise 3**
  - Due date extended to today Feb 12
- **Exercise 4**
  - Due date Friday Feb 15
- **Reading**
  - Chapter 5 Sections 5.1 – 5.5 on Integrals and Derivatives (pages 140 – 165)
    - Turn-In Questions
      - 2 Questions on the reading due next Tuesday

Numerical Computing

- Representing Numbers
  - Bits, Bytes, and Words
  - Fixed Points (int or long)
  - Floating Points (float, complex)
- Floating Point Arithmetic
- Computational Errors
  - Range Errors
  - Round-Off Errors

Representing Numbers

- Binary bits
  - Units of Memory
  - All Numbers Eventually are Represented in Binary Form
    - Finite Precision → Limits & Approximations
- Word length
  - Number of Bits to Store a Number
    - Often Given in Bytes where:

      1 byte = 1B = 8 bits = 8 b

      1 kB = 2^{10} bytes = 1024bytes

      1 MB = 1024*1024 bytes

Fixed Points (int and long)

- Python int type
  - 2^N-1 integers represented by N bits
    - 1st bit gives the sign, remaining N-1 bits give the value
      - 64-bit integer:

$$-9223372036854775806 \leq \text{int}_{64} \leq 9223372036854775807$$

- Python long type
  - integers of unlimited size

```
hpc-login 618% python
>>> a = 12312312352348578236412634287364087236508127650182736458034765018734 58
>>> a
12312312352348578236412634287364087236508127650182736458034765018734 58L
>>> bin(a)
'0b1011011010101100111101010101101011111010100001000000000111000001011010011100110101010011011000010111111110011100110101010000011010110001101100100100100010110101000101010111010100011011111110010010011000001011111110100001100110010'
>>> a.bit_length()
230
```

Python does not have a problem with integer overflows or underflows

Floating Points (float)

- Scientific work mainly uses floating-point numbers
- Floating-Point Notation
  - $x = (-1)^s \times \text{Mantissa} \times 2^{exp}$

$mantissa = m_1 2^{-1} + m_2 2^{-2} + \ldots$

$bias = 0111\ 1111_2 = 127_{10}$

	s	exp+bias	Mantissa	
$float_{64}$	0	00001111111	10000000000000000000000000000000000000000000000000	$= \frac{1}{2}$

Range

$\pm 2.48 \times 10^{-324} \leq float_{64} < \pm 1.8 \times 10^{308}$

Precision

$float_{64}$: 16 decimals

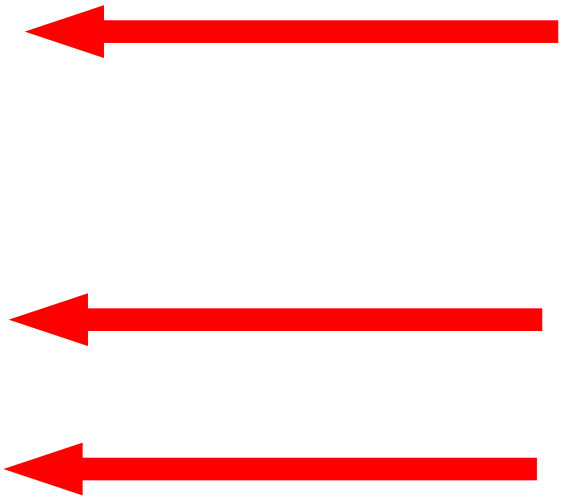
Don't confuse Python's $float_{64}$ with $float_{32}$ used in other programming languages

Computational Errors

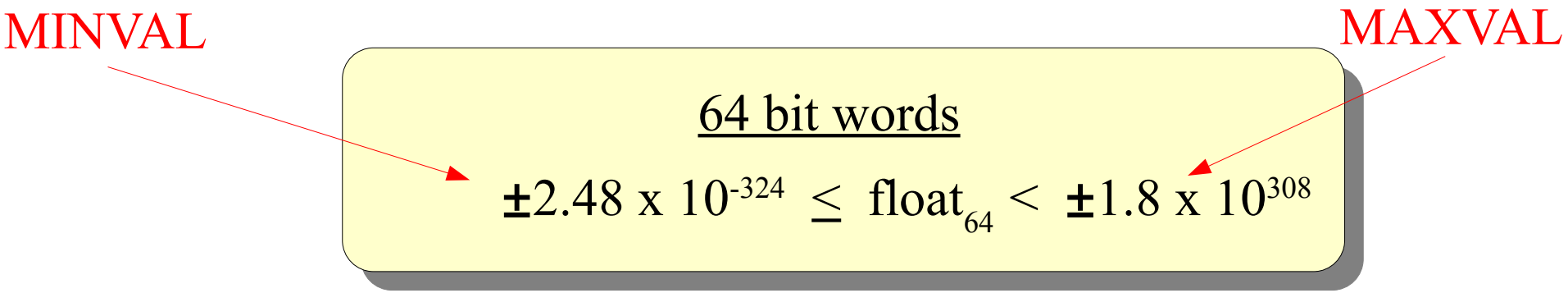
- Human Errors
  - Blunders
- Random Errors
  - Acts of Nature
- Approximation Errors

$$e^x \simeq \sum_n^N (-x)^n/n!$$

- Range Errors
- Round-off Errors



Range Errors



- If a number **x** is larger than the MAXVAL, an overflow occurs
- If a number **x** is smaller than the MINVAL, an underflow occurs

The resulting value may be a 0, inf, -inf, or nan (not a number).

```
hpc-login 618% python
>>> z = 2.48e-324
>>> print(z)
5e-324
>>> z = 2.47e-324
>>> print(z)
0.0
```

```
hpc-login 618% python
>>> x = 1.7e308
>>> print(x)
1.7e+308
>>> x = 1.8e308
>>> print(x)
inf
```

Floating Point Precision

- Floating Point numbers are stored as binary fractions and not decimal fractions
  - Example: 0.1254 = b0.001

0.125 = 1/10 + 2/100 + 5/1000

b0.001 = 0/2 + 0/4 + 1/8

Floating Point Precision

- Floating Point numbers are stored as binary fractions and not decimal fractions
  - Example: 0.1254 = b0.001

    0.125 = 1/10 + 2/100 + 5/1000

    b0.001 = 0/2 + 0/4 + 1/8

- Most decimal fractions cannot be represented exactly as binary fractions

```
hpc-login 618% python
>>> 0.1
0.1
>>> 0.2
0.2
>>> 0.1 + 0.2 == 0.3
False
>>> 0.1 + 0.2
0.30000000000000004
```

Floating Point Arithmetic

- Floating Point is NOT exact
  - example: adding floats

$$\begin{array}{rl} & 100000 \times 2^{10} \\ + & 123456 \times 2^{8} \\ \hline \end{array}$$

Floating Point Arithmetic

- Floating Point is NOT exact
  - example: adding floats

$$\begin{array}{ll} & 100000 \times 2^{10} \\ + & 012345 \times 2^{9} \\ \hline \end{array}$$

exponential shifting

Floating Point Arithmetic

- Floating Point is NOT exact
  - example: adding floats

$$\begin{array}{rl} & 100000 \times 2^{10} \\ + & 001234 \times 2^{10} \\ \hline \end{array}$$

exponential shifting

Floating Point Arithmetic

- Floating Point is NOT exact
  - example: adding floats

$$\begin{array}{rl} & 100000 \times 2^{10} \\ + & 123456 \times 2^{8} \\ \hline & 101234.56 \times 2^{10} \end{array}$$ exact

$$\begin{array}{rl} & 100000 \times 2^{10} \\ + & 001234 \times 2^{10} \\ \hline & 101234 \times 2^{10} \end{array}$$

exponential shifting

low-order bits are lost

Round-Off Error

Round-Off Errors

- Example: A Simple Sum $S = \sum \frac{1}{n}$

$$S_{up} = \sum_{n=1}^{N} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$$

round-off error

Round-Off Errors

- Example: A Simple Sum $S = \sum \frac{1}{n}$

$$S_{up} = \sum_{n=1}^{N} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$$

round-off error

$$S_{down} = \sum_{n=N}^{1} \frac{1}{n} = \frac{1}{N} + \frac{1}{N-1} + \frac{1}{N-2} + \ldots + 1$$

$$S_{up} \neq S_{down}$$

more precise

Round-Off Errors & Accuracy

- Machine Accuracy ϵ

The largest number such that
$1.0 + \epsilon = 1.0$

computational value $\rightarrow$ $x_c = x\,(1 + \epsilon_x)$ $\quad |\epsilon_x| \leq \epsilon$

true value

Subtractive Cancelation Errors of Round-Off Errors

- Subtractive Cancelation

$$a = b - c \quad \rightarrow \quad a_c = b_c - c_c \qquad\qquad a_c = a(1+\epsilon_a)$$

$$a(1+\epsilon_a) = b(1+\epsilon_b) - c(1+\epsilon_c)$$

Subtractive Cancelation Errors of Round-Off Errors

- Subtractive Cancelation

$$a = b - c \quad \rightarrow \quad a_c = b_c - c_c \qquad a_c = a(1+\epsilon_a)$$

$$a(1+\epsilon_a) = b(1+\epsilon_b) - c(1+\epsilon_c)$$

$$\epsilon_a = \epsilon_b b/a - \epsilon_c c/a - 1 + (b - c)/a$$

-1 + 1 = 0

$$\boxed{\epsilon_a = \epsilon_b b/a - \epsilon_c c/a}$$

Subtractive Cancelation Errors of Round-Off Errors

- Subtractive Cancelation

$$a = b - c \quad \rightarrow \quad a_c = b_c - c_c \qquad a_c = a(1+\epsilon_a)$$

$$a(1+\epsilon_a) = b(1+\epsilon_b) - c(1+\epsilon_c)$$

$$\epsilon_a = \epsilon_b b/a - \epsilon_c c/a - 1 + (b - c)/a$$

-1 + 1 = 0

$$\boxed{\epsilon_a = \epsilon_b b/a - \epsilon_c c/a}$$

if a is small then $b \approx c$

$$\epsilon_a \approx b/a(|\epsilon_b| + |\epsilon_b|) \qquad |\epsilon_i| \leq \epsilon$$

If you subtract two large numbers and end up with a small one, the result will have a large uncertainty.

Subtractive Cancelation Errors

- Example:

$$a x^2 + b x + c = 0$$

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x'_{1,2} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}$$

if $b^2 >> 4ac$

then for:

$b>0$	x_1 & x'_2	are less accurate	→ use x_2 & x'_1
$b<0$	x_2 & x'_1	are less accurate	→ use x_1 & x'_2

Round-Off Errors in Multiplication of Floating Point Numbers

- Multiplicative Errors

$$a = b * c \quad \rightarrow \quad a_c = b_c * c_c$$

$$a_c = a(1+\epsilon_a)$$

$$a(1+\epsilon_a) = b(1+\epsilon_b) * c(1+\epsilon_c)$$

$$1+\epsilon_a = 1+\epsilon_b+\epsilon_c+\epsilon_b\epsilon_c \rightarrow 0$$

$$\boxed{\epsilon_a = \epsilon_b + \epsilon_c}$$

Since ϵ_b and ϵ_c can have opposite signs the total error can be larger or smaller than the individual errors, but the overall distribution of errors is larger

In general, for calculations use Floating Precision

- Scientific programming uses mainly floating-point type numbers.
  - As a general rule, when in doubt, use floating-point numbers.
    - this is true more so in C, C++, ...
  - Be aware that in some cases integer arithmetic is more precise.

Wait! Python has a Decimal Module

- The decimal module has user settable precision
  - Floating points have up 15 decimal places
- The decimal module also preserves significance.

Why not use decimals every time rather than floats?

The main reasons are **efficiency** and increased **complexity**. Floating point operations are carried out much, Much, MUCH FASTER than Decimal operations.

When to use Decimal instead of Float

- Use in financial applications that need exact decimal representation.
  - When we want to control the level of precision required.
  - When we want to implement the notion of significant decimal places.

```
hpc-login 711% python
>>> from decimal import Decimal
>>> Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal(0.1) + Decimal(0.2) == Decimal(0.3)
False
>>> Decimal('0.10')
Decimal('0.10')
>>> Decimal('0.1') + Decimal('0.2') == Decimal('0.3')
True
```

Math Constants

Python Math Module

math.pi	**3.141592653589793238462**
math.e	**2.718281828459045235360**

Predefined Double Precision in C & C++

```
/* Some useful constants defined in /usr/include/math.h  */

# define M_E          2.7182818284590452354   /* e */
# define M_LOG2E       1.4426950408889634074   /* log_2 e */
# define M_LOG10E      0.43429448190325182765  /* log_10 e */
# define M_LN2         0.69314718055994530942  /* log_e 2 */
# define M_LN10        2.30258509299404568402  /* log_e 10 */
# define M_PI          3.14159265358979323846  /* pi */
# define M_PI_2        1.57079632679489661923  /* pi/2 */
# define M_PI_4        0.78539816339744830962  /* pi/4 */
# define M_1_PI        0.31830988618379067154  /* 1/pi */
# define M_2_PI        0.63661977236758134308  /* 2/pi */
# define M_2_SQRTPI    1.12837916709551257390  /* 2/sqrt(pi) */
# define M_SQRT2       1.41421356237309504880  /* sqrt(2) */
# define M_SQRT1_2     0.70710678118654752440  /* 1/sqrt(2) */
```

Constants: scipy.constants

```
hpc-login 169% python
>>> import scipy.constants as sc

>>> print("c:", sc.c, "hbar:", sc.hbar, "1/alpha:", 1/sc.alpha)
c: 299792458.0 hbar: 1.05457172534e-34 1/alpha: 137.035999074
```

See: `pydoc scipy.constants`

```
hpc-login 170% pydoc scipy.constants
Help on package scipy.constants in scipy:
NAME
    scipy.constants
FILE
    /usr/lib64/python2.7/site-packages/scipy/constants/__init__.py
DESCRIPTION
     ...
    Mathematical constants
    ======================
    ============
================================================================
    ``pi``         Pi
    ``golden``     Golden ratio
    ============
================================================================

    Physical constants
    ==================
    ``c``            speed of light in vacuum
    ``mu_0``         the magnetic constant :math:`\mu_0`
    ``epsilon_0``    the electric constant (vacuum permittivity),
:math:`\epsilon_0`
    ``h``            the Planck constant :math:`h`
    ``hbar``         :math:`\hbar = h/(2\pi)`
    ``G``            Newtonian constant of gravitation
    ``g``            standard acceleration of gravity
    ``e``            elementary charge
```

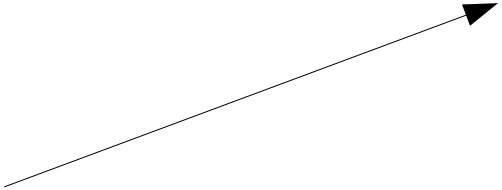
NumPy Reference

http://docs.scipy.org/doc/numpy/reference/

Routines

Array creation routines
Array manipulation routines
Binary operations
String operations
C-Types Foreign Function Interface (numpy.ctypeslib)
Datetime Support Functions
Data type routines
Optionally Scipy-accelerated routines (numpy.dual)
Mathematical functions with automatic domain (numpy.emath)
Floating point error handling
Discrete Fourier Transform (numpy.fft)
Financial functions
Functional programming

Indexing routines
Input and output
Linear algebra (numpy.linalg)
Logic functions
Masked array operations
Mathematical functions
Matrix library (numpy.matlib)
Miscellaneous routines
Padding Arrays
Polynomials
Random sampling (numpy.random)
Set routines
Sorting, searching, and counting
Statistics



Statistics

Order statistics

`amin` (a[, axis, out, keepdims])	Return the minimum of an array or minimum along an axis.
`amax` (a[, axis, out, keepdims])	Return the maximum of an array or maximum along an axis.
`nanmin` (a[, axis, out, keepdims])	Return minimum of an array or minimum along an axis, ignoring any NaNs.
`nanmax` (a[, axis, out, keepdims])	Return the maximum of an array or maximum along an axis, ignoring any NaNs.
`ptp` (a[, axis, out])	Range of values (maximum - minimum) along an axis.
`percentile` (a, q[, axis, out, ...])	Compute the qth percentile of the data along the specified axis.
`nanpercentile` (a, q[, axis, out, ...])	Compute the qth percentile of the data along the specified axis, while ignoring nan values.

Averages and variances

`median` (a[, axis, out, overwrite_input, keepdims])	Compute the median along the specified axis.
`average` (a[, axis, weights, returned])	Compute the weighted average along the specified axis.
`mean` (a[, axis, dtype, out, keepdims])	Compute the arithmetic mean along the specified axis.
`std` (a[, axis, dtype, out, ddof, keepdims])	Compute the standard deviation along the specified axis.
`var` (a[, axis, dtype, out, ddof, keepdims])	Compute the variance along the specified axis.
`nanmedian` (a[, axis, out, overwrite_input, ...])	Compute the median along the specified axis, while ignoring NaNs.
`nanmean` (a[, axis, dtype, out, keepdims])	Compute the arithmetic mean along the specified axis, ignoring NaNs.
`nanstd` (a[, axis, dtype, out, ddof, keepdims])	Compute the standard deviation along the specified axis, while ignoring NaNs.
`nanvar` (a[, axis, dtype, out, ddof, keepdims])	Compute the variance along the specified axis, while ignoring NaNs.

Correlating

`corrcoef` (x[, y, rowvar, bias, ddof])	Return Pearson product-moment correlation coefficients.
`correlate` (a, v[, mode])	Cross-correlation of two 1-dimensional sequences.
`cov` (m[, y, rowvar, bias, ddof, fweights, ...])	Estimate a covariance matrix, given data and weights.

Histograms

`histogram` (a[, bins, range, normed, weights, ...])	Compute the histogram of a set of data.
`histogram2d` (x, y[, bins, range, normed, weights])	Compute the bi-dimensional histogram of two data samples.
`histogramdd` (sample[, bins, range, normed, ...])	Compute the multidimensional histogram of some data.
`bincount` (x[, weights, minlength])	Count number of occurrences of each value in array of non-negative ints.
`digitize` (x, bins[, right])	Return the indices of the bins to which each value in input array belongs.

Let's get working