PHY407: Computational Physics
Fall, 2017

Instructor: Paul Kushner

TAs: Heather Fong, Alex Cabaj, Haruki Hirasawa

Lecture 2

- Greetings/check-in
- Numerical errors
- Integration and approximation (algorithmic) errors.

What are some sources of error in computational physics?

What are some sources of error in computational physics?

- Input errors
- Errors in setting up the discretized mathematical model (conceptual/mathematical)
- Algorithm errors
  - Roundoff/numerical
  - Approximation errors
  - Ill-conditioned methods (instability)
- Output errors

What are some sources of error in computational physics?

- Input errors
- Errors in setting up the discretized mathematical model (conceptual/mathematical)
- Algorithm errors
  - Roundoff/numerical
  - Approximation errors
  - Ill-conditioned methods (instability)
- Output errors

This week.

Remember that computers are machines with limitations

ENIAC (1946):
30 tonnes
100 kHz, 400 byte memory
18,000 vacuum tubes

http://www.computerhistory.org/revolution/b
rth-of-the-computer/4/78/319

Apple iPhone8 (2017):
148g ~ 10^{-6} ENIACs
Hexa core 1.4GHz ~ 10^{4} ENIACs
256MB ~ 10^{6} ENIACs

http://www.gsmarena.com/apple_iphone_8-8573.php

Recall: Round-off Error

- Under what circumstances is the following possible?

$$(x+y)+z \neq x + (y + z)$$

Look at this Python session:

```
>>> x = 1.0e20
>>> y = -1.0e20
>>> z = 1.0
>>> (x+y)+z
1.0
>>> x + (y+z)
0.0
```

Round-off error in this step!

What is a floating point number?

- In a floating point number, the number of digits before or after the decimal place is not fixed (it “floats”).
- A nice discussion of floating point numbers can be found at http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#689 and in the Sirca and Horvat textbook available online through the U of T library.
- Floating point arithmetic on a machine is limited by the number of bits in your floating point numbers.
- This limit can be imposed on the machine, or you can limit it yourself.
- E.g. in `numpy` you can work in 32-bit (single precision, `float32`) or 64-bit (double precision, `float64`).
- There are standards listed on the next slide.

Limitations on Floats in Computers

Table 1.1 The smallest and largest exponents and approximate values of some important numbers representable in single- and double-precision floating-point arithmetic in base two, according to the IEEE 754 standard. Only positive values are listed

Precision	Single ("`float`")	Double ("`double`")
$e_{\max}$	127	1023
$e_{\min} = 1 - e_{\max}$	-126	-1022
Smallest normal number	$\approx 1.18 \times 10^{-38}$	$\approx 2.23 \times 10^{-308}$
Largest normal number	$\approx 3.40 \times 10^{38}$	$\approx 1.80 \times 10^{308}$
Smallest representable number	$\approx 1.40 \times 10^{-45}$	$\approx 4.94 \times 10^{-324}$
Machine precision, ε_{M}	$\approx 1.19 \times 10^{-7}$	$\approx 2.22 \times 10^{-16}$
Format size	32 bits	64 bits

You can access these properties on your machine.

Go to `FunWithFloats.py`

Sirca and Horvat

FunWithFloats.py | MachineError.py | RoundoffErrorPower.py

```python
from numpy import finfo, float64, float32
print "attributes you can access in finfo(float64) ", dir(finfo(float64))
print "maximum numbers in 64 bit and 32 bit precision: ", finfo(float64).max, finfo(float32).max
print "minimum numbers in 64 bit and 32 bit precision: ", finfo(float64).min, finfo(float32).min
print "epsilon for 64 bit and 32 bit: ", finfo(float64).eps, finfo(float32).eps
print "Should be epsilon for this machine if it's 64 bit", float64(1)+finfo(float64).eps-float64(1)
print "Should be zero", float64(1)+finfo(float64).eps/2.0-float64(1)
```

FunWithFloats.py | MachineError.py | RoundoffErrorPower.py

```python
from numpy import finfo, float64, float32
print "attributes you can access in finfo(float64) ", dir(finfo(float64))
print "maximum numbers in 64 bit and 32 bit precision: ", finfo(float64).max, finfo(float32).max
print "minimum numbers in 64 bit and 32 bit precision: ", finfo(float64).min, finfo(float32).min
print "epsilon for 64 bit and 32 bit: ", finfo(float64).eps, finfo(float32).eps
print "Should be epsilon for this machine if it's 64 bit", float64(1)+finfo(float64).eps-float64(1)
print "Should be zero", float64(1)+finfo(float64).eps/2.0-float64(1)
```

Debug I/O #2 | Python Shell #1 | Python Shell #2

Commands execute without debug. Use arrow keys for history. Options

```
[evaluate FunWithFloats.py]
attributes you can access in finfo(float64)  ['__class__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_finfo_cache', '_init', '_str_eps', '_str_epsneg', '_str_max', '_str_resolution',
'_str_tiny', 'dtype', 'eps', 'epsneg', 'iexp', 'machar', 'machep', 'max', 'maxexp', 'min', 'minexp',
'negep', 'nexp', 'nmant', 'precision', 'resolution', 'tiny']
maximum numbers in 64 bit and 32 bit precision:  1.79769313486e+308 3.40282e+38
minimum numbers in 64 bit and 32 bit precision:  -1.79769313486e+308 -3.40282e+38
epsilon for 64 bit and 32 bit:  2.22044604925e-16 1.19209e-07
Should be epsilon for this machine if it's 64 bit 2.22044604925e-16
Should be zero 0.0
>>>
```

FunWithFloats.py | MachineError.py | RoundoffErrorPower.py

```python
from numpy import finfo, float64, float32
print "attributes you can access in finfo(float64) ", dir(finfo(float64))
print "maximum numbers in 64 bit and 32 bit precision: ", finfo(float64).max, finfo(float32).max
print "minimum numbers in 64 bit and 32 bit precision: ", finfo(float64).min, finfo(float32).min
print "epsilon for 64 bit and 32 bit: ", finfo(float64).eps, finfo(float32).eps
print "Should be epsilon for this machine if it's 64 bit", float64(1)+finfo(float64).eps-float64(1)
print "Should be zero", float64(1)+finfo(float64).eps/2.0-float64(1)
```

Debug I/O #2 | Python Shell #1 | Python Shell #2

Commands execute without debug. Use arrow keys for history. Options

```
[evaluate FunWithFloats.py]
attributes you can access in finfo(float64)  ['__class__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_finfo_cache', '_init', '_str_eps', '_str_epsneg', '_str_max', '_str_resolution',
'_str_tiny', 'dtype', 'eps', 'epsneg', 'iexp', 'machar', 'machep', 'max', 'maxexp', 'min', 'minexp',
'negep', 'nexp', 'nmant', 'precision', 'resolution', 'tiny']
maximum numbers in 64 bit and 32 bit precision:  1.79769313486e+308 3.40282e+38
minimum numbers in 64 bit and 32 bit precision:  -1.79769313486e+308 -3.40282e+38
epsilon for 64 bit and 32 bit:  2.22044604925e-16 1.19209e-07
Should be epsilon for this machine if it's 64 bit 2.22044604925e-16
Should be zero 0.0
>>>
```

- On 64-bit computers, there will always be rounding after 16 significant figures.
- Don't assume that `7.0/3.0-4.0/3.0-1.0` will result in `0.0`! (Try it.)

Error Propagation Examples

- Newman defines the error constant as

$$\sigma = Cx$$

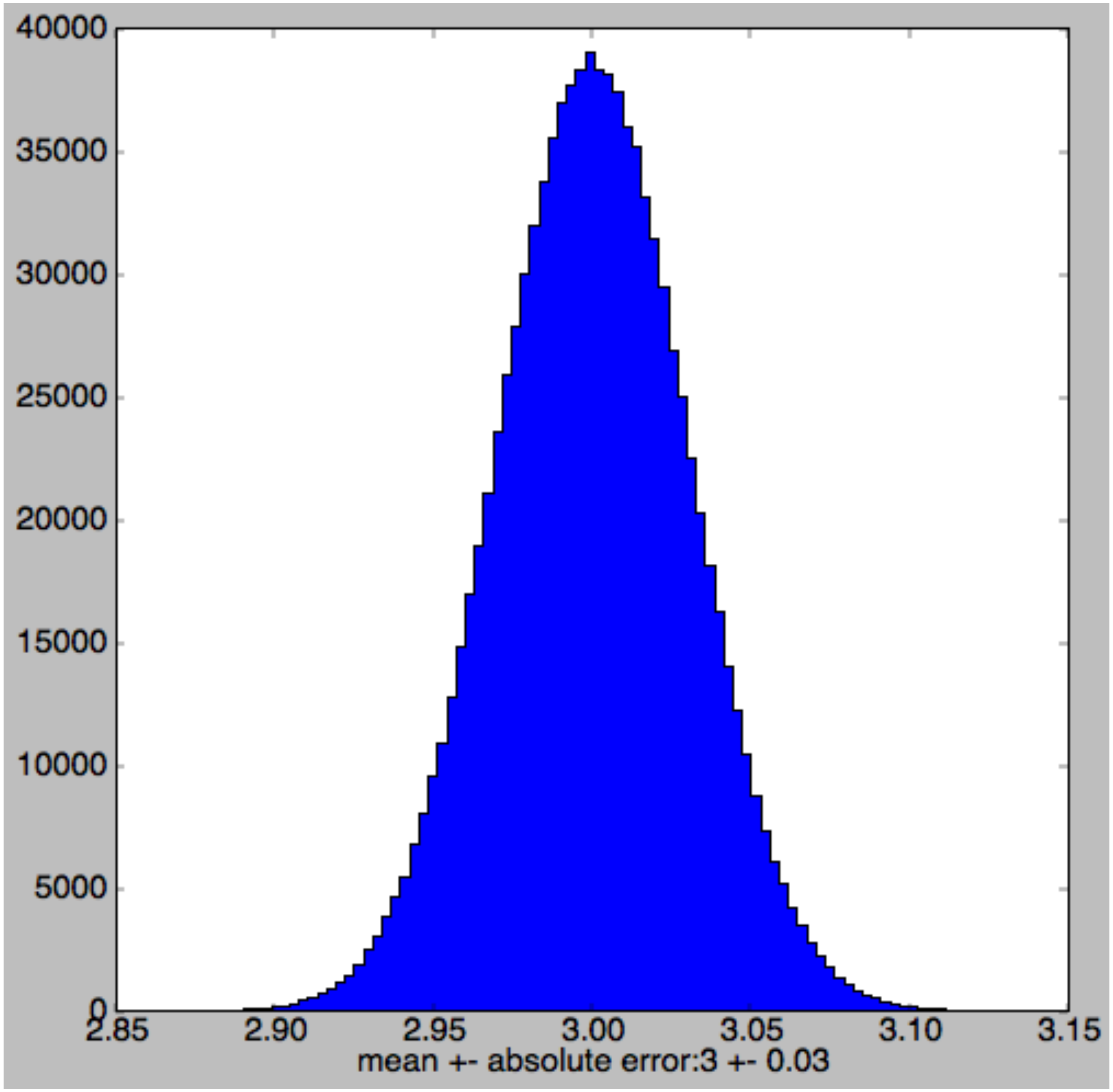
- The *error constant* C is the fractional error for a single floating point number.
- For a 64 bit float, $C \sim 10^{-16}$ and is similar to the machine precision $\varepsilon_{M.}$
- We simply can't know a number better than this on the computer (otherwise it wouldn't be a limit on the precision).
- This fractional error is different on different computers but should be independent from number to number.
- Errors propagate statistically like they do in experimental physics.

Impact of 1% Error

Suppose C was 10^{-2}?

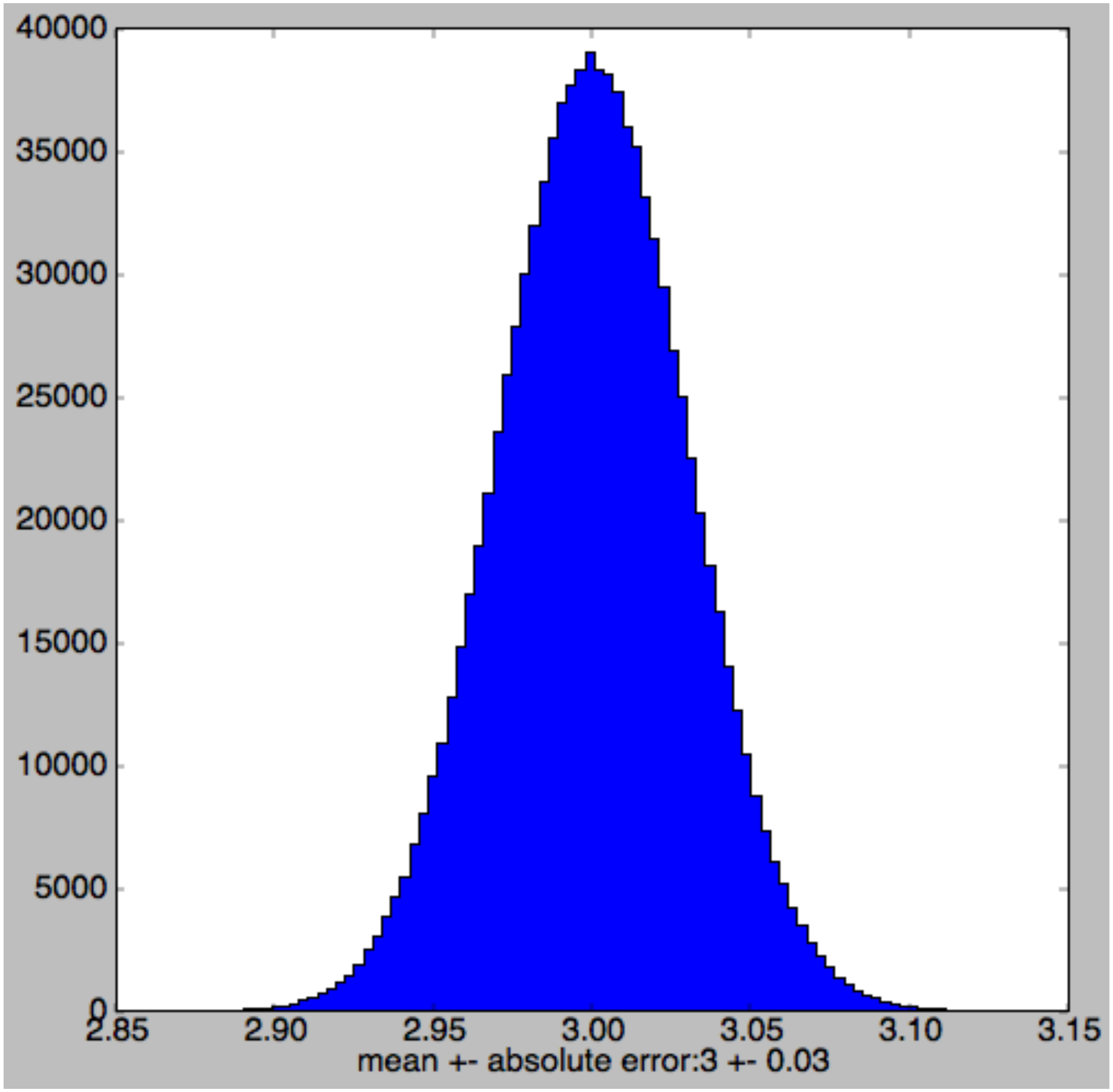
Impact of 1% Error

Suppose C was 10^{-2}?



Impact of 1% Error

Suppose C was 10^{-2}?



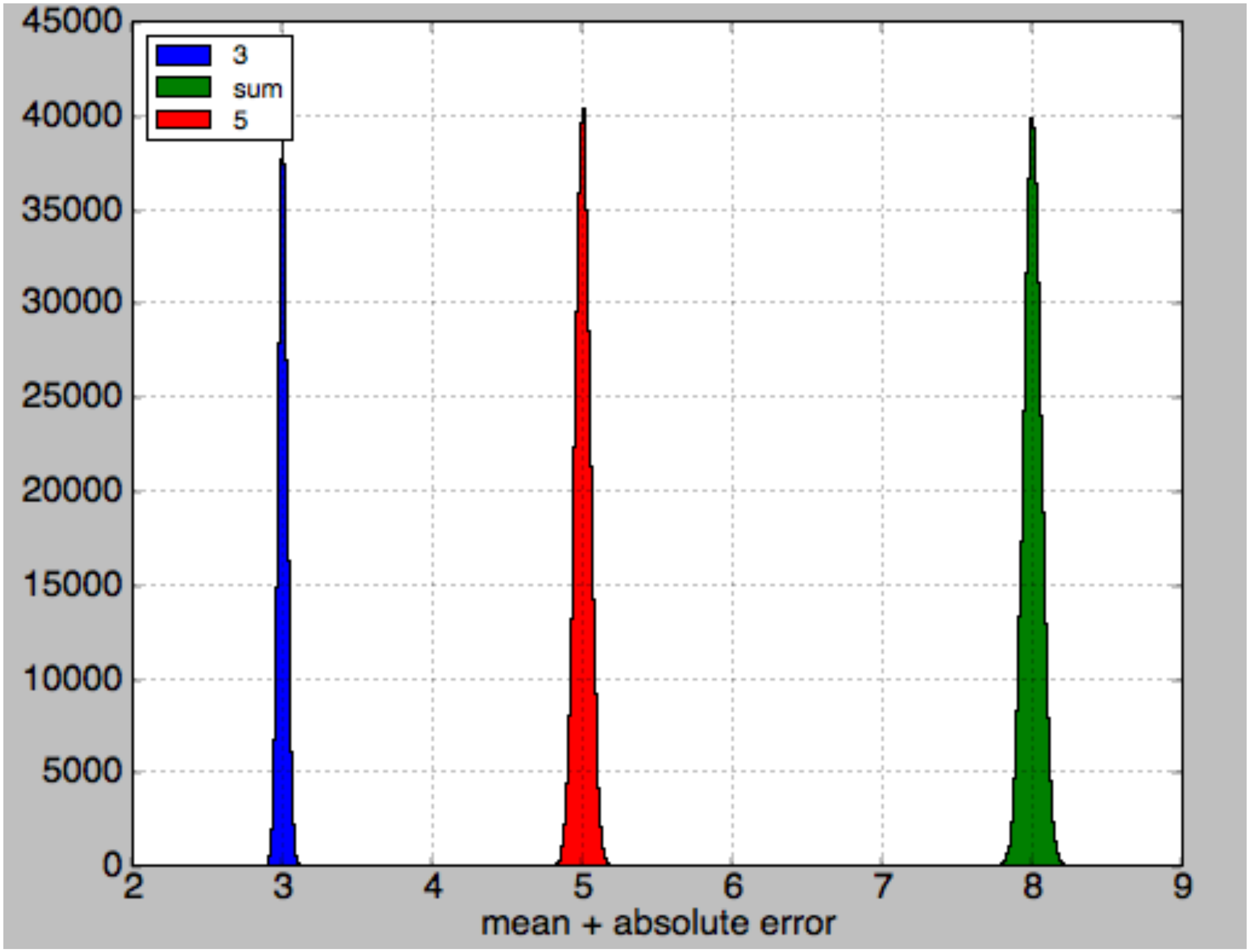
This could be the distribution you'd get when evaluating "3.0" across an ensemble of computers.

Error Propagation

What if we add a couple of numbers?

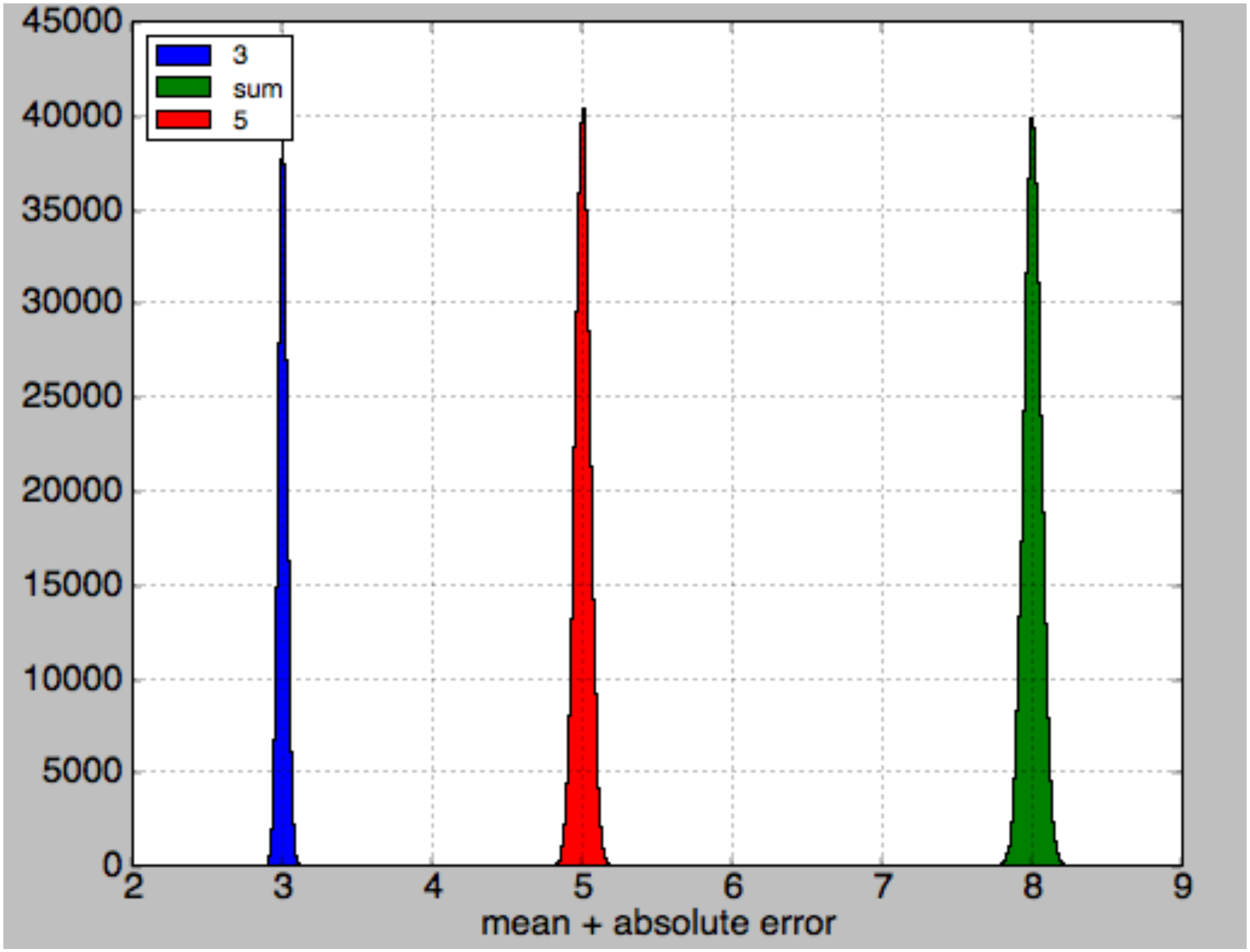
Error Propagation

What if we add a couple of numbers?



Error Propagation

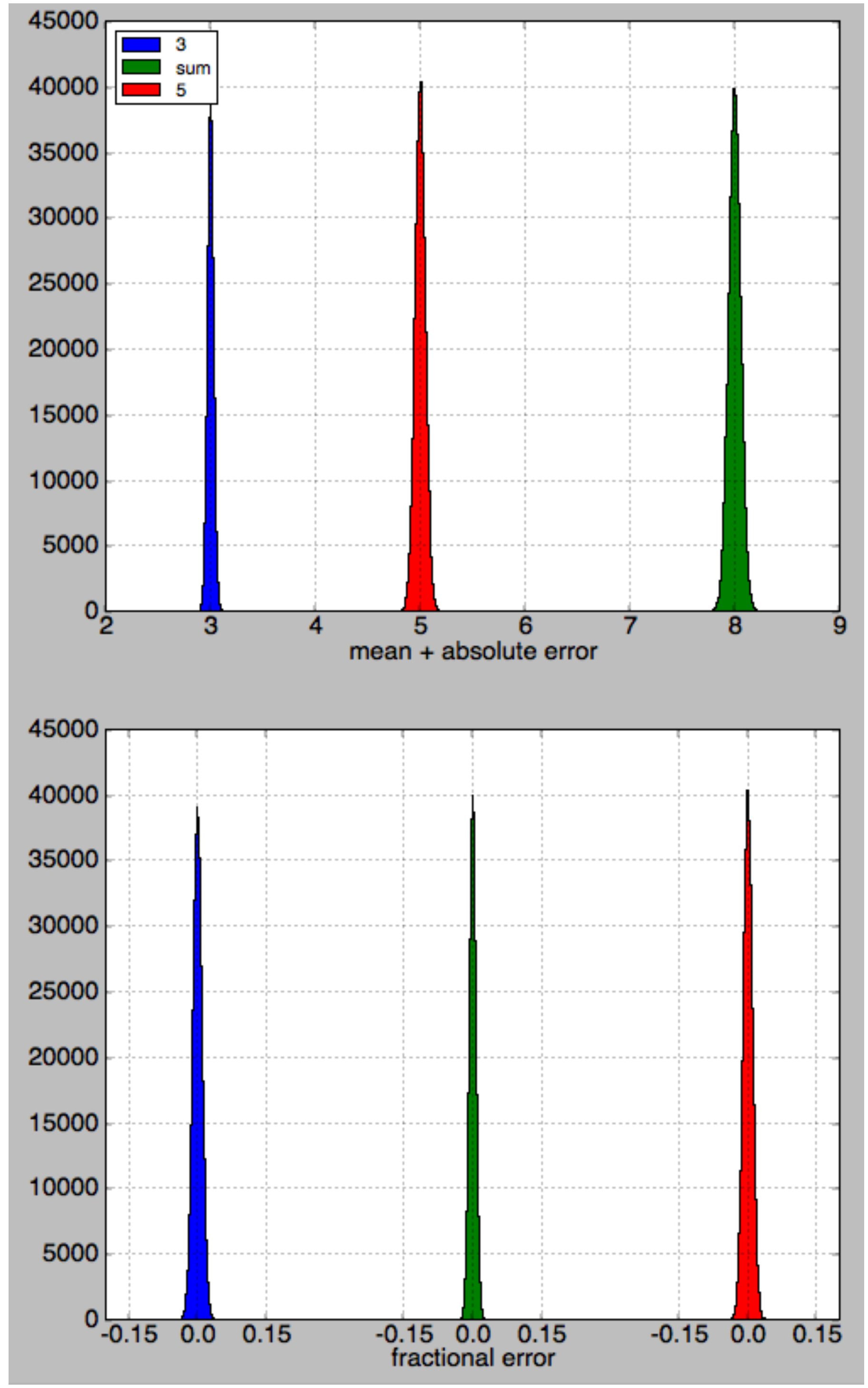
What if we add a couple of numbers?



Errors propagate indepen-dently, like errors in the lab.

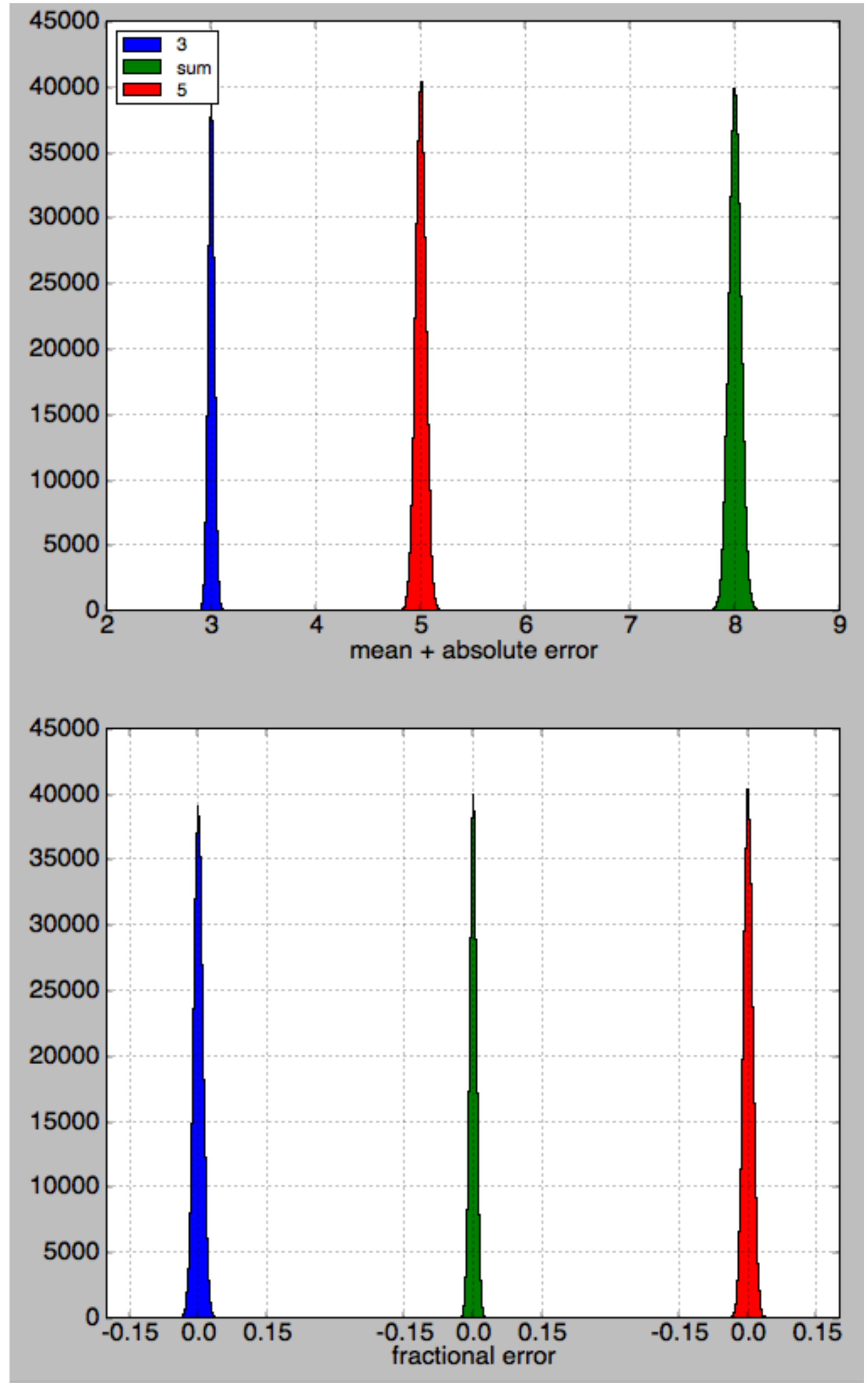
The largest error dominates the sum.

We can also calculate the fractional error (in the bottom graph).



We can also calculate the fractional error (in the bottom graph).

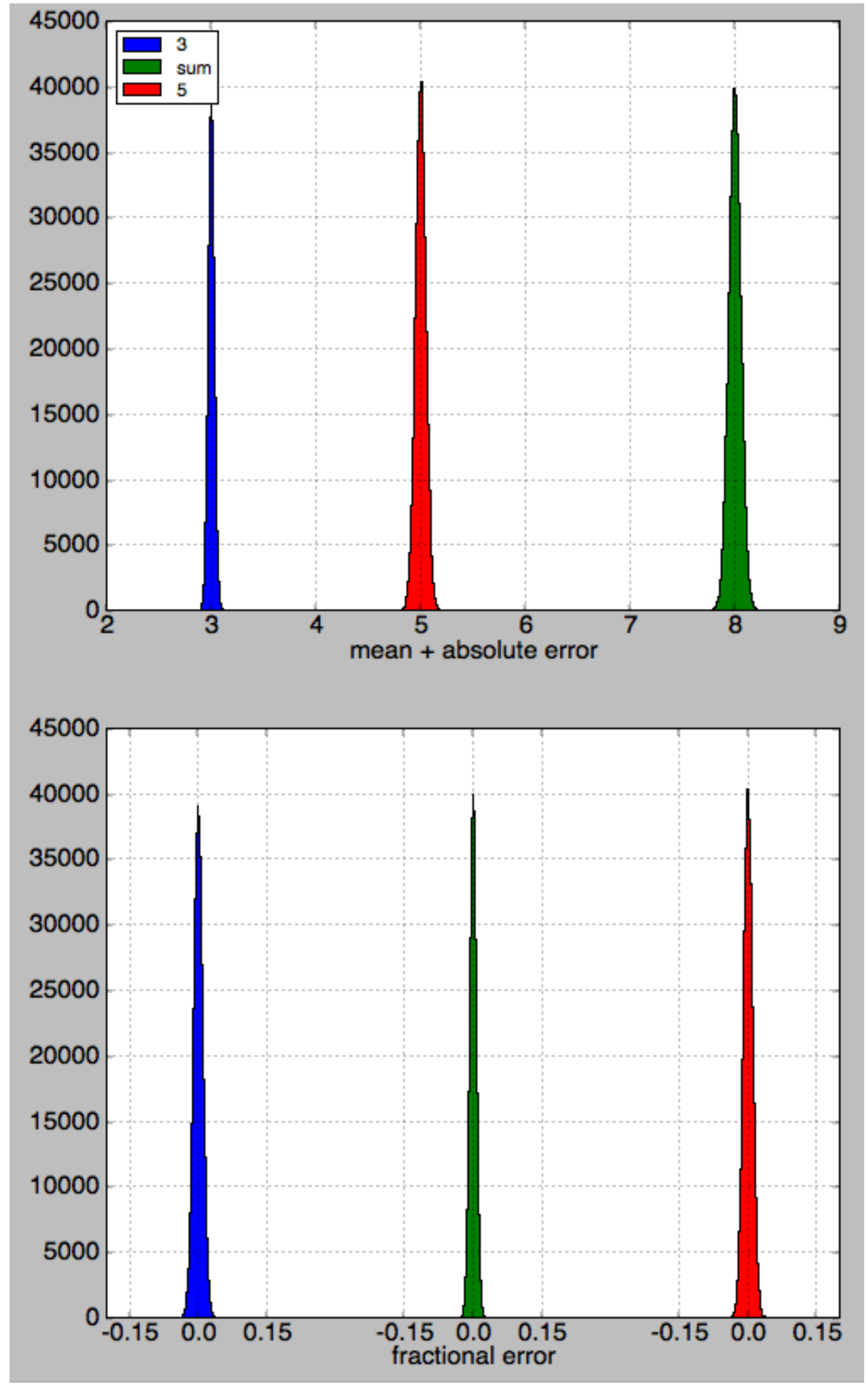
In this example, it's about the same fractional uncertainty for the two numbers and their sum.



We can also calculate the fractional error (in the bottom graph).

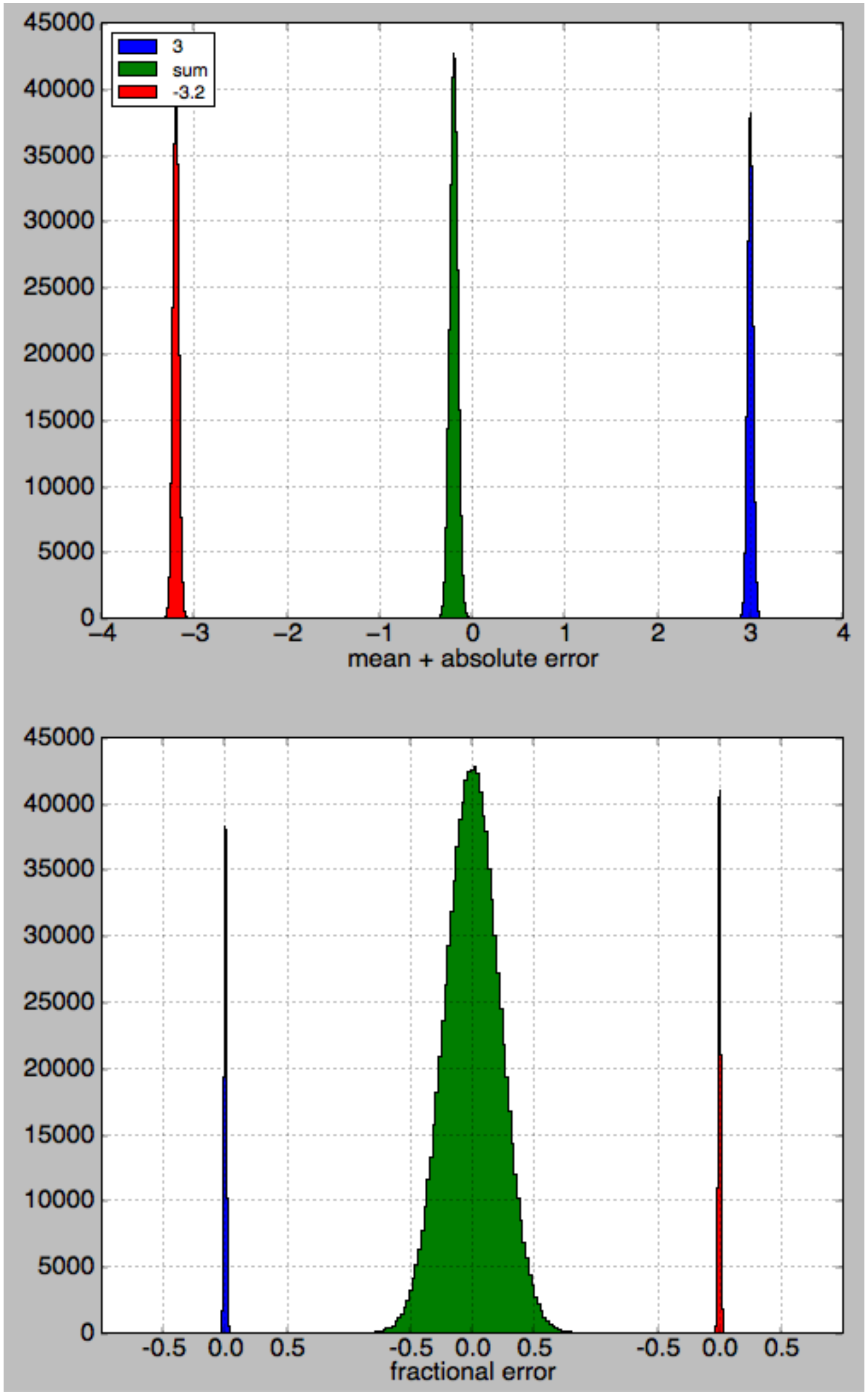
In this example, it's about the same fractional uncertainty for the two numbers and their sum.

But fractional errors can be a lot worse if we subtract large numbers!



In this example, we add 3.0 and -3.2 along with their associated uncertainty.

The error in the sum is around 50% - it's a little hard to tell that the sum is different from zero.



Hey! How did you do those calculations?

- See `MachineError.py`

```
#import normal distribution
from numpy.random import normal
#define array size
N = 1000000
#define C, which is our simulated error constant
C=1e-2
#define numbers
(x1,x2) = (3,-3.2)
#define errors in terms of C
sigma1 = C*abs(x1)
sigma2 = C*abs(x2)
#define distributions to those numbers satisfying sigma = Cx
#This is how we simulate error.
d1  = normal(loc=x1, scale=sigma1, size=N)
d2  = normal(loc=x2, scale=sigma2, size=N)
#then add up the distributions
#then calculate the distribution of the sums.
sumd = d1 + d2
```

Take Home Message

- Don't count on exact evaluation of floating point numbers on your computer!
- In particular, never include branch logic like this

```
if (float(x) == 0.0):
  print 'hello world'
else ...
```

Take Home Message

- Don’t count on exact evaluation of floating point numbers on your computer!
- In particular, never include branch logic like this

```
if (float(x) == 0.0):
  print ‘hello world’
else ...
```

- Instead, use something like

```
if (abs(float(x)) < 1e-10):
  print ‘hello world’
else ...
```

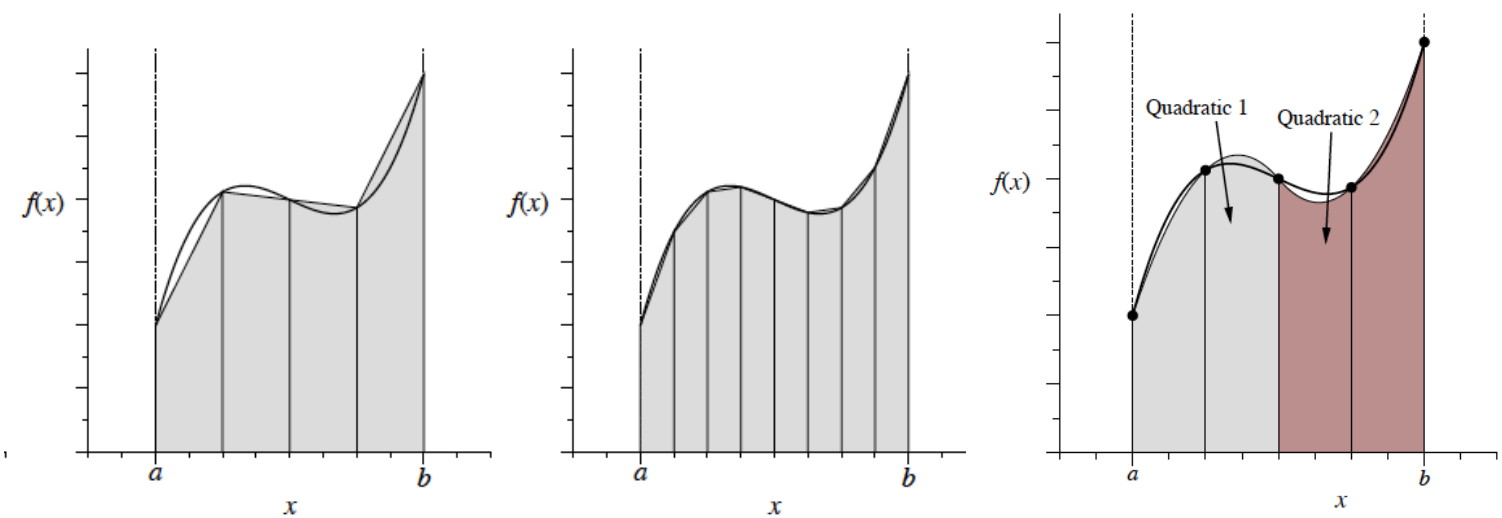
Numerical Integration

- This is our first discussion of numerical methods.
- The best numerical methods are
  - Accurate, stable, robust
  - Easy on resources: fast, not too much memory
  - Simple (subjective).
- Rule of thumb:
  - For many quick-and-dirty calculations, simple is best.
  - More sophistication is often required for more accurate or faster methods.
- Be ready to understand, defend, and document a numerical method's accuracy and resource requirements.

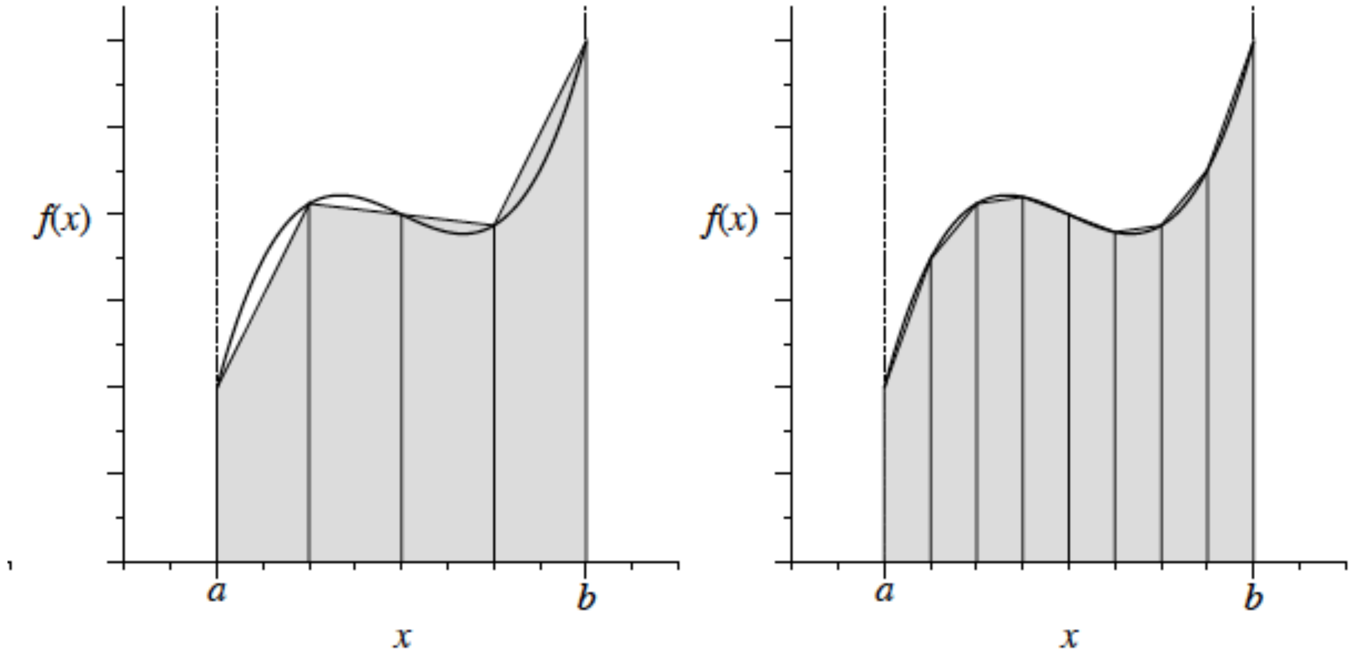
Numerical Integration

- Think of integrals as areas under curves or surfaces.
- Approximate these areas in terms of simple shapes (rectangles, trapezoids, rectangles with parabolic tops)

$$\int_a^b f(x)\,dx \approx$$



Trapezoidal Rule



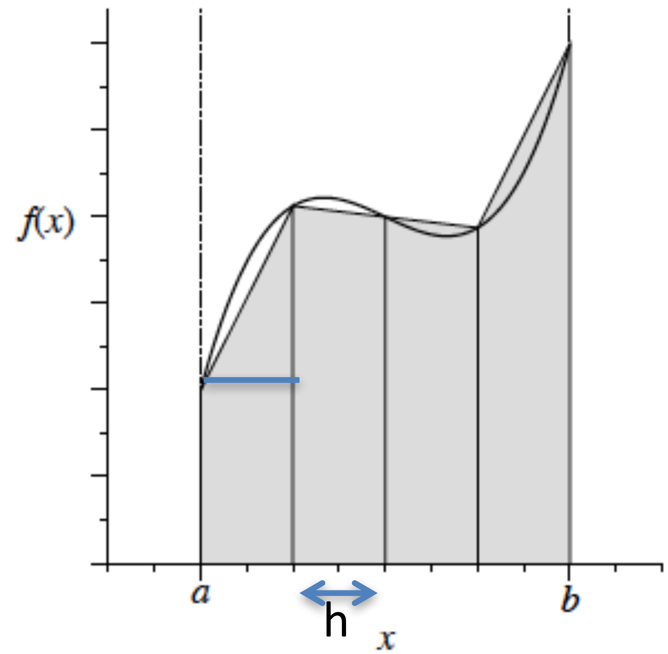
- Break up interval into N equal slices.
- Approximate function as a line segment in each slice.
- More slices → better approximation to function

Trapezoidal Rule

- N slices from a to b means that slice width:

  h = (b – a)/N

- area of ‘k’ slice’s trapezoid: (Rectangle + Triangle)



$$A_k = f(x_k) * h + 0.5\ h *[\ f(x_k+h)-f(x_k)]$$
$$=0.5 * h *[f(x_k) + f(x_k+ h)]$$

- To cover the interval from a to b: each slide edge on the inside of the interval (i.e. not at a or b) gets included in 2 areas → multiply those by 2

- Total area (our approximation for the integral) (and using x_k=a+k*h):

$$I(a,b) = h\left[\tfrac{1}{2}f(a) + \tfrac{1}{2}f(b) + \sum_{k=1}^{N-1} f(a+kh)\right]$$

Trapezoidal Rule

$$A_k = h * [0.5*f(a) + 0.5*f(b) + \Sigma_{k=1}^{N-1}[f(x_k) + f(x_k + h)]$$

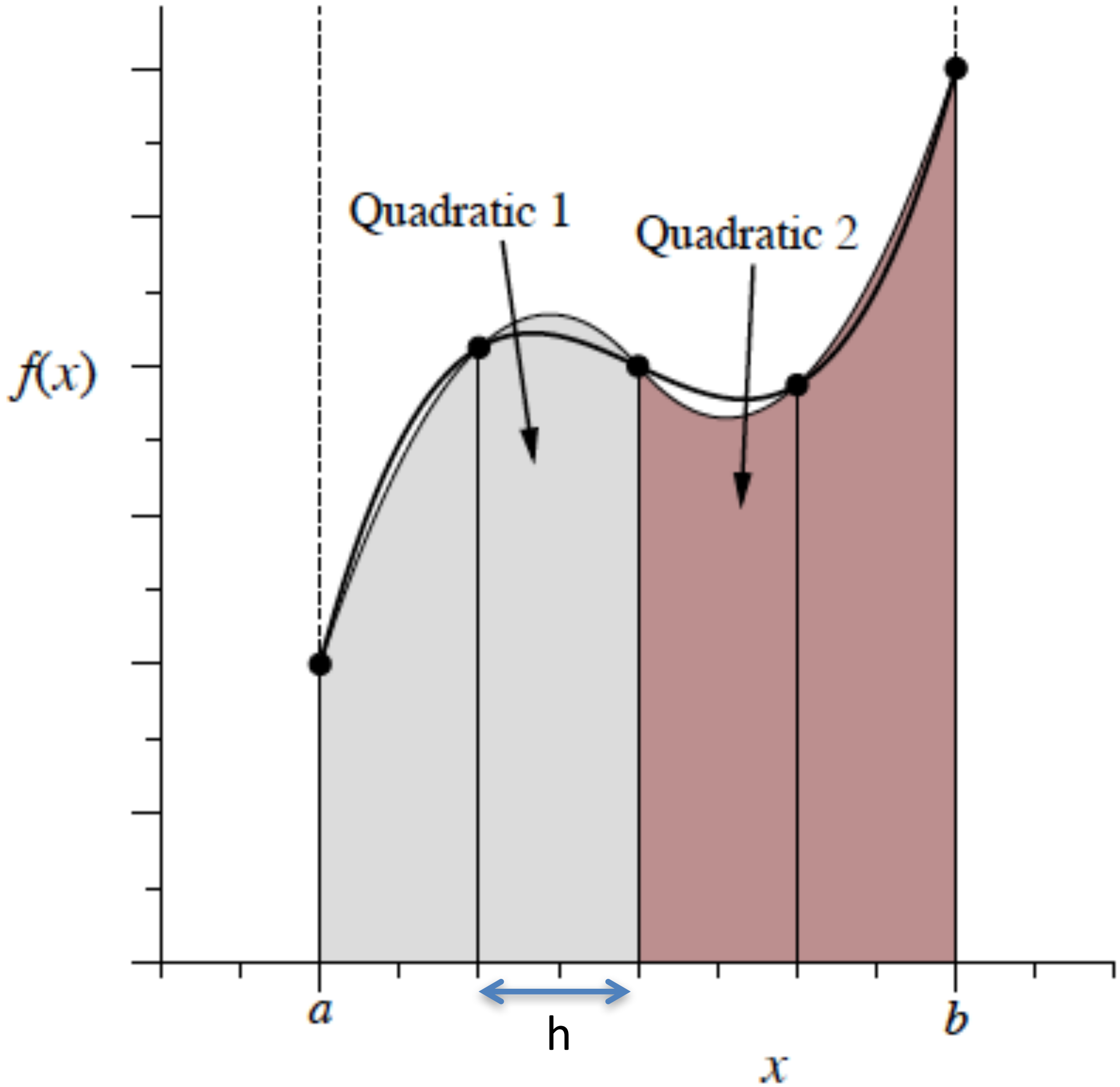
```
def f(x):
   return x**4 - 2*x + 1

N = 10
a = 0.0
b = 2.0
h = (b-a)/N

s = 0.5*f(a) + 0.5*f(b)
for k in range(1,N):
   s += f(a+k*h)

print(h*s)
```

here are the edge points

here are all the values at the interior points being added up

Simpson's Rule

- Again, break up interval into N equal slices of width h
- Approximate function as a QUADRATIC for every **2** slices
  - Need 2 slices because you need 3 points to define a quadratic
- More slices → better approximation to function



Simpson's Rule

- Area of each 2-slice quadratic (see text for how to determine quadratic from 3 point evaluations):

$$A_k = \tfrac{1}{3}h\left[f(a+(2k-2)h) + 4f(a+(2k-1)h) + f(a+2kh)\right]$$

- Adding up all the slices gives:

$$I(a,b) = \tfrac{1}{3}h\left[f(a) + f(b) + 4\sum_{\substack{k\ \text{odd}\\ 1\ldots N-1}} f(a+kh) + 2\sum_{\substack{k\ \text{even}\\ 2\ldots N-2}} f(a+kh)\right]$$

- Notice your sums are over even and odd k values. In python you can implement this in a for loop with:

```
for k in range(1,N,2)    #for the odd terms

for k in range(2,N,2)  #for the even terms
```

Error Estimation

- What do we mean by an error on an integral?

Error Estimation

- What do we mean by an error on an integral?

$$I = \int_a^b f(x)\,dx = \frac{1}{2} h \sum_{k=1}^{N} [f(x_{k-1}) + f(x_k)] + \varepsilon$$

Error Estimation

- What do we mean by an error on an integral?

Error specific to that method

$$I = \int_a^b f(x)\,dx = \frac{1}{2}h\sum_{k=1}^{N}[f(x_{k-1}) + f(x_k)] + \boxed{\varepsilon}$$

Method, like trapezoidal rule.

Error Estimation

- Trapezoidal rule is a "first-order" integration rule, i.e. accurate up to and including terms proportional to *h*. Leading order approximation error is of order h^2

$$\boxed{\varepsilon = \frac{1}{12}h^2\left[f'(a) - f'(b)\right] + h.o.t}$$

- Simpson's rule is a "third-order" integration rule, i.e. accurate up to and including terms proportional to h^3 . Leading order approximation error is of order h^4 (even though the order of the polynomial is only 1 degree higher!)

$$\boxed{\varepsilon = \frac{1}{\cancel{90}}h^4\left[f'''(a) - f'''(b)\right] + h.o.t}$$

180

This is the formula in Newman. The correct formula should have 180 instead of 90.

Error Estimation

- Trapezoidal rule is a "first-order" integration rule, i.e. accurate up to and including terms proportional to *h*. Leading order approximation error is of order h^2

$$\varepsilon = \frac{1}{12} h^2 \left[f'(a) - f'(b) \right] + h.o.t$$

Trapezoidal Rule approaches machine precision for $N \sim 10^7 - 10^8$

- Simpson's rule is a "third-order" integration rule, i.e. accurate up to and including terms proportional to h^3 . Leading order approximation error is of order h^4 (even though the order of the polynomial is only 1 degree higher!)

$$\varepsilon = \frac{1}{\cancel{90}} h^4 \left[f'''(a) - f'''(b) \right] + h.o.t$$

180

This is the formula in Newman. The correct formula should have 180 instead of 90.

Error Estimation

- Trapezoidal rule is a "first-order" integration rule, i.e. accurate up to and including terms proportional to *h*. Leading order approximation error is of order h^2

$$\boxed{\varepsilon = \frac{1}{12} h^2 \left[f'(a) - f'(b) \right] + h.o.t}$$

Trapezoidal Rule approaches machine precision for $N \sim 10^7 - 10^8$

- Simpson's rule is a "third-order" integration rule, i.e. accurate up to and including terms proportional to h^3 . Leading order approximation error is of order h^4 (even though the order of the polynomial is only 1 degree higher!)

$$\boxed{\varepsilon = \frac{1}{\cancel{90}} h^4 \left[f'''(a) - f'''(b) \right] + h.o.t}$$

180

Machine precision for $N \sim 10^3 - 10^4$

Newman: Doubling Rule for Practical Integral Estimation

- The error estimate for the Trapezoidal Rule is

$$\boxed{\varepsilon = \frac{1}{12}h^2\left[f'(a) - f'(b)\right] + h.o.t \approx Ch^2}$$

- Even if you don't know the function you are integrating over, you can still estimate *C:*
  - *Calculate the integral at N and 2N, then*

$$\varepsilon = \frac{1}{3}(I_2 - I_1)$$

Summary & Status

- ☑ Covered some python basics and pseudocoding …
- ☑ Finished our first lab and pre-lab 02
- ☑ Discussed roundoff error and simple integration method.
- ❑ Tutorial 02 and Lab 02: Wednesday lab, due Friday
  - Programming tips, roundoff error, numerical integration, differentiation.
  - Feedback on Lab01.
- ❑ Lab 03 released this week, Lecture 3 Monday, Tutorial 03 Wednesday:
  - More numerical integration and differentiation: Gaussian quadrature etc..
- ❑ Also coming up: Assign and discuss class project.