

Computational Physics

Lecture 3

sieversj@ukzn.ac.za

git clone https://github.com/ukzncompphys/lecture3_2018.git

tutorial problems available from:

https://github.com/ukzncompphys/tutorials_2018

Due Monday Apr. 23

Variables

```
Jonathans-MacBook-Pro:lecture3 sievers$ cat simple_variables.py
x=5
y=3.0
z=x+y
print 'z is ' + repr(z)
print 'z is a ' + repr(type(z))
print 'x is a ' + repr(type(x))
print 'y is a ' + repr(type(y))

Jonathans-MacBook-Pro:lecture3 sievers$ python simple_variables.py
z is 8.0
z is a <type 'float'>
x is a <type 'int'>
y is a <type 'float'>
Jonathans-MacBook-Pro:lecture3 sievers$
```

- Python has several built-in variable types. These include strings, floats, ints, boolean.
- You can create variables by assigning to them.
- You can see what type variables are with the *type* command.
- You can print their values with the *print* command. *repr* will return value as a string
- strings can be combined with '+'. Types may be automatically converted.

```
>>> z=3
>>> print 'z is ',z
z is 3
```

Numpy

- Python has no built in arrays. Use NUMerical PYthon (numpy) for math functionality
- Many functions work on numpy arrays - usually much (much) faster than going through interpreter
- *numpy.arange* works like *range*, but returns numpy array instead of a list.

note - *numpy.arange* with three arguments will space values by 3rd argument.

```
Jonathans-MacBook-Pro:lecture3 sievers$ cat numpy_example.py
import numpy
x=numpy.arange(0,10)
print x**2
print 'type(x) is ' + repr(type(x))

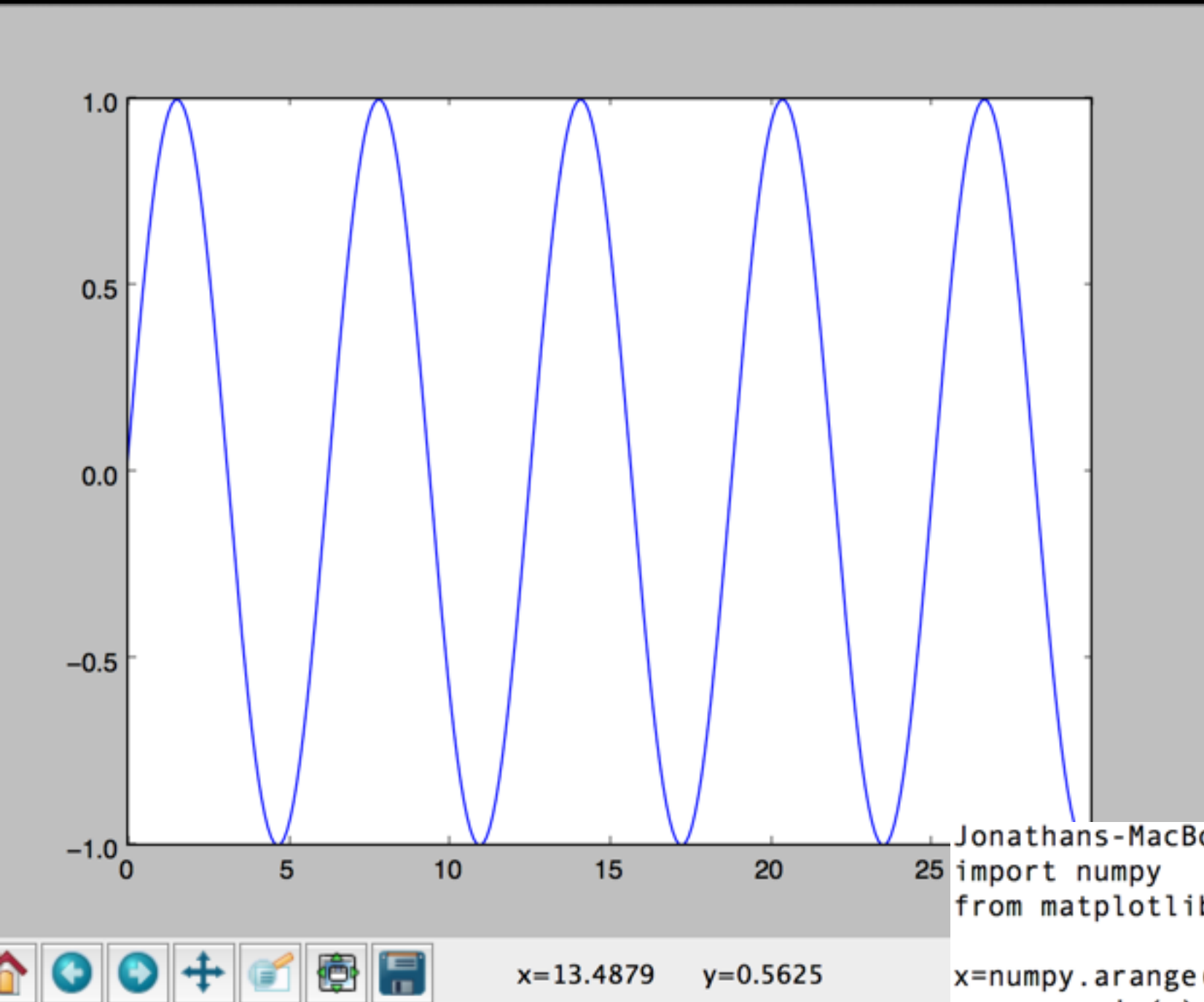
x=range(0,10)
print 'type(x) is now ' + repr(type(x))
print x**2

Jonathans-MacBook-Pro:lecture3 sievers$ python numpy_example.py
[ 0  1  4  9 16 25 36 49 64 81]
type(x) is <type 'numpy.ndarray'>
type(x) is now <type 'list'>
Traceback (most recent call last):
  File "numpy_example.py", line 8, in <module>
    print x**2
TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
Jonathans-MacBook-Pro:lecture3 sievers$
```

Plotting

- Let's make a plot! Say, $\sin(x)$ from 0 to 30
- How would we do this? To the tubes!

Example



```
Jonathans-MacBook-Pro:lecture3 sievers$ cat matplotlib_example.py
```

```
import numpy
from matplotlib import pyplot as plt
```

```
x=numpy.arange(0,30,0.1)
y=numpy.sin(x)
```

```
plt.plot(x,y)
plt.show()
```

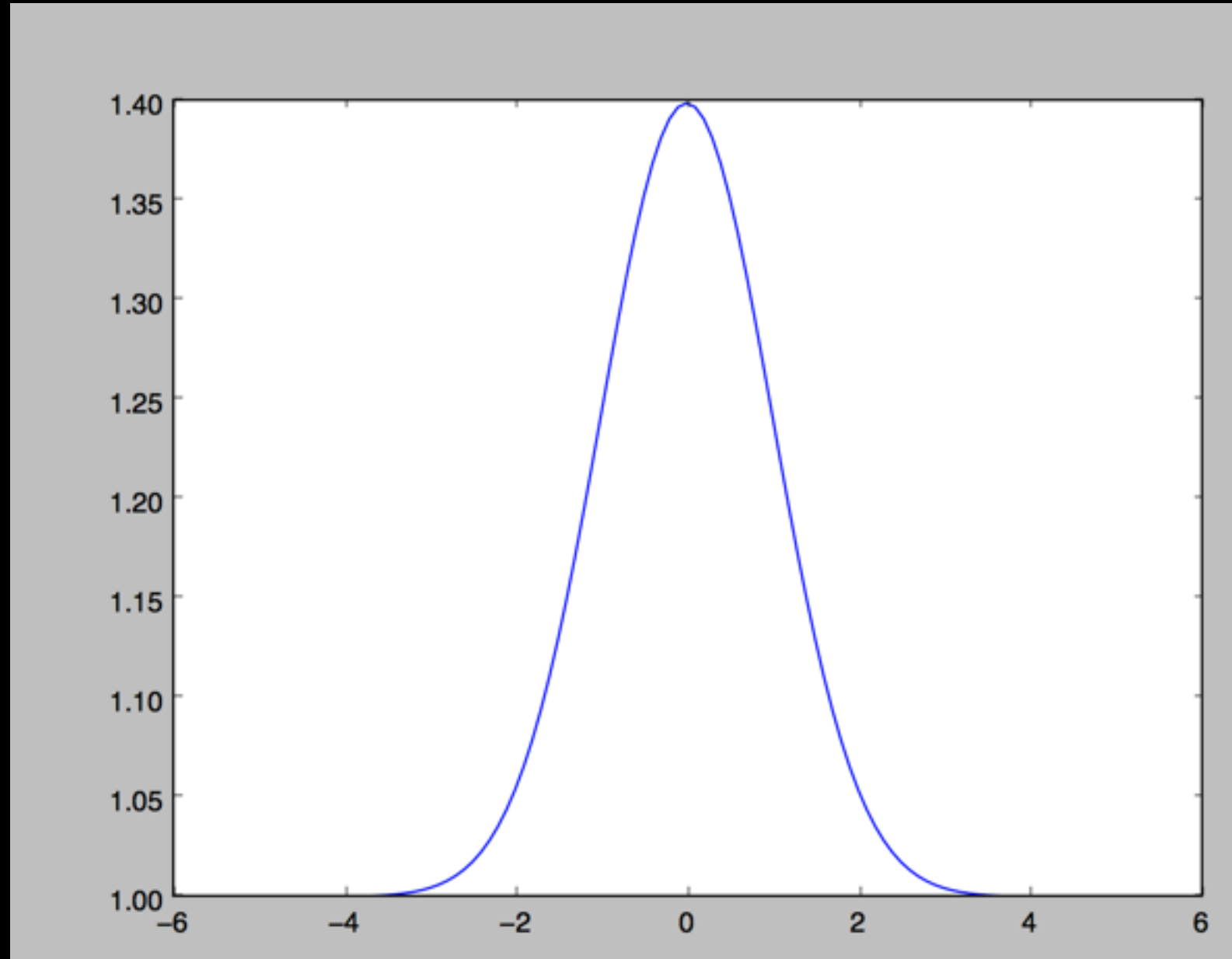
```
Jonathans-MacBook-Pro:lecture3 sievers$ python matplotlib_example.py
```

Functions in Python

```
import numpy
from matplotlib import pyplot as plt
#functions defined with def
#arguments can be given default values
def mygauss(x,cent=0,sig=0.1):
    y=numpy.exp(-0.5*(x-cent)**2/sig**2)
    #pick this normalization so area under
    #gaussian is one.
    y=1+y/numpy.sqrt(2*numpy.pi*sig**2)
    return y

#only run this part if script is executed
if __name__ == "__main__":
    dx=0.1
    x=numpy.arange(-5,5,dx)
    y=mygauss(x,0,1)
    y2=mygauss(x,sig=1)
    print 'y total is ' +repr(y.sum()*dx)

    plt.plot(x,y)
    plt.show()
```

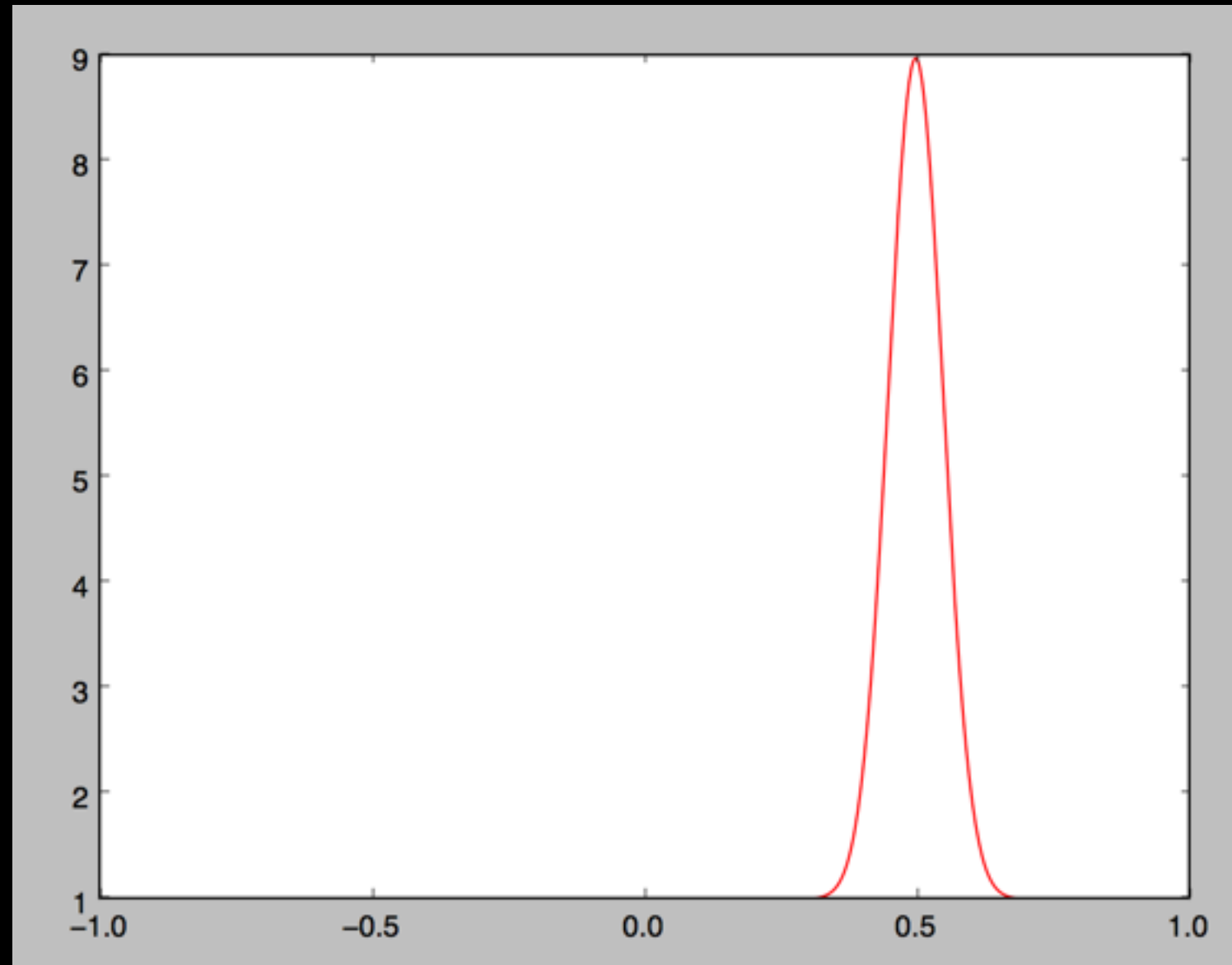


numpy arrays know how to do a lot of things. Can sum all elements with e.g. `y.sum()`. arrays also know their dimensions (`shape`), `min`, `max`, etc.

Functions Ctd.

```
import numpy
from matplotlib import pyplot as plt
#we can import functions we just wrote!
import func_example

x=numpy.arange(-1,1,0.002)
#functions are referenced by the file they're in
y=func_example.mygauss(x,cent=0.5,sig=0.05)
#we can assign the function to a variable
gg=func_example.mygauss
y2=gg(x,cent=0.5,sig=0.05)
delt=numpy.abs(y2-y)
print 'error is ' + repr(delt.sum())
#will output:
#error is 0.0
plt.plot(x,y,'r')
plt.show()
```



Integration

- We know have the tools to do some simple definite integrals
- Recall fundamental definition - $\sum (f(x_i) * dx)$ as $dx \rightarrow 0$
- How would we approximate integral of sin from 0 to pi?
- Before we do that, what **should** the answer be?

Example

- Here's some code that does the numerical integral of sin while varying the step size. How well does it work?
- you might also want to play with `numpy.linspace...`

```
import numpy

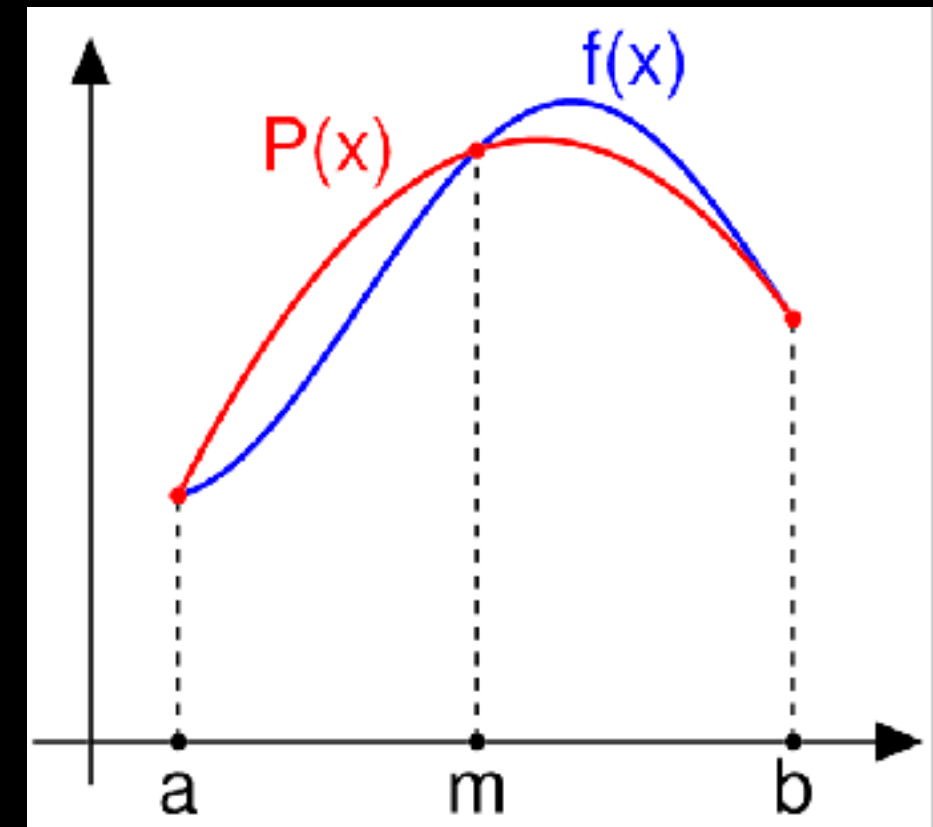
x0=0
x1=numpy.pi

mydelts=[0.5,0.1,0.03,0.01,0.003,0.001]
for dx in mydelts:
    x=numpy.arange(x0,x1,dx)
    y=numpy.sin(x)
    tot=y.sum()*dx
    print 'integral is ' + repr(tot) + ' with dx=' + repr(dx)
```

```
Jonathans-MacBook-Pro:lecture3 sievers$ python sin_integral.py
integral is 1.9836405445028298 with dx=0.5
integral is 1.999547959712598 with dx=0.1
integral is 1.9999407675824561 with dx=0.03
integral is 1.9999900283082466 with dx=0.01
integral is 1.9999992133611066 with dx=0.003
integral is 1.9999999540409921 with dx=0.001
Jonathans-MacBook-Pro:lecture3 sievers$
```

Simpson's Rule

- Let's integrate a quadratic over three points.
- Draw a straight line between the left and right points. The middle point is now off the straight line by $y_{\text{mid}} - 0.5 * (y_{\text{left}} + y_{\text{right}})$
- What is the average value of $(1-x^2)$ between -1 and 1?
- Area is now $1/2 * (y_{\text{left}} + y_{\text{right}}) + 2/3 * (y_{\text{mid}} - 0.5 * (y_{\text{left}} + y_{\text{right}}))$
- simplify: $\text{area} = 1/6 * y_{\text{left}} + 2/3 * y_{\text{mid}} + 1/6 * y_{\text{right}}$.
- for a bunch of points, string together segments, y_{right} become y_{left} of the next segment.
- Simpson's rule: $\text{integral} = dx * (1/6 y_0 + 2/3 y_{\text{odd}} + 1/3 y_{\text{even}} + 1/6 y_{\text{last}})$



4th Order Runge-Kutte

- Sometimes we want to integrate ODE's, $dy/dx=f(x,y)$
- Tricker than simple integration, because we can't evaluate at arbitrary points since y is changing.
- One standard technique is 4th-order Runge-Kutta, analagous to Simpson's rule. Make estimates of function using left edge, 2 in center, and one at right.
- RK4 reduces to Simpson's rule if $dy/dx=f(x)$ only (not $f(x,y)$). For smooth functions, RK4 should be accurate to 4th order.

RK4 Recipe

- for step size h , let $k_1 = h * f(x, y)$ (left edge)
- $k_2 = h * f(x + h/2, y + k_1/2)$ (first mid-point estimate)
- $k_3 = h * f(x + h/2, y + k_2/2)$ (second mid-point estimate)
- $k_4 = h * f(x + h, y + k_3)$ (right edge)
- $y(x+h) = y(x) + (k_1 + 2k_2 + 2k_3 + k_4)/6$

In Practice

```
import numpy
def myfun(x,y,a):
    #evaluate dydx=a*x*y
    return x*y*a
def rkstep(x,y,h,a,func):
    k1=h*func(x,y,a);
    k2=h*func(x+0.5*h,y+0.5*k1,a);
    k3=h*func(x+0.5*h,y+0.5*k2,a);
    k4=h*func(x+h,y+k3,a);
    dy=(k1+2*k2+2*k3+k4)/6
    return dy
y0=2.0
x0=4.0
x1=10
a=0.5
h=0.01
y=y0
for x in numpy.arange(x0,x1,h):
    #print "(x,y)="+repr(x)+' '+repr(y)
    y=y+rkstep(x,y,h,a,myfun)
print "at end (x,y)="+ repr(x+h) + ' ' + repr(y)
#can solve analytically:
#dy/y=axdx, log(y)=0.5*ax^2+c, y=c*exp(0.5*ax^2)
#at (x0,y0)=c=y0/exp(0.5*ax0**2)
c=y0/numpy.exp(0.5*a*x0**2)
print "predicted: " + repr(c*numpy.exp(0.5*a*(x+h)**2))
```

```
Jonathans-MacBook-Pro-3:lecture3_2015 sievers$ python rk4.py
at end (x,y)=9.999999999999998721 2637630368.8623924
predicted: 2637631468.9647431
```

Fourier Transforms

- Functions can be represented in many different ways
- We normally use “real” space - $f(x)$
- Generally, arbitrarily many transforms exist to represent functions in different spaces - $F(y)=Af(x)$ for some matrix A and some new variable y .
Iff A is invertible, $f(x)=A^{-1}F(y)$
- One important basis nature has picked out is complex exponentials/sines and cosines. Fundamental across physics, particularly quantum mechanics.

Fundamental Definition

- $F(k) = \int f(x) \exp(-2\pi i k x) dx$ (where $k = 1/\omega$)
- Integral gets rid of x , replaces with k . New function has amplitude and phase as a function of k .
- Quantum mechanics - de Broglie says $p = \hbar k$. So, Fourier transform position to get momentum.
- Fourier transform electric field $E(t)$ to get frequency spectrum.
- Fourier transform to get fast correlations, convolutions, many other things.

DFT (Discrete FT)

- Computers don't do continuous. Not enough RAM for starters...
- Function exists over finite range in x at finite number of points.
- If input function has n points, output can only have n k 's.
- Gives rise to discrete Fourier Transform (DFT)
- $F(k) = \sum f(x) \exp(-2\pi i k x / N)$ for N points and $0 \leq k < N$
- What would DFT of $f(0)=1$, otherwise $f(x)=0$ look like?
- What would DFT of $f(x)=1$ look like?
- DFTs have subtle behaviours not seen in continuous, infinite FTs.

Inverse

- One way to think about DFT is as a matrix multiply.
- $F(k) = Af$, $A_{mn} = \exp(2\pi i mn/N)$
- But look - $A_{mn} = A_{nm}$, so matrix is symmetric.
- Also, columns are orthogonal under conjugation:
 $\sum \exp(-2\pi i kx) \exp(2\pi i k'x) = \sum \exp(2\pi i (k' - k)x)$. N if $k' = k$, otherwise 0 .
- So, $A^{-1} = 1/N \cdot \text{conj}(A)$. $\text{IFT} = 1/N \sum F(k) \exp(-2\pi i kx)$.
- Get back to where we started by just doing another DFT with a sign flip, then divide by # of data points.
- Alternative: divide by \sqrt{N} in both DFT and IFT, (not standard computationally)

Numpy Complex

```
import numpy
def exp_prod(m,n,N):
    #define imaginary unity
    J=numpy.complex(0,1)
    #now rest of code is just like for real numbers
    x=numpy.arange(0.0,N)*2*J*numpy.pi/N
    return numpy.sum(numpy.exp(-1*x*m)*numpy.exp(x*n))
if __name__=="__main__":
    print exp_prod(0,0,8)
    print exp_prod(2,4,8)
    print exp_prod(3,3,8)
    print exp_prod(0,7,8)
```

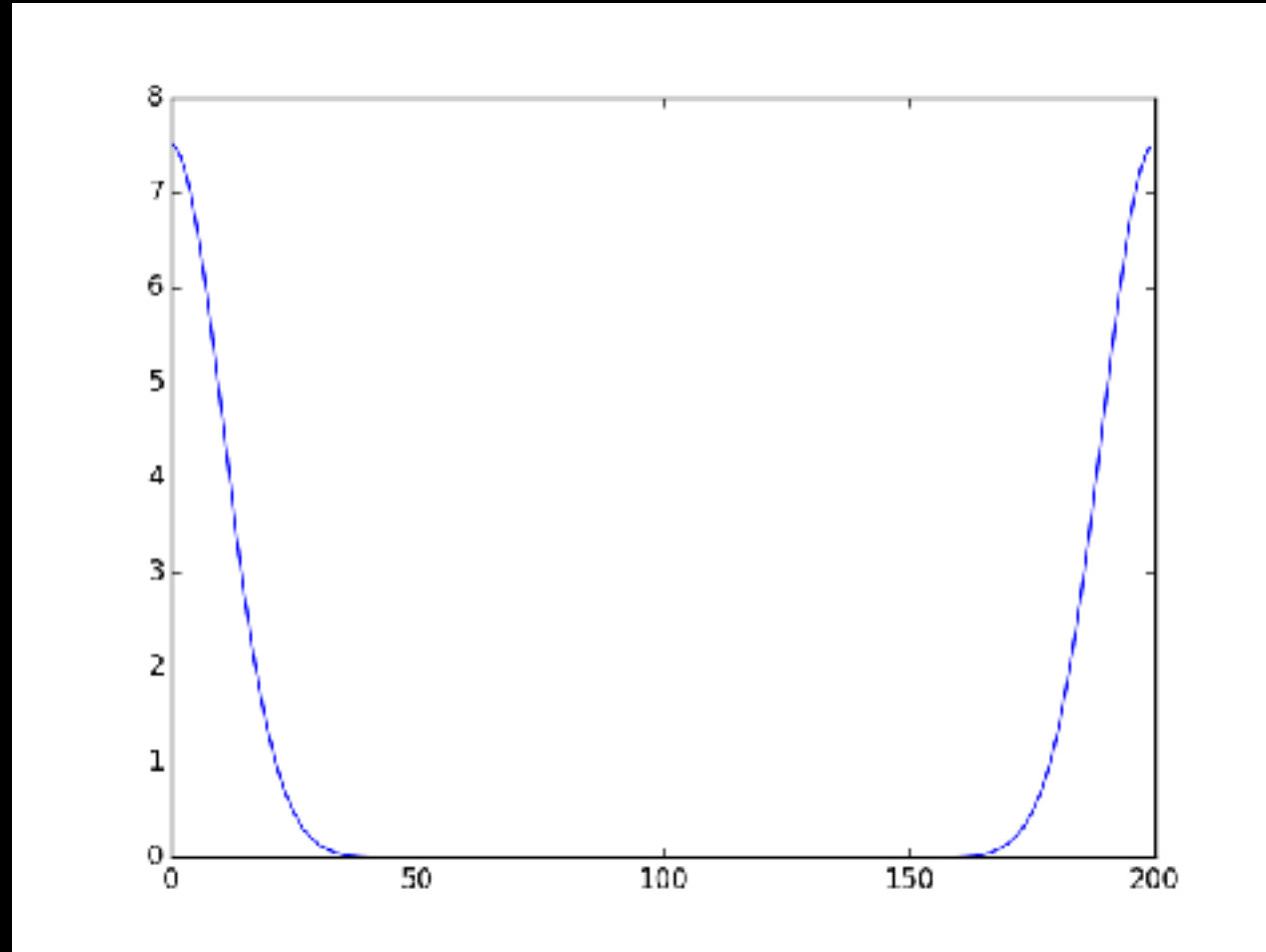
```
Jonathans-MacBook-Pro:lecture4 sievers$ python dft_columns.py
(8+0j)
(-4.28626379702e-16+4.4408920985e-16j)
(8+0j)
(3.44169137634e-15-1.11022302463e-15j)
Jonathans-MacBook-Pro:lecture4 sievers$
```

- Let's check orthogonality, need complex #'s.
- `numpy.complex(re,im)` will make a complex #
- numpy functions usually defined for complex #'s.

DFTs with Numpy

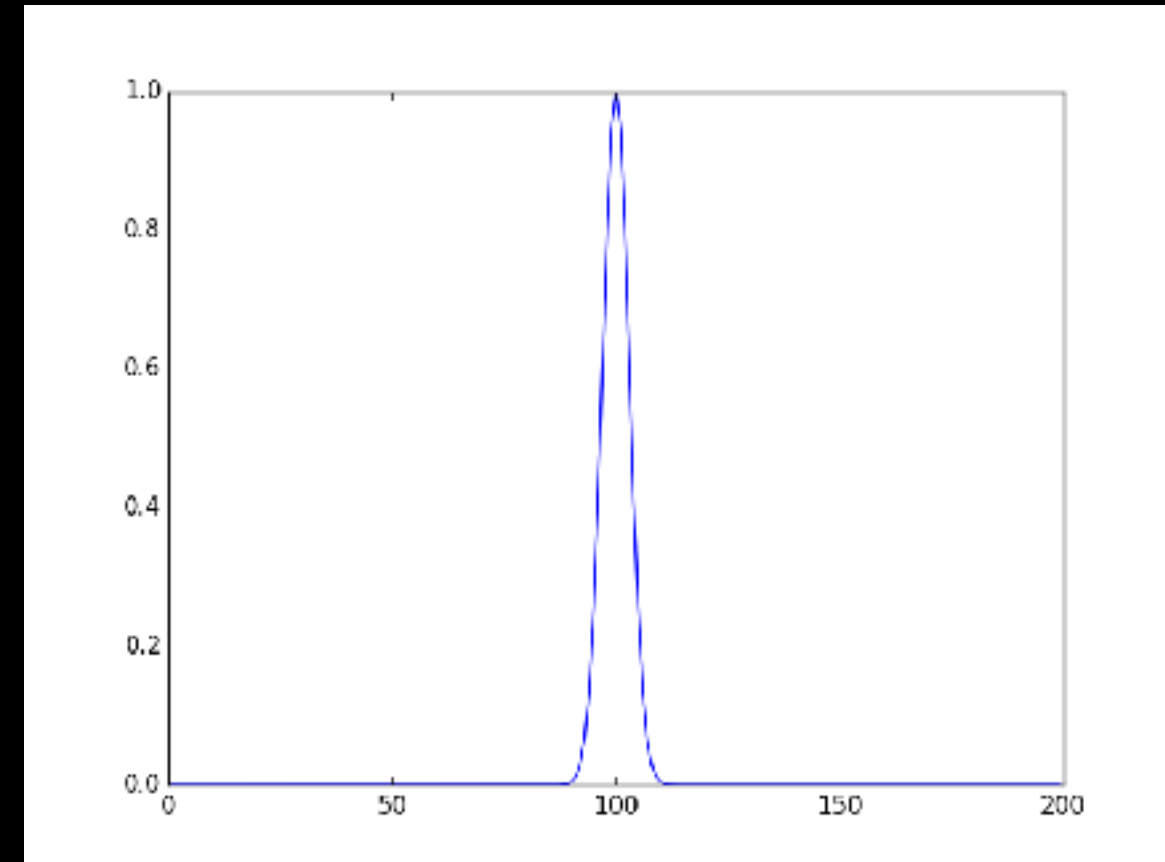
- Numpy has many Fourier Transform operations
- (for reasons to be seen) they are called *Fast* Fourier Transforms - FFT is one way of implementing DFTs.
- FFT's live in a submodule of numpy called FFT
- `xft=numpy.fft.fft(x)` takes DFT
- `x=numpy.fft.ifft(x)` takes inverse DFT
- Numpy normalizes such that $f == \text{fft}(\text{ifft}(f)) == \text{ifft}(\text{fft}(f))$

DFT in Action



```
import numpy
from matplotlib import pyplot as plt

x=numpy.arange(-10,10,0.1)
y=numpy.exp(-0.5*x**2/(0.3**2))
yft=numpy.fft.fft(y)
plt.plot(numpy.abs(yft))
plt.savefig('gauss_dft')
plt.show()
```



- Right: input Gaussian
- Top: DFT of the Gaussian

Periodicity

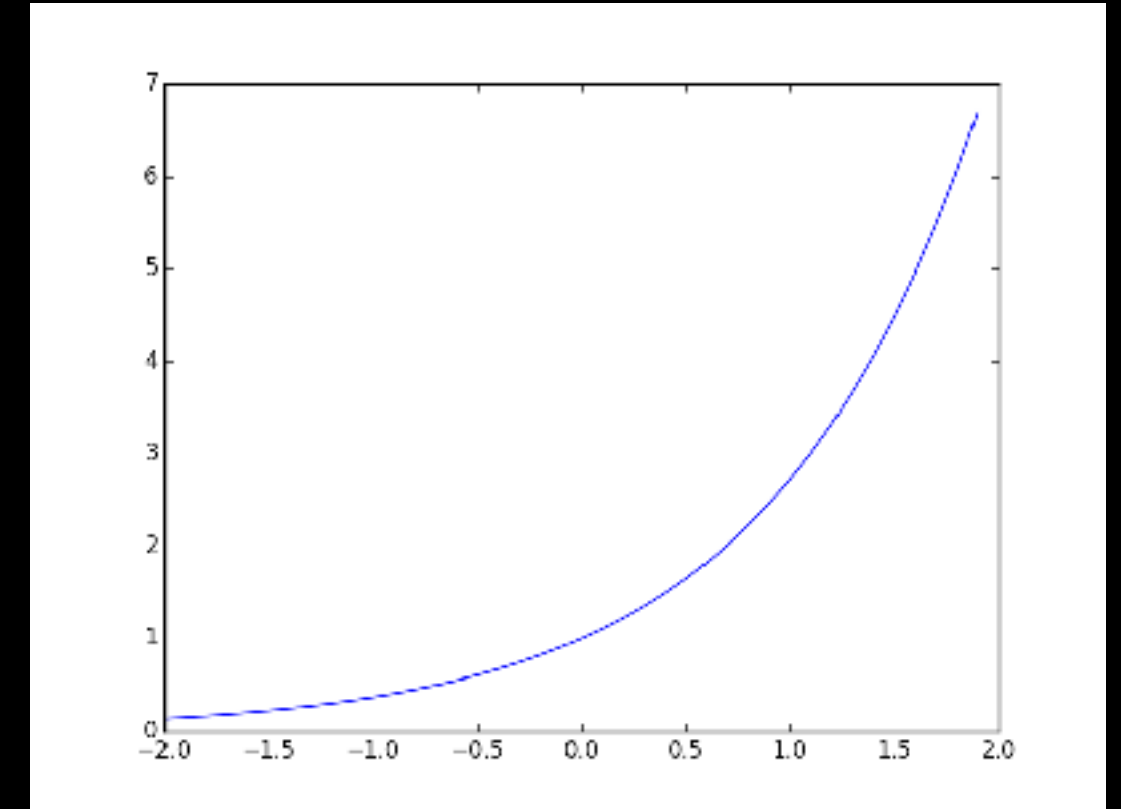
- $f(x) = \sum F(k) \exp(-2\pi i k x / N)$
- What is $f(x+N)$? $\sum F(k) \exp(-2\pi i k (x+N) / N)$
- $= \sum F(k) \exp(-2\pi i k) \exp(-2\pi i k x / N)$.
- $\exp(-2\pi i k) = 1$ for integer k , so $f(x+N) = f(x)$
- DFT's are periodic - they just repeat themselves ad infinitum.

Periodicity

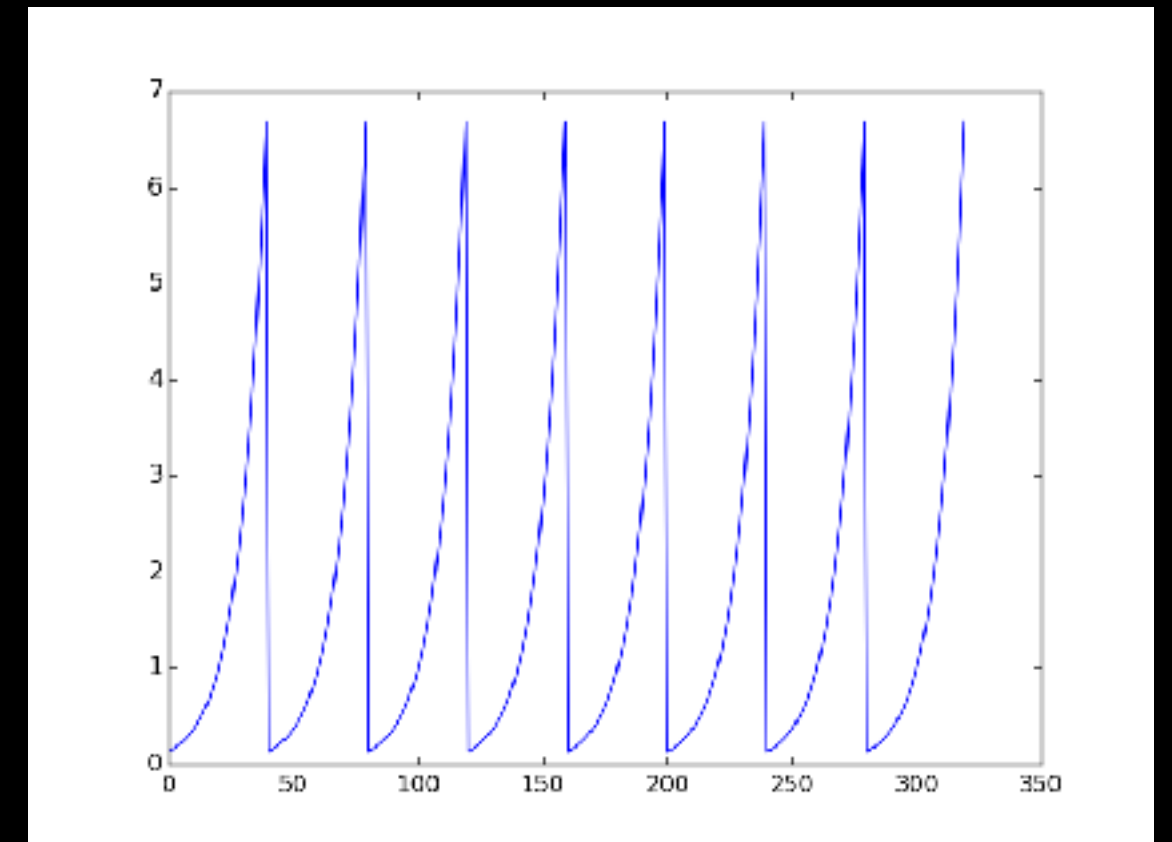
```
import numpy
from matplotlib import pyplot as plt

x=numpy.arange(-2,2,0.1)
y=numpy.exp(x)
plt.plot(x,y)
plt.savefig('fft_exp')
plt.show()

yy=numpy.concatenate((y,y))
yy=numpy.concatenate((yy,yy))
yy=numpy.concatenate((yy,yy))
plt.plot(yy)
plt.savefig('fft_exp_repeating')
plt.show()
```



- You may think you're taking top transform. You're not - you're taking the bottom one.
- In particular, jumps from right edge to left will strongly affect DFT



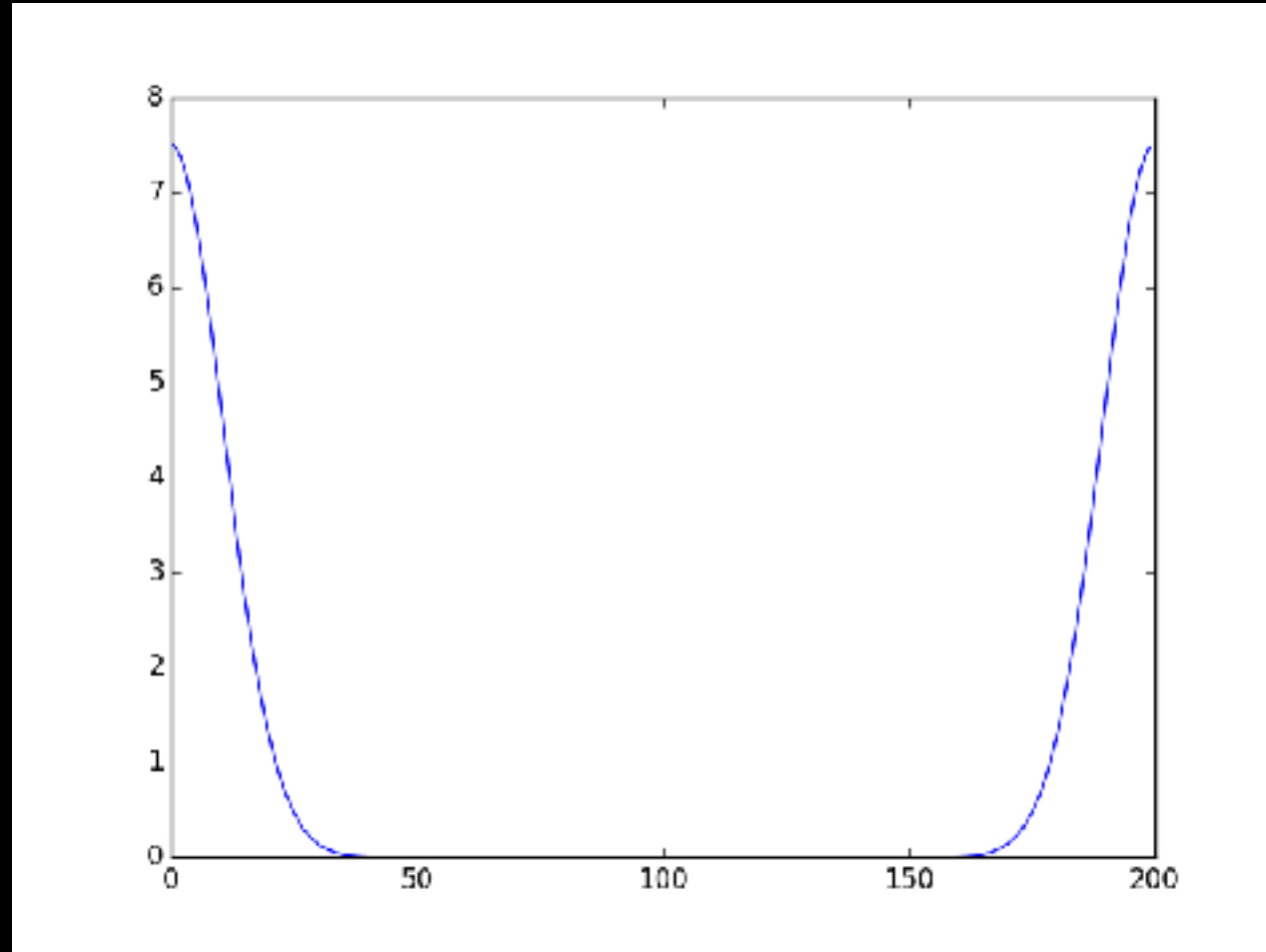
Aliasing

- $f(x) = \sum F(k) \exp(2\pi i k x / N)$
- What if I had higher frequency, $k > N$? let $k^* = k - N$ (i.e. k^* low freq.)
- $f(x) = \sum F(k) \exp(2\pi i (k^* + N)x / N) = \sum F(k) \exp(2\pi i x) \exp(2\pi i k^* x / N)$
- for x integer, middle term goes away: $\sum F(k^* + N) \exp(2\pi i k^* x / N)$
- High frequencies behave exactly like low frequencies - power has been *aliased* into main frequencies of DFT.
- Always keep this in mind! Make sure samples are fine enough to prevent aliasing.

Negative Frequencies

- All frequencies that are N apart behave identically
- DFT has frequencies up to $(N-1)$.
- Frequency $(N-1)$ equivalent to frequency (-1) . You will do better to think of DFT as giving frequencies $(-N/2, N/2)$ than frequencies $(0, N-1)$
- *Sampling theorem*: if function is band-limited - highest frequency is ν - then I get full information if I sample *twice* per frequency, $dt = 1/(2\nu)$. Factor of 2 comes from aliasing.

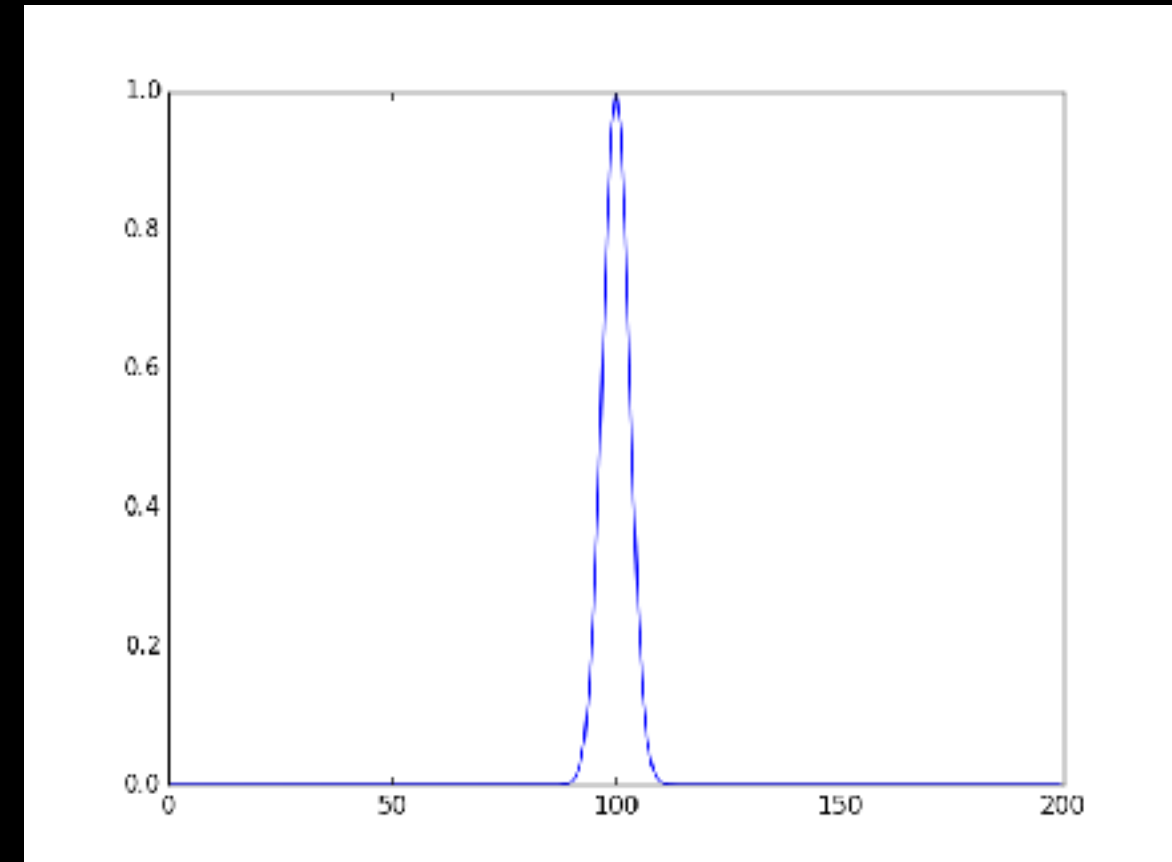
DFT in Action, Redux



```
import numpy
from matplotlib import pyplot as plt

x=numpy.arange(-10,10,0.1)
y=numpy.exp(-0.5*x**2/(0.3**2))
yft=numpy.fft.fft(y)
plt.plot(numpy.abs(yft))
plt.savefig('gauss_dft')
plt.show()
```

- FFT makes more sense now - negative frequencies have been aliased to high frequency.



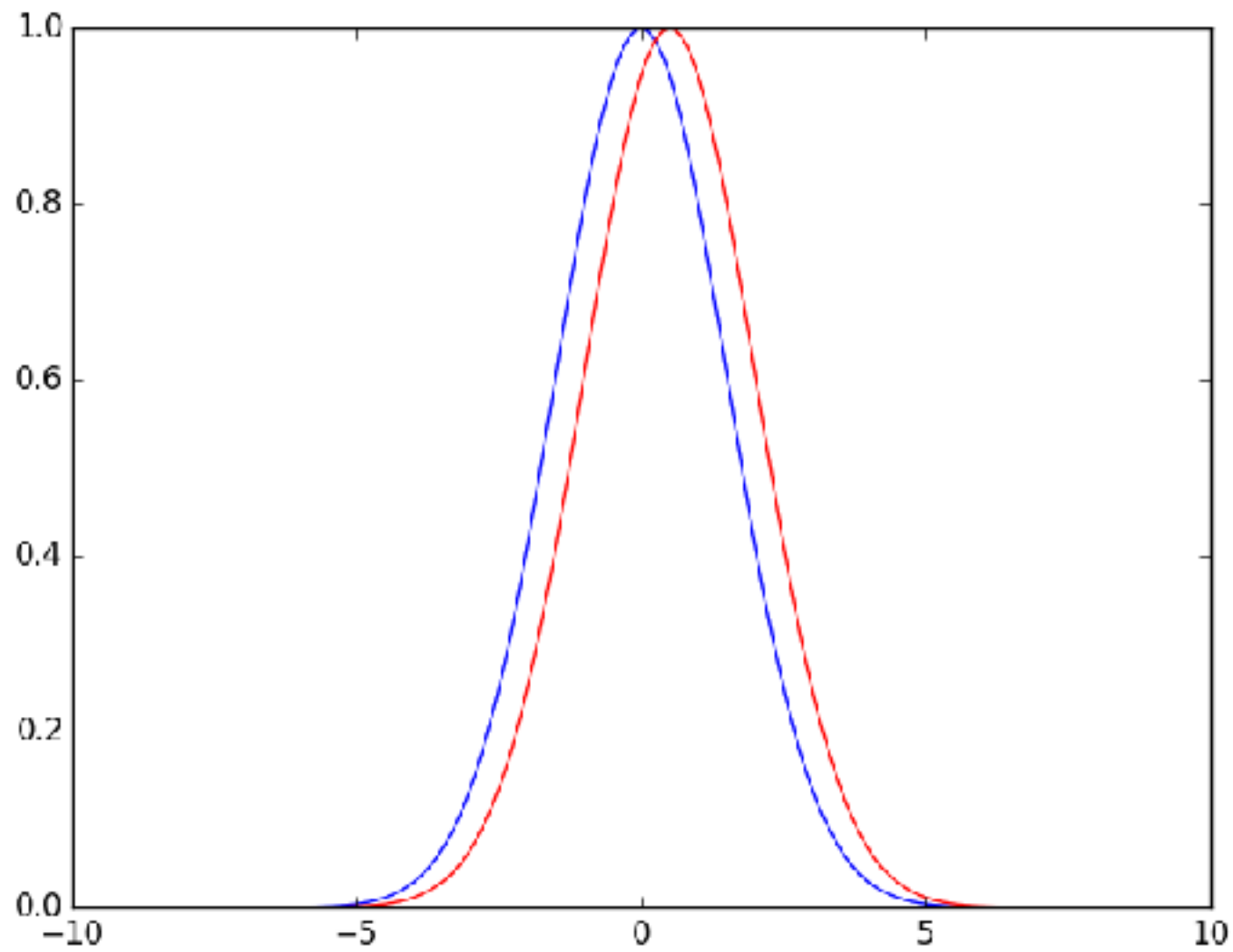
Flipping

- What is DFT of $f(-x)$?
- $\sum f(-x) \exp(-2\pi i k x / N)$, $x^* = -x$, $\sum f(x^*) \exp(-2\pi i k (-x) / N)$
- $\text{DFT}(f(-x)) = \sum f(x) \exp(2\pi i k x / N) = \text{conj}(F(k))$

Shifting

- What is $\text{FFT}(x+dx)$? $\sum f(x+dx)\exp(-2\pi i k x/N)$.
- $x^*=x+dx$: $F(k)=\sum f(x^*)\exp(-2\pi i k (x^*-dx)/N)$
- $F(k)=\exp(2\pi i k dx/N)\sum f(x^*)\exp(-2\pi i k x^*/N)$
- So, just apply a phase gradient to the DFT to shift in x

Shifting Example



```
import numpy
from matplotlib import pyplot as plt

x=numpy.arange(-10,10,0.1)
y=numpy.exp(-0.5*x**2/(1.5**2))
N=x.size
kvec=numpy.arange(N)
yft=numpy.fft.fft(y)
J=numpy.complex(0,1)
dx=5.0;
yft_new=yft*numpy.exp(-2*numpy.pi*J*kvec*dx/N)
y_new=numpy.real(numpy.fft.ifft(yft_new))
plt.plot(x,y)
plt.plot(x,y_new,'r')
plt.savefig('shifted_gaussian')
plt.show()
```

Real Data Symmetry

- If I know $F(k)$, what is $F(N-k)$ if $f(x)$ is real?
- $F(N-k)=F(-k)$ (from alias theorem)
- $F(-k)=\sum f(x)\exp(-2\pi i(-k)x/N)$. let $x^*=-x$
- $F(-k)=\sum f(-x^*)\exp(2\pi i k x^*/N) = \text{conj}(F(k))$ by flipping
- So, if $f(x)$ is real, $F(k)=\text{conj}(F(N-k))$
- If N even, $F(N/2)=\text{conj}(F(N/2))$, so $F(N/2)$ must be real.

```
>>> x=numpy.random.randn(8)
>>> xft=numpy.fft.fft(x)
>>> for xx in xft:
...     print xx
...
(-4.53568815727+0j)
(-0.174046761579+2.08827239558j)
(2.15348308858+2.32162497273j)
(-0.423040513854-3.72126858798j)
(2.75685372591+0j)
(-0.423040513853+3.72126858798j)
(2.15348308858-2.32162497273j)
(-0.174046761579-2.08827239558j)
>>>
```

Convolution Theorem

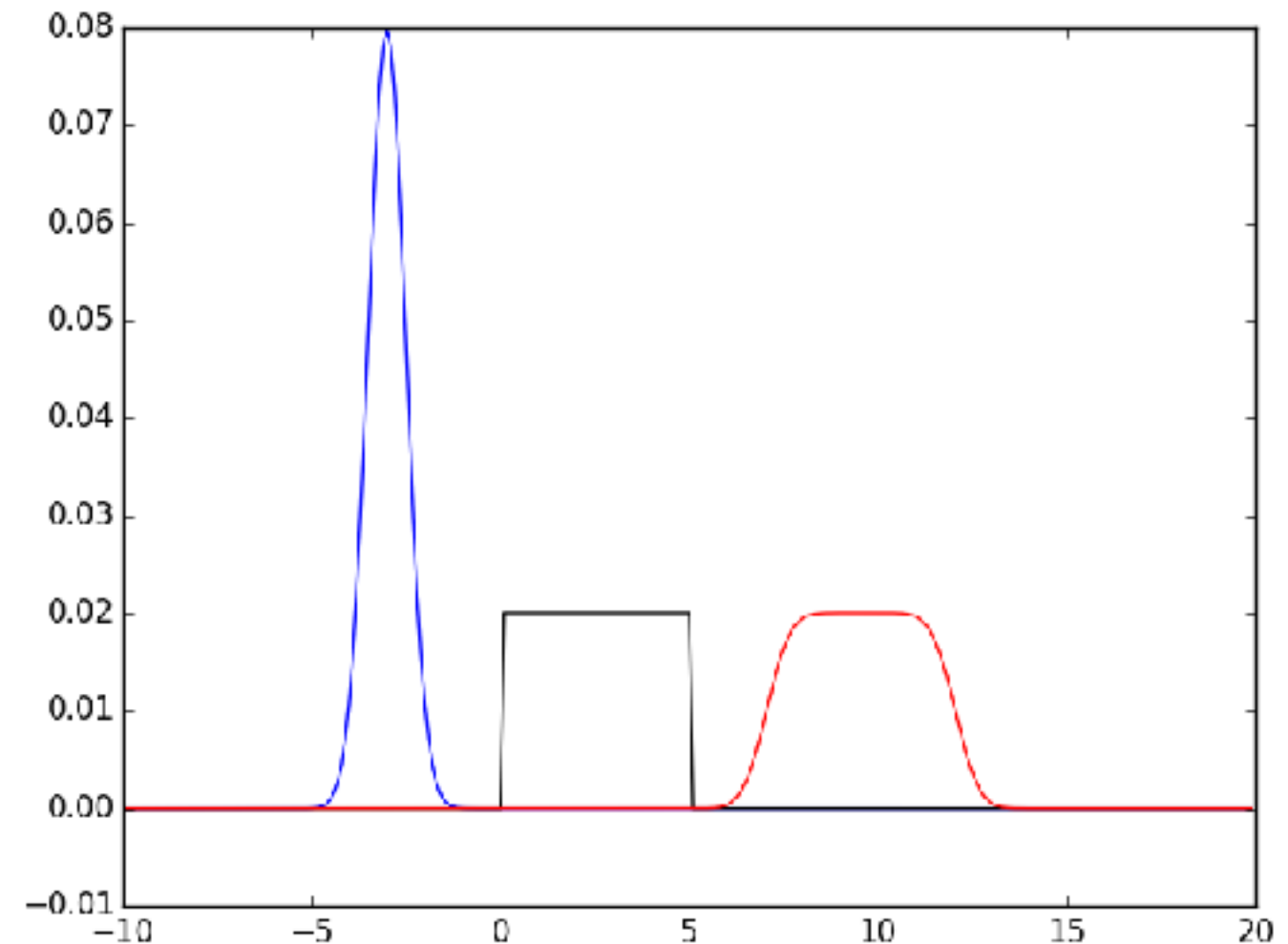
- Convolution defined to be $\text{conv}(y) = f \otimes g = \int f(x)g(y-x)dx$
- $\sum_x \sum F(k) \exp(2\pi i k x) \sum \text{conj}(G(k')) \exp(2\pi i k' x) \exp(-2\pi i k' y/N)$
- Reorder sum: $\sum \sum F(k) \text{conj}(G(k')) \exp(-2\pi i k' y/N) \sum_x \exp(2\pi i (k+k')x)$
- equals zero unless $k' = -k$. Cancels one sum, conjugates G
- $f \otimes g = \sum F(k)G(k) \exp(2\pi i k y/N) = \text{ift}(\text{dft}(f) * \text{dft}(g))$
- So, to convolve two functions, multiply their DFTs and take the IFT

Convolution Example

```
from numpy import arange,exp,real
from numpy.fft import fft,ifft
from matplotlib import pyplot as plt
def conv(f,g):
    ft1=fft(f)
    ft2=fft(g)
    return real(ifft(ft1*ft2))

x=arange(-10,20,0.1)
f=exp(-0.5*(x+3)**2/0.5**2)
g=0*x;g[(x>0)&(x<5)]=1
g=g/g.sum()
f=f/f.sum()
h=conv(f,g)

plt.plot(x,f,'b')
plt.plot(x,g,'k')
plt.plot(x,h,'r')
plt.savefig('convolved')
plt.show()
```



Fast Fourier Transform

- How many operations does a DFT take?
- Have an N by N matrix operating on a vector of length N - clearly N^2 operations, right?
- Nope! Otherwise we'd never use them. What's actually going on?
- Note $\text{DFT} = \sum f(x) \exp(-2\pi i k x / N) = \sum f_{\text{even}}(x) \exp(-2\pi i k (2x) / N) + \sum f_{\text{odd}}(x) \exp(-2\pi i k (2x+1) / N)$
- $= F_{\text{even}} + \exp(-2\pi i k / N) F_{\text{odd}}$. Let $W_k = \exp(-2\pi i k / N)$
- if $k > N/2$, then $k^* = k - N/2$ and $\text{DFT} = F_{\text{even}} + \exp(-2\pi i k^* / N + i\pi) F_{\text{odd}} = F_{\text{even}} - W_k F_{\text{odd}}$.

FFT cont'd

- So $F(k) = F_{\text{even}}(k) + W_k F_{\text{odd}}(k)$ ($k < N/2$) or $F_{\text{even}}(k) - W_k F_{\text{odd}}(k)$ ($k \geq N/2$)
- So, can get *all* the frequencies if I have 2 half-length FFTs.
- Well, just do the same thing again. FFT of a single element is itself.
- This algorithm works for arrays whose length is a power of 2
- Popularized by Cooley/Tukey in early computer days. Later found to go back to Gauss in 1805. Changes computational work from n^2 to $n \log n$.

Sample FFT

- Routine uses *recursion* - function calls itself. Recursion can be very powerful, but also easy to goof.
- `numpy.concatenate` will combine arrays - note that they have to be passed in as a tuple, hence the extra set of parenthesis
- Modern FFT routines deal with arbitrary length arrays. Fastest Fourier Transform in the West (FFTW) standard packaged - usually used by numpy.

```
from numpy import concatenate,exp,pi,arange,complex
def myfft(vec):
    n=vec.size
    #FFT of length 1 is itself, so quit
    if n==1:
        return vec
    #pull out even and odd parts of the data
    myeven=vec[0::2]
    myodd=vec[1::2]

    nn=n/2;
    j=complex(0,1)
    #get the phase factors
    twid=exp(-2*pi*j*arange(0,nn)/n)

    #get the dfts of the even and odd parts
    eft=myfft(myeven)
    oft=myfft(myodd)

    #Now that we have the partial dfts, combine them with
    #the phase factors to get the full DFT
    myans=concatenate((eft+twid*oft,eft-twid*oft))
    return myans
```

```
>>> import myft
>>> x=numpy.random.randn(32)
>>> xft1=numpy.fft.fft(x)
>>> xft2=myft.myfft(x)
>>> print numpy.sum(numpy.abs(xft1-xft2))
2.33937690259e-13
>>>
```

Tutorial (also do lecture 1 problems)

- Write a python script to make a vector of n evenly spaced numbers between 0 and $\pi/2$. i.e. $x[0]=0$, $x[-1]=\pi/2$ (5)
- Use this vector to integrate $\cos(x)$ from 0 to $\pi/2$ for a range # of points using the simple method. include 10,30,100,300,1000 points between 0 and $\pi/2$. How does error scale with # of points? (5)
- Python supports array slicing - $x[5:10:2]$ will take points 5,7,9 from x . $x[5::2]$ will take points 5,7,9... from x . How can I take all odd points from an array? How can I take all even points from an array, but skipping the first and last points? (5)
- Write a python function to integrate this vector using Simpson's rule. How does error scale with # of points? How many points did we need to use in part 2 to get same accuracy as 11 points with Simpson's rule? (10)
- Plot the errors as a function of # of points using Simpson's rule and standard sum. You will want to use a log scale here - look at logplot.py in the github distribution (5)

Bonus Points

- the `scipy` module has built in integration functions in `scipy.integrate`. The `quad` routine will do numerical integrals. `quad` will try to put its effort where the function changes quickly.
- Look at `scipy_quad_example.py`, which uses `scipy` to integrate our Gaussian function over two different ranges. The integrals should be (almost) identical - yet they are not. Can you figure out why? (5)
- Can you write another function that will always give the correct answer to this integral? (5) Hint - you may want to do two integrals instead of one.