python™

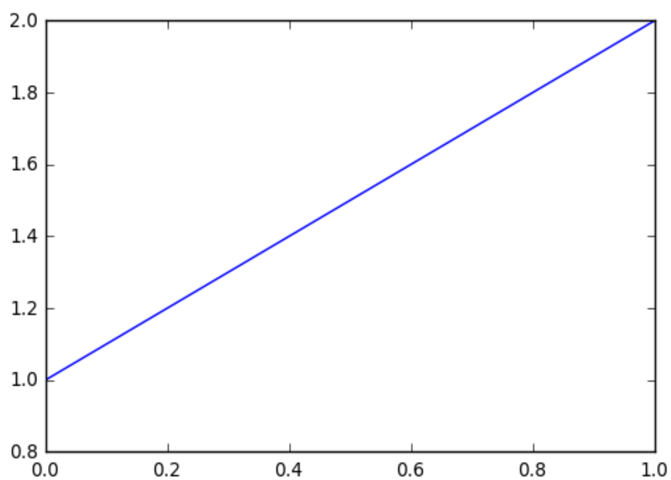
Continued

Jupyter

File Edit View Insert Cell Kernel Widgets Help

CellToolbar

```
In [1]: from pylab import *

In [2]: x=array([1.0,2.0])

In [3]: plot(x)
Out[3]: [<matplotlib.lines.Line2D at 0x7bf6be0>]

In [4]: show()
```




```
In [ ]: %pylab inline # If plot appears in a separate window

In [9]: x=randn(10)
        z=mean(x)
        print 'The mean is ', z # Use Ctrl + Enter

The mean is  -0.529317164575

In [ ]:
```

No new cell added to notebook

Cell can be edited and re-run

Other Shortcuts

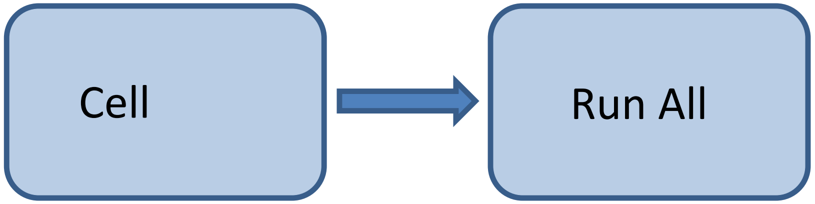
Ctrl + Shift + Enter

or

Alt + Enter

Executes the current cell and adds a new cell

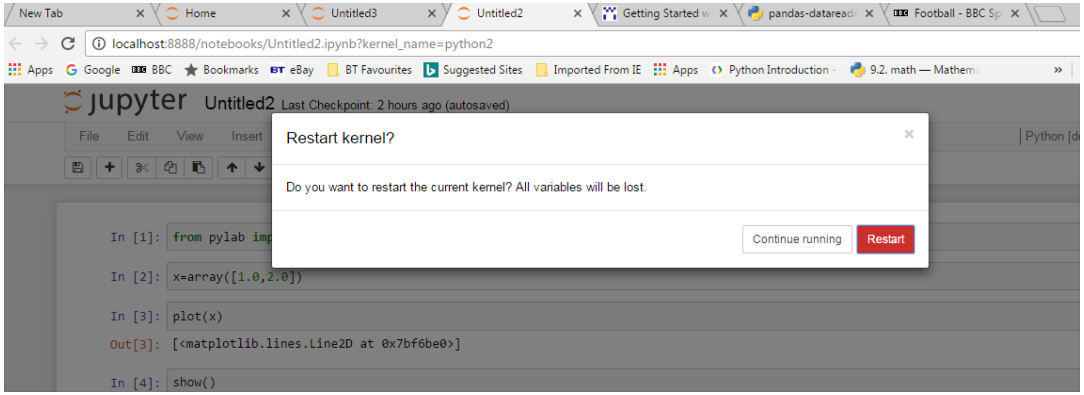
Other Shortcuts



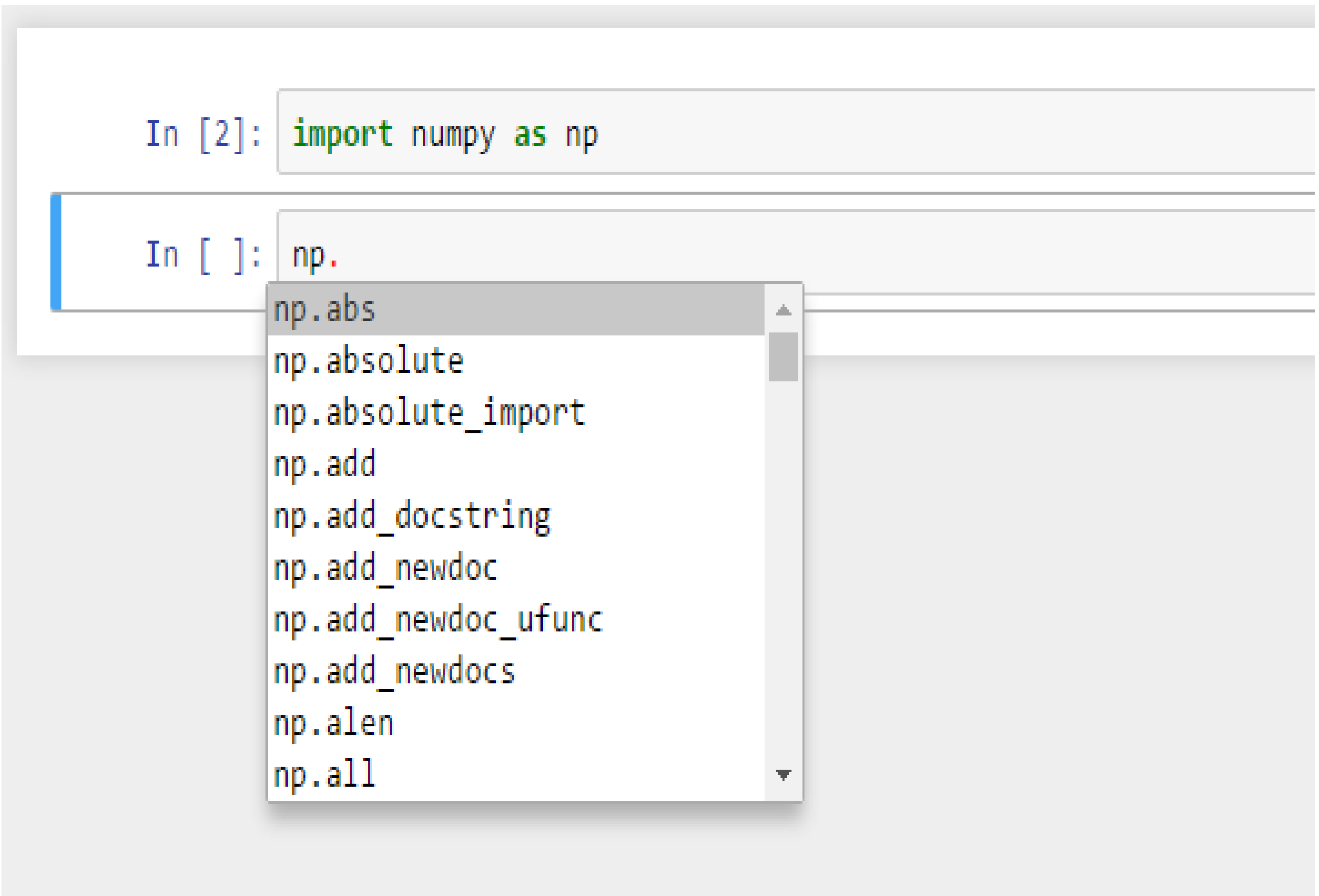
This evaluates all cells in the order they appear in the notebook. Note: cell numbers are incremented to reflect the re-evaluation

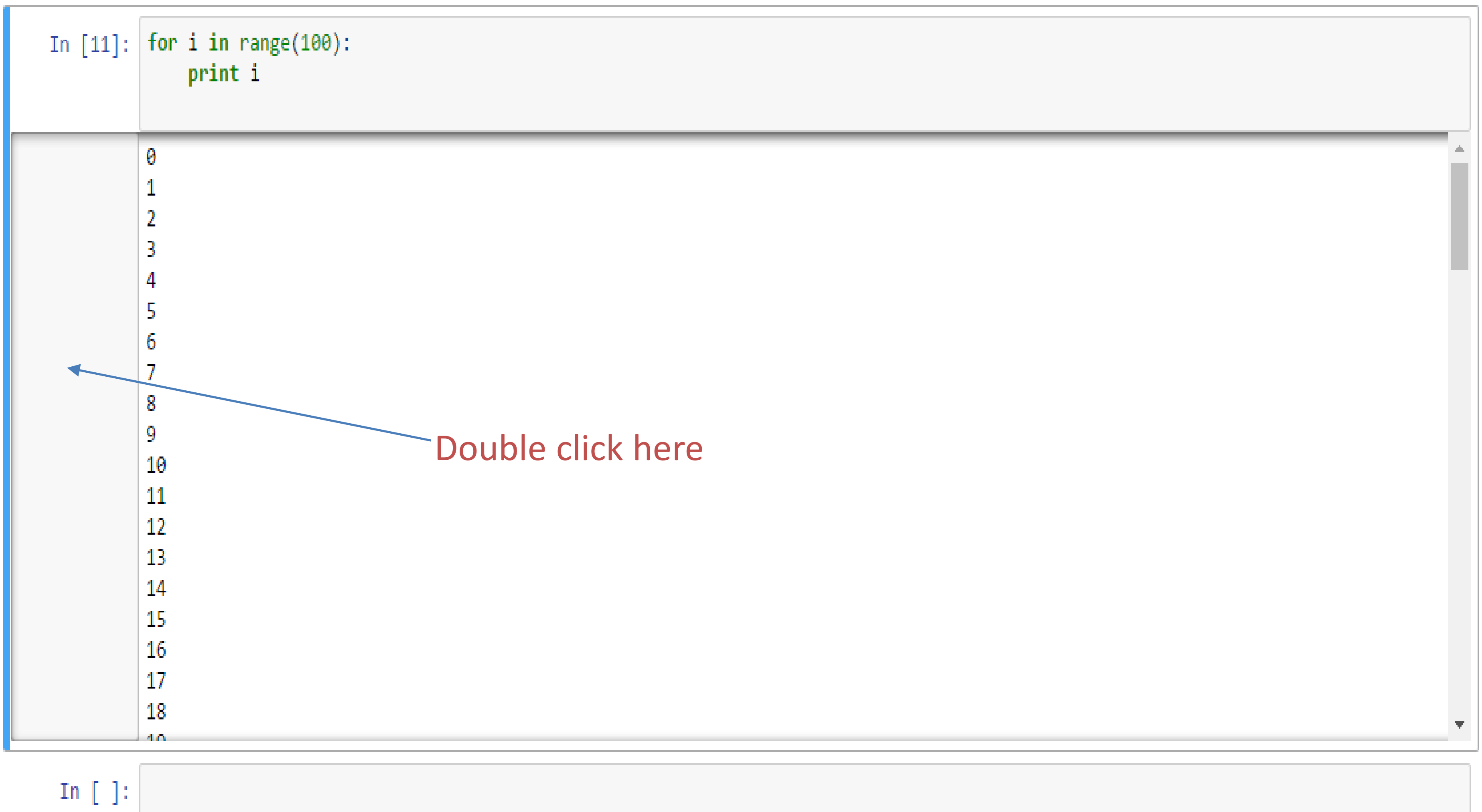
Kernel

IPython execution is done against a hidden kernel. This can be restarted to the same state at as the initial opening of the notebook. Restarting the kernel disconnects from the current kernel, stops the kernel and then connects to a new kernel. This is similar to using a ‘reset’ key. All numbering reset to 1



tab and autocomplete




In [11]:
for i in range(100):
    print i
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
Double click here
In [ ]:

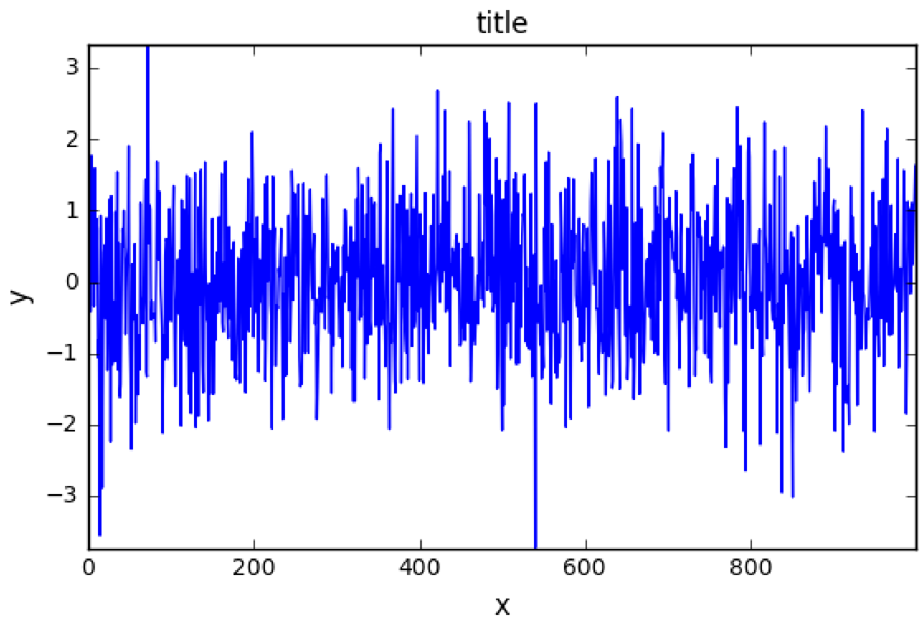
2D Plotting

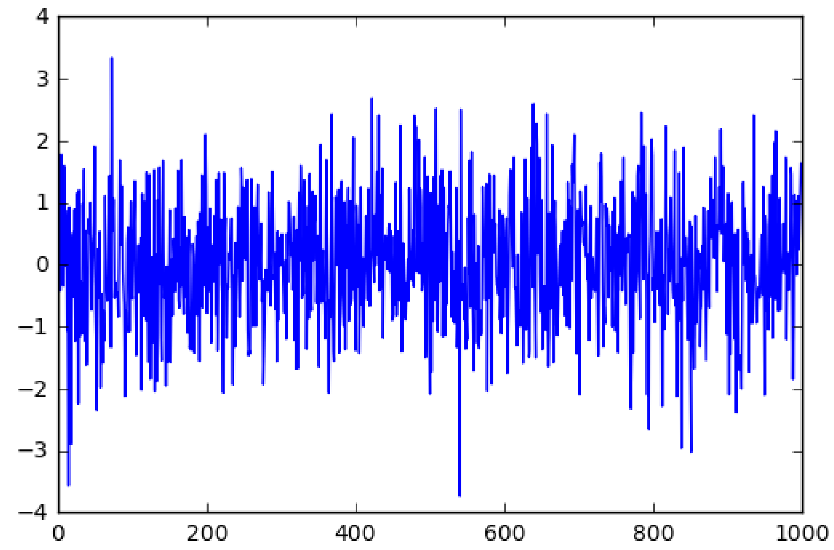
Two basic but very useful functions

- plt.autoscale : autoscale can be used to set limits within a figure's axes.
- plt.tight_layout : tight_layout will remove wasted space around a figure.

The use of which greatly improve the appearance of figures.

With and Without



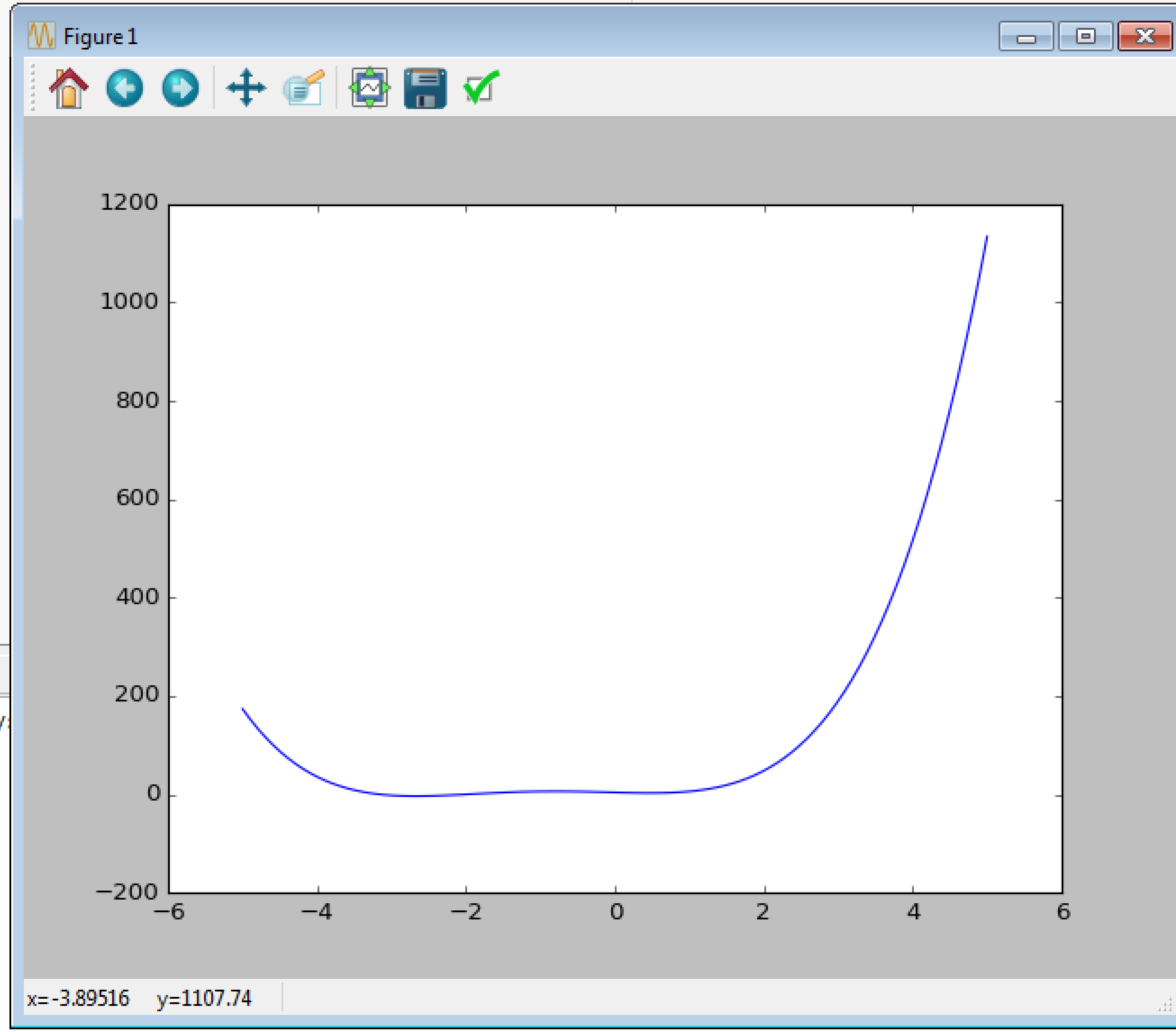



```
y=randn(1000)
plot(y)
ylabel('y', fontsize=12)
xlabel('x', fontsize=12)
title('title')
plt.autoscale(tight='x')
plt.tight_layout()
show()
```

Controlling Graphics

Colour		Marker		Line Style	
Black	**k**	Point	**.**	Solid	**-**
Blue	**b**	Pixel	**,**	Dashed	**--**
Cyan	**c**	Circle	**o**	Dash-dot	**-.**
Green	**g**	Square	**S**	Dotted	**:**
Magenta	**m**	Diamond	**D**		
Red	**r**	Thin Diamond	**d**		
White	**w**	Cross	**x**		
Yellow	**y**	Plus	**+**		
		Star	*****		
		Hexagon	**H**		
		Pentagon	**p**		
		Triangles	**^ v < >**		
		Horizontal Line	**_**		

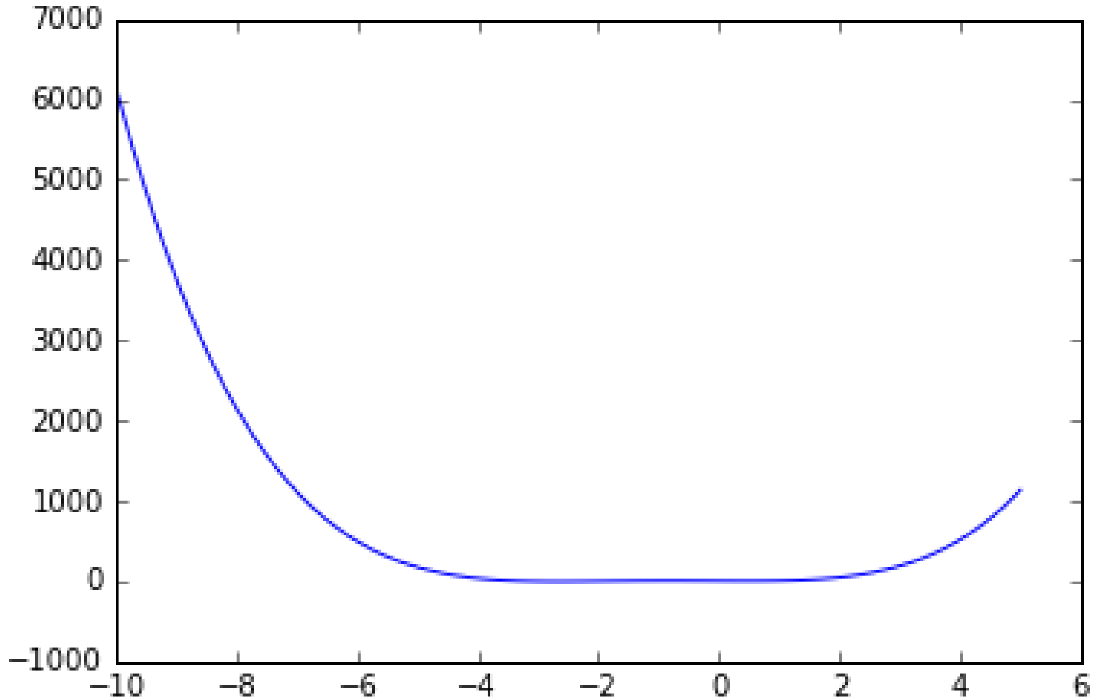
Further Plotting



Python

```
Welcome to Canopy's interactive data-analy
 with pylab-backend set to: qt
Type '?' for more information.

In [1]: from matplotlib import *

In [2]: from pylab import *

In [3]: x = linspace(-5, 5, 100)

In [4]: y=4*x**3 + (x-2)**2 + x**4

In [5]: figure()
Out[5]: <matplotlib.figure.Figure at 0xa5f7da0>

In [6]: plot(x, y, 'b')
Out[6]: [<matplotlib.lines.Line2D at 0xa92d2e8>]
```

```
In [13]: %matplotlib inline

In [14]: plot(x, y, 'b')
Out[14]: [<matplotlib.lines.Line2D at 0xabd3518>]
```



Dates and Times

Date and time manipulation is provided by a built-in Python module **datetime**.

```
In [1]: import datetime as dt

In [2]: yr, mo, dd = 2016, 11, 2

In [3]: dt.date(yr, mo, dd)
Out[3]: datetime.date(2016, 11, 2)

In [4]: hr, mm, ss, ms= 9, 21, 12, 21

In [5]: dt.time(9, 21, 12, 21)
Out[5]: datetime.time(9, 21, 12, 21)
```

Dates created using date do not allow times.

Dates which require a time stamp can be created using datetime, which combine the inputs from date and time, in the same order.

```
In [6]: dt.datetime(yr, mo, dd,hr, mm, ss, ms)
Out[6]: datetime.datetime(2016, 11, 2, 9, 21, 12, 21)
```

yr mo day hr, mn, sec, ms

Manipulating Dates

Date-times and dates (but not times, and only within the same type) can be subtracted to produce a timedelta, which consists of three values, days; seconds; microseconds. Time deltas can also be added to dates and times compute different dates – although date types will ignore any information in the time delta hour or millisecond fields.

```
import datetime as dt
yr, mo, dd = 2016, 11, 2
hr, mm, ss, ms= 9, 21, 12, 21
dt.date(yr, mo, dd)
d1 = dt.datetime(yr, mo, dd, hr, mm, ss, ms)
d2 = dt.datetime(yr+1, mo, dd, hr+2, mm, ss, ms)
d2-d1
```

days

```
Out[24]:
```

secs

```
datetime.timedelta(365, 7200)
```

How long does your code take?

Importing the module timeit gives the length of execution time (secs) required to run a specific piece of code

```
In [34]: import timeit
         def fun1(x, y):
             return x **2 + y**3
         t_start = timeit.default_timer()
         z = fun1 (109.2,367.1)
         t_end = timeit.default_timer ()
         cost = t_end - t_start
         print ('Time cost of this function is %f' % cost)

         Time cost of this function is 0.000715
```

Special Functions

We now turn our attention to special functions that can be manipulated in Python

Error Function

$$\frac{dy}{dx} - 2xy = 2$$

$$y(0) = 1$$

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-s^2} ds$$

Complimentary Error Function

$$erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-s^2} ds$$

$$\frac{2}{\sqrt{\pi}} \underbrace{\int_0^\infty e^{-s^2} ds}_{\frac{1}{2}\sqrt{\pi}}$$

$$\frac{2}{\sqrt{\pi}} \left[\int_0^x + \int_x^\infty \right]$$

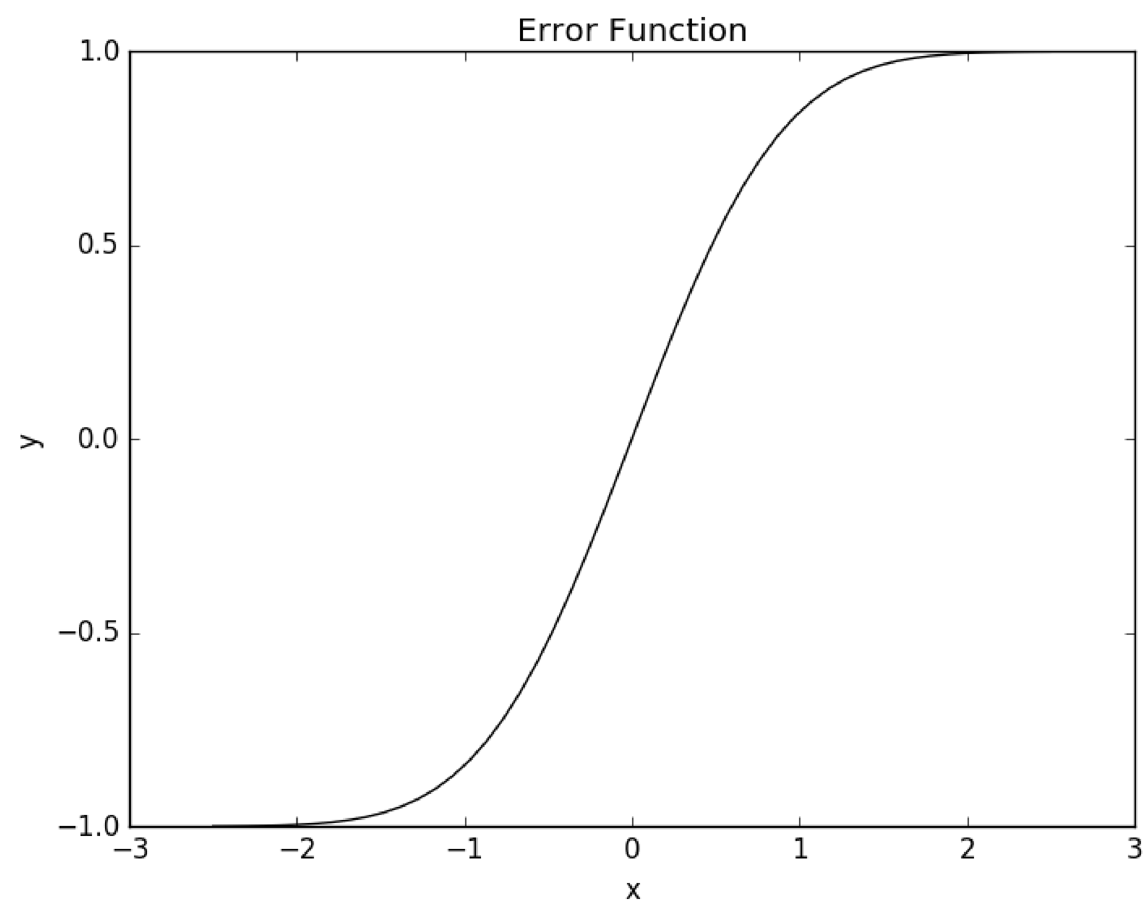
$$erf(x) + erfc(x) = 1$$

```
In [1]: from math import *

In [2]: from pylab import *

In [3]: from scipy import *

In [4]: erf(1.3)
Out[4]: 0.9340079449406524

In [5]: erfc(1.3) # 1-erf(1.3)
Out[5]: 0.06599205505934768
```

print erf(1.3) + erfc(1.3)

```
In [1]: import math

In [2]: import pylab

In [3]: import scipy.special

In [4]: x=linspace(-2.5,2.5,50)

In [5]: y=scipy.special.erf(x)

In [6]: plot(x,y,'black')
Out[6]: [<matplotlib.lines.Line2D at 0xabc4400>]

In [7]: xlabel('x')
Out[7]: <matplotlib.text.Text at 0xaab7668>

In [8]: ylabel('y')
Out[8]: <matplotlib.text.Text at 0xaacb8d0>

In [9]: title('Error Function')
Out[9]: <matplotlib.text.Text at 0xab81898>
```

f



CDF

for $N(0,1)$

```
In [4]: from scipy import stats
        y = stats.norm.cdf(1.2)
        print(y)

0.884930329778
```

x = linspace[-2.5, 2.5, 100]

Exercises

3 $+0.138389$

4 $+1.633051$

Calculate the following:

1. $erf(2.3)$; $erf(1.0)$
2. Calculate $erfc(2.3)$; $erfc(1.0)$
3. $\int_{1}^{2.3} e^{-s^2} ds$
4. $\int_{-1}^{4} e^{-s^2} ds$

$$\int_a^b e^{-x^2} dx$$

Exercises

$$\boxed{2\,\mathbb{P}[X<a]-1} \quad a \to 1.2$$

If $X \sim N(0,1)$; calculate the following probabilities using the CDF

0.884 $\quad$ 0.115 $\quad$ 0.769

1. $\mathbf{P}(X < 1.2)$; $\quad$ $\mathbf{P}(X > 1.2)$; $\quad$ $\mathbf{P}(|X|<1.2)$
2. $\mathbf{P}(-2 < X < 1.5)$ 0.91

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}s^2}\,ds = \mathbb{P}[X<x]$$

$\hookrightarrow$ CDF

The ndarray type

NumPy provides a special data type: **ndarray** (n-dimensional array).

- Unlike tuples and lists, arrays can only store objects of the same type
- Makes operations on arrays much faster than on lists; in addition, arrays take less memory than lists.
- Arrays provide powerful extensions to the list indexing mechanism.

Universal Function

- A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features.
- In Numpy, universal functions are instances of the numpy.ufunc class. Many of the built-in functions are implemented in compiled C code, but ufunc instances can also be produced using the frompyfunc factory function.

Producing $N(0,1)$ from $U(0,1)$

Use the relationship between the error function and the Normal CDF

$$N(x) = \frac{1}{2}\left(1 + erf\left(\frac{x}{\sqrt{2}}\right)\right)$$

M.C

CDF

D

numpy contains functions with better efficiency

- A lot of commonly used mathematical functions are included in numpy, e.g:

  np.log np.maximum np.sin np.exp np.abs

- In most cases, numpy functions are more efficient than the similar functions in the math library, especially for large scale data.

Special Arrays

$1 + 0i$ $\sqrt{-1} = j$

There are a number of ways to define and populate common arrays; some that promote efficiency. Python has functions to achieve this.

ones

$[1, 1, \ldots, 1]$

generates an array of 1s and is generally called with one argument, a tuple, containing the size of each dimension.

ones takes an optional second argument (dtype) to specify the data type. If omitted, the data type is float.

```
In [11]: x=ones(10)

In [12]: x
Out[12]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

rows

row

```
In [18]: y=ones((2,3)) #initialising a 2x3 array with ones

In [19]: y
Out[19]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

column

```
In [28]: M=4; N=3

In [29]: A=ones((M,N)) # filling MxN array with ones
```

rows

```
In [30]: A
Out[30]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

4 rows

3 columns

float

dtype = complex

```
In [31]: x=ones(10,complex) #define array with ones of type complex

In [32]: x
Out[32]:
array([ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,
        1.+0.j,  1.+0.j,  1.+0.j])

In [38]: x = ones((M,N), dtype='int64') #64 bit integers

In [39]: x
Out[39]:
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]], dtype=int64)
```

zeroes

generates an array of 0s in the same way as ones. Hence another way to initialise arrays. As before if data type omitted, then float by default

Examples of zeroes

```
In [43]: M=N=3

In [44]: x = zeros((M,N)) # M by N array of 0s

In [45]: y = zeros((M,M,N)) # 3D array of 0s

In [46]: z = zeros((M,N),dtype= 'int32') # 32 bit integers

In [47]: x
Out[47]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

In [48]: y
Out[48]:
array([[[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]],

       [[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]],

       [[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]]])

In [49]: z
Out[49]:
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

empty

produces an empty (uninitialized) array to hold generated by another procedure. empty takes an optional second argument (dtype) which specifies the data type. If omitted, the data type is float.

```
In [52]: x = empty((M,N)) # M by N empty array

In [53]: y = empty((N,N,N,N)) # 4D empty array

In [54]: z = empty((M,N),dtype='float32') # 32 bit floats (single precision)
```

x = empty((3,4))

x

eye, identity

Essentially two similar functions, eye generates the equivalent of the identity matrix. Ones down leading diagonal, zeroes everywhere else.

n(n))   N x N

```
In [59]: IN
Out[59]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

In [60]: IN=identity(N)

In [61]: IN
Out[61]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

matrix_power

Given a matrix A and positive integer value n we can calculate A^n using matrix_power(A,n).

```
In [65]: A=matrix([[1,2],[3,4]])

In [66]: matrix_power(A,2)
Out[66]:
matrix([[ 7, 10],
        [15, 22]])
```

A^n

linalg.solve

Assumes that det ≠ 0

Solving linear systems.

$A \underline{x} = \underline{b}$

$$\left.\begin{aligned} ax + by &= p \\ cx + dy &= q \end{aligned}\right\}$$

$A \begin{pmatrix} \text{row } 1 \\ \text{row } 2 \end{pmatrix}$

```
In [25]: A = ([[4,3], [5,-3]])

In [26]: y = ([6,21])

In [27]: x = linalg.solve(A, y) # solve Ay=x

In [28]: print x
[ 3. -2.]
```

y = RHS of linear system

```
In [31]: from numpy.linalg import *

In [32]: # Now use x=solve(A,y)
```

$A \underline{x} = y$

unknown

Exercises

$$\underline{A}\,\underline{x} = \underline{b}$$

$$A = \begin{pmatrix} 4 & 3 \\ 3 & -5 \end{pmatrix} \quad \underline{b} = \begin{pmatrix} 19 \\ 7 \end{pmatrix}$$

- Solve $\left.\begin{aligned} 4x + 3y &= 19 \\ 3x - 5y &= 7 \end{aligned}\right\}$

- Solve $\left.\begin{aligned} 2x + y - 2z &= 10 \\ 3x + 2y + 2z &= 1 \\ 5x + 4y + 3z &= 4 \end{aligned}\right\}$

24

- Solve $\left.\begin{aligned} x + 2y - 3z &= 6 \\ 2x - y + 4z &= 2 \\ 4x + 3y - 2z &= 14 \end{aligned}\right\}$

Eigenvalues

λ s.t $\underline{x}$...

$A\underline{x} = \lambda \underline{x}$

e-value

eigenvector

Recall the matrix $\begin{pmatrix} 3 & 3 & 3 \\ 3 & -1 & 1 \\ 3 & 1 & -1 \end{pmatrix}$

with eigenvalues $\lambda_1 = 6; \lambda_2 = -3; \; \lambda_3 = -2$

$$\begin{pmatrix} 1/\sqrt{3} & 0 & 2/\sqrt{6} \\ -1/\sqrt{3} & 1/\sqrt{2} & 1/\sqrt{6} \\ -1/\sqrt{3} & -1/\sqrt{2} & 1/\sqrt{6} \end{pmatrix}$$

$|A - \lambda I| = 0 \longrightarrow \lambda_i$

Computing in Python

$$\begin{pmatrix} 3 & 3 & 3 \\ 3 & -1 & 1 \\ 3 & 1 & -1 \end{pmatrix}$$

```
In [39]: A=array([[3,3,3],[3,-1,1],[3,1,-1]])

In [40]: print eig(A)
(array([ 6., -3., -2.]), array([[ 0.81649658,  0.57735027,  0.        ],
       [ 0.40824829, -0.57735027, -0.70710678],
       [ 0.40824829, -0.57735027,  0.70710678]]))

In [41]: print det(A)
36.0

In [42]: print inv(A)
[[ 0.          0.16666667  0.16666667]
 [ 0.16666667 -0.33333333  0.16666667]
 [ 0.16666667  0.16666667 -0.33333333]]
```

normalized e-vectors

given vector $\underline{x}$

$$\frac{\underline{x}}{|\underline{x}|} = \hat{\underline{x}}$$

Exercise

Compute the eigenvalues (and normalised eigenvectors) of the matrix

$$A = \begin{pmatrix} 2 & 0 & 1 \\ -1 & 2 & 3 \\ 1 & 0 & 2 \end{pmatrix}$$

Ex: $B = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 3 & 0 \\ 6 & 9 & 4 \end{pmatrix}$ $\quad |B| = ?$

$C = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ $\quad |C| = ?$ $\quad C^{-1} = -\frac{1}{2}\begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} = \begin{pmatrix} -2 & 1 \\ 1.5 & -0.5 \end{pmatrix}$

Numerical Integration

$$\int_{-1}^{1} e^{x} \cos x \, dx$$

integrate.quad is a function for adaptive numerical quadrature of one-dimensional integrals.

$$\int_{-5}^{+5} e^{-x^2} \, dx$$

```python
import numpy as np
from math import *
from scipy import integrate

def function(x):
    return np.exp(-x**2)
value, error = integrate.quad(function,-5,5)
print(value)
print(error)
print 'The square root of pi is ', pi**0.5
```

*

e^{-x^2}

Python

```
In [70]: %run "c:\users\riaz\appdata\local\temp\tmpiwbsst.py"
1.7724538509
4.63662302997e-14
The square root of pi is  1.77245385091
```

$$\int_{1}^{1.5} x^2 \ln x \, dx$$

log(x, e)

Numerical Analysis
Burden & Faires

Gaussian quadrature scheme

Exercises

Use integrate.quad to calculate the integrals given earlier

Using error fn.

Root finding – Bisection, Newton

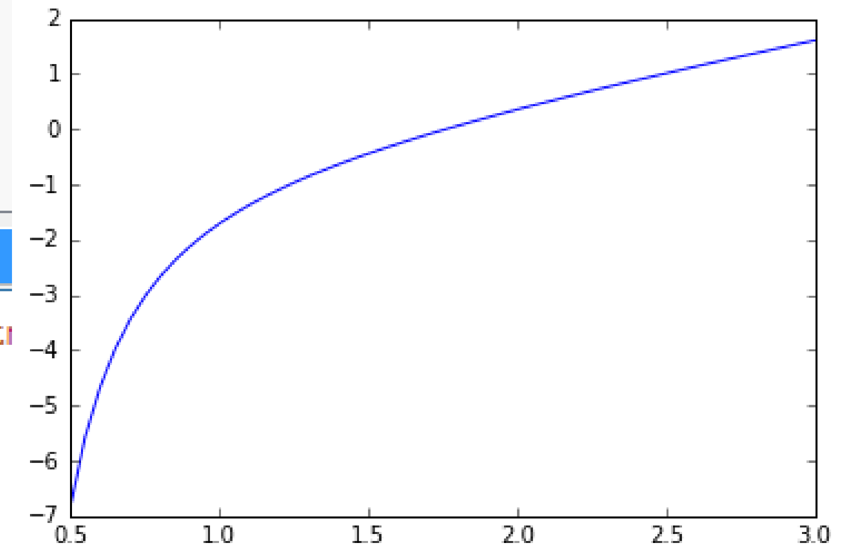
```python
from math import *
from scipy import optimize
def function(x):
    return (x-exp(1.0/x))

# find zero of function with initial point 1 by Newton -Raphson
value1 = optimize.newton (function , 1)
print 'Newton-Raphson gives', value1
# find zero between (1,2) by bisection
value2 = optimize.bisect(function , 1, 2)
print 'Bisection gives', value2
```

$f(x) = x - e^{1/x}$

$f(x) = 0 \iff x = e^{1/x}$

Python

```
In [143]: %run "c:\users\riaz\appdata\local\temp\t
Newton-Raphson gives 1.76322283435
Bisection gives 1.76322283435
```



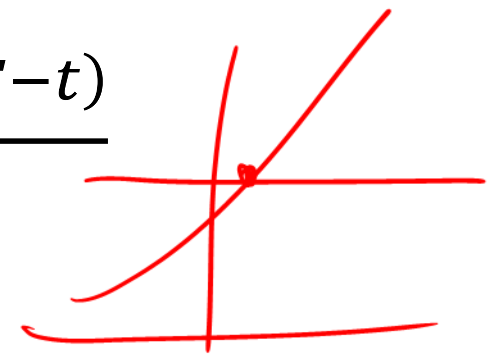
$f(x) = x - \cos x$; $x_0 = 1$; $[0, 2]$

Exercises

$= \left(-SN(-d_1) + Ee^{-r(T-t)}N(-d_2) \right)$

- Use both Bisection and Newton-Raphson methods to solve the equation $x^{x/3} - 2 = 0$.
- Use the Black-Scholes option pricing formula for a call given by $C = SN(d_1) - Ee^{-r(T-t)}N(d_2)$

where
$$d_{1,2} = \frac{log\left(\frac{S}{E}\right) + \left(r \pm \frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}$$



$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}s^2} ds$$

$S = E = 100$

$r = 0.05$

$\sigma = 0.2$

$T = 1 \quad (T-t)$

Probability and Statistics Functions

NumPy and SciPy have been introduced earlier. Both packages contain powerful functionality for performing important computation for the purposes of simulation, probability distributions and statistics operations.

To import, use import numpy as np and then calling np.random.rand, for example, although there are a number of ways.

rand, rand_sample( ) $U(0,1)$

rand and random_sample are uniform random number generators, i.e. $U(0,1)$ which are identical except that rand takes a variable number of integer inputs – one for each dimension – while random_sample takes a n-element tuple.

To sample $U(a, b)$, $b > a$; multiply the output of random_sample by $(b - a)$and add a.

$$\boxed{(b-a)\times\text{random\_sample}(\quad) + a \sim U(a,b)}$$

$(b - a) \times U(0,1) + a$ rand( )

Example use

rand( )

4 columns of pairs

```
In [20]: rand(3,2,4)
Out[20]:
array([[[ 0.73170027,  0.86928959,  0.34703755,  0.67843162],
        [ 0.97274622,  0.11809435,  0.4696357 ,  0.60403146]],

       [[ 0.91070563,  0.22309582,  0.66621535,  0.25170298],
        [ 0.40987248,  0.69587554,  0.07663795,  0.26369868]],

       [[ 0.80214693,  0.02357692,  0.14175233,  0.72294442],
        [ 0.57736639,  0.90774534,  0.06725288,  0.82607822]]])

In [21]: random_sample(5)
Out[21]: array([ 0.29255845,  0.66728231,  0.88173569,  0.75799814,  0.7720085 ])
```

1

2

3

ones(L, M, N)

L blocks of (M x N)

randn and standard_normal

$N(0,1)$

randn and standard_normal $N(0,1)$ are standard normal random number generators. randn, like rand, takes a variable number of integer inputs, and standard_normal takes an n-element tuple. Both can be called with no arguments to generate a single standard normal (e.g. randn()).

ϕ

Random numbers don’t exist!

Computer simulated random numbers are usually constructed from very complex but ultimately deterministic functions. These are not purely random, but are pseudo-random. All pseudo-random numbers in NumPy use one core random number generator based on the Mersenne Twister, a generator which can produce a very long series of pseudo-random data before repeating (up to 2^{19937}-1 non-repeating values). It also provides a much larger number of distributions to choose from.

Random Array Functions 1

shuffle( ) randomly reorders the elements of an array (only single element parameter)

```
In [6]: x=[1,2,3,4,5,6,7,8,9,10]

In [7]: x
Out[7]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [8]: shuffle(x)

In [9]: x
Out[9]: [6, 4, 8, 9, 2, 1, 10, 7, 5, 3]

In [10]: shuffle(x)

In [11]: x
Out[11]: [2, 8, 5, 3, 1, 4, 7, 9, 6, 10]
```

define array

randomly reorder

New x

Random Array Functions 2

permutation( ) returns randomly reordered elements of an array as a copy while not directly changing the input.

```
In [22]: x=arange(10) #initialise array [0,10]

In [23]: x
Out[23]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [24]: y=permutation(x) # create array y, copy of x

In [25]: y
Out[25]: array([5, 4, 3, 6, 8, 9, 1, 0, 2, 7])

In [26]: x # print x which is unaltered
Out[26]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

reordered x

Random Number Generators

$B(n, p)$

$p = \mathbb{P}(\text{success})$ $q = 1 - p$

binomial

The function binomial(n, p) generates a sample from the Binomial(n, p) distribution. binomial$\big(n, p, (a, b)\big)$ draws an array of dimension a by b from the Binomial(n, p) distribution.

```
In [2]: binomial(30,0.25)
Out[2]: 5

In [3]: binomial(30,0.25, (2,3))
Out[3]:
array([[10,  9, 10],
       [ 8,  4,  6]])
```

uniform

uniform( ) draws a uniform random variable on (0,1). uniform(low, high) generates a uniform on (l , h). uniform(low, high, (m, n)) generates a m by n array of uniforms on (l , h).

```
In [2]: uniform(1,4)
Out[2]: 1.6434898012426484

In [3]: uniform(1,4,(2,2))
Out[3]:
array([[ 3.95529755,  1.80478578],
       [ 3.48723773,  1.33462507]])
```

$U(a,b)$ $a < b$

lognormal

lognormal() generates a draw from a Log-Normal distribution with $\mu = 0$; $\sigma = 1$.

lognormal(mu, sigma, (m,n)) generates a m by n array or Log-Normally distributed data where the underlying Normal distribution has mean parameter μ and scale parameter σ.

```
In [4]: z=lognormal(0.0,1.0,(3,3))

In [5]: z
Out[5]:
array([[  1.31186793,  12.70848672,   5.20855017],
       [  1.13120087,   1.18552074,   3.79738026],
       [  1.6662822 ,   0.33650869,   2.18319642]])
```

normal $X \sim N(\mu, \sigma^2)$ $\phi = \frac{X-\mu}{\sigma}$

normal( ) generates draws from a standard Normal (Gaussian). normal(mu, sigma) generates draws from a Normal with mean μ and standard deviation σ. normal(mu, sigma, (n,m)) generates a n by m array of draws from a Normal with mean μ and standard deviation σ. normal(mu, sigma) is equivalent to

$$X = \mu + \sigma\varphi$$

which gives $X \sim N(\mu, \sigma^2)$.

poisson

poisson( ) generates a draw from a Poisson distribution with $\lambda = 1$. poisson(lambda) generates a draw from a Poisson distribution with expectation λ. poisson(lambda, (n,m)) generates a n by m array of draws from a Poisson distribution with expectation λ.

```
In [6]: poisson(1)
Out[6]: 0

In [7]: poisson(1)
Out[7]: 2

In [8]: poisson(3,(2,2))
Out[8]:
array([[0, 2],
       [2, 1]])
```

Seed is better!

numpy.random.seed is a more useful function for initializing the random number generator, and can be used in one of two ways. seed( ) will initialize (or reinitialize) the random number generator using some actual random data provided by the operating system. seed(*s*) takes a vector of values (can be scalar) to initialize the random number generator at particular state. seed(*s*) is particularly useful for producing simulation studies which are reproducible.

In the following sample code, calls to seed() produce different random numbers, since these reinitialize using random data from the computer, while calls to seed(0) produce the same (sequence) of random numbers.

```
In [20]: seed(0)

In [21]: standard_normal()
Out[21]: 1.764052345967664

In [22]: seed(0)

In [23]: standard_normal()
Out[23]: 1.764052345967664

In [24]: seed()

In [25]: randn()
Out[25]: 1.8389480935275662

In [26]: seed()

In [27]: randn()
Out[27]: -0.5762730584167648
```

Statistics

Now we explore the statistical functions available.

mean computes the average of an array. An optional second argument provides the axis to use (default is to use entire array). mean can be used either as a function or as a method on an array.

```
In [22]: x
Out[22]:
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
        5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])

In [23]: x.mean() # method
Out[23]: 4.75

In [24]: mean(x) # function
Out[24]: 4.75
```

OOP

8

std

std(   ) computes the standard deviation of an array. An optional second argument provides the axis to use (default is to use entire array). As with the mean function std can be used either as a function or as a method on an array.

```
In [1]: from numpy import *

In [2]: a = random.standard_normal(1000) # using method

In [3]: a = standard_normal(1000) # using function

In [4]: std(a)
Out[4]: 0.9838637471999022

In [5]: mean(a)
Out[5]: 0.058909504169302032
```

a.std

a.mean

var

var( ) calculates the variance of an array as function or method. Also has an optional second argument to provide the axis to use (default is to use entire array).

```
In [6]: a.var()
Out[6]: 0.96798787305423306
```

method

var(a) function

Generate random numbers by Python

Random numbers can be generated by numpy.random.import numpy.random as npr

import matplotlib.pyplot as plt import numpy as np

- X = npr.standard_normal((5000)) $N(0,1)$
- Y = npr.normal(1, 1, (5000)) $N(\mu, \sigma^2)$
- Z = npr.uniform(-3, 3, (5000)) $U(a,b)$
- W = npr.lognormal(0, 1, (5000)) LN

The code

Importation of libraries

```
import numpy.random as npr
import matplotlib.pyplot as plt
import numpy as np
from pylab import *
```

Random No.s are generated

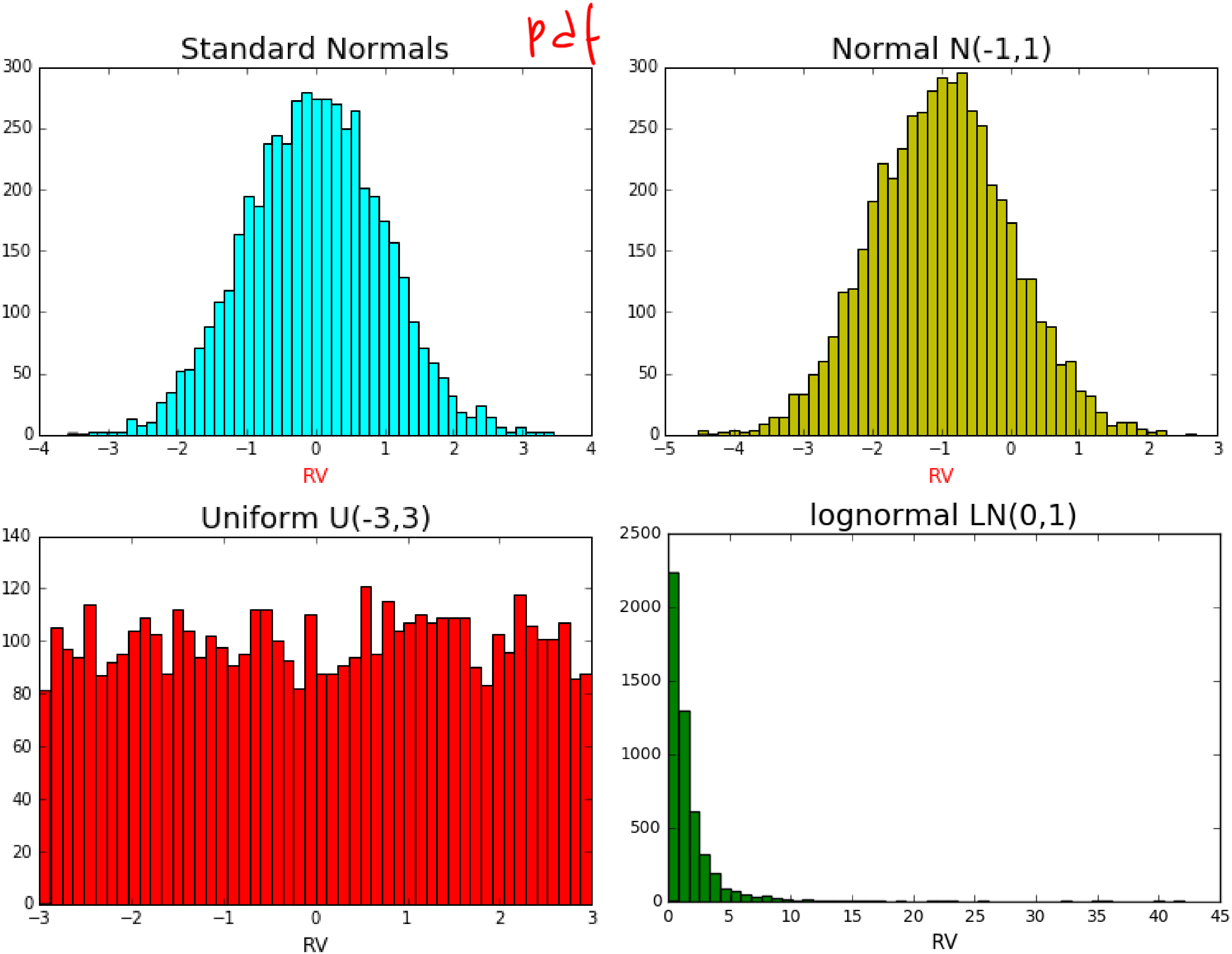
```
X = npr.standard_normal((5000)) # N(0,1)
Y = npr.normal(-1, 1, (5000)) # N(mu, var)
Z = npr.uniform(-3, 3, (5000)) # U(a,b)
W = npr.lognormal(0, 1, (5000))
```

```
plt.hist(X, bins = 50, cumulative=False, color='cyan')
plt.title('Standard Normals', fontsize = 18)
plt.xlabel('RV', color='r', fontsize=12)
show()
```

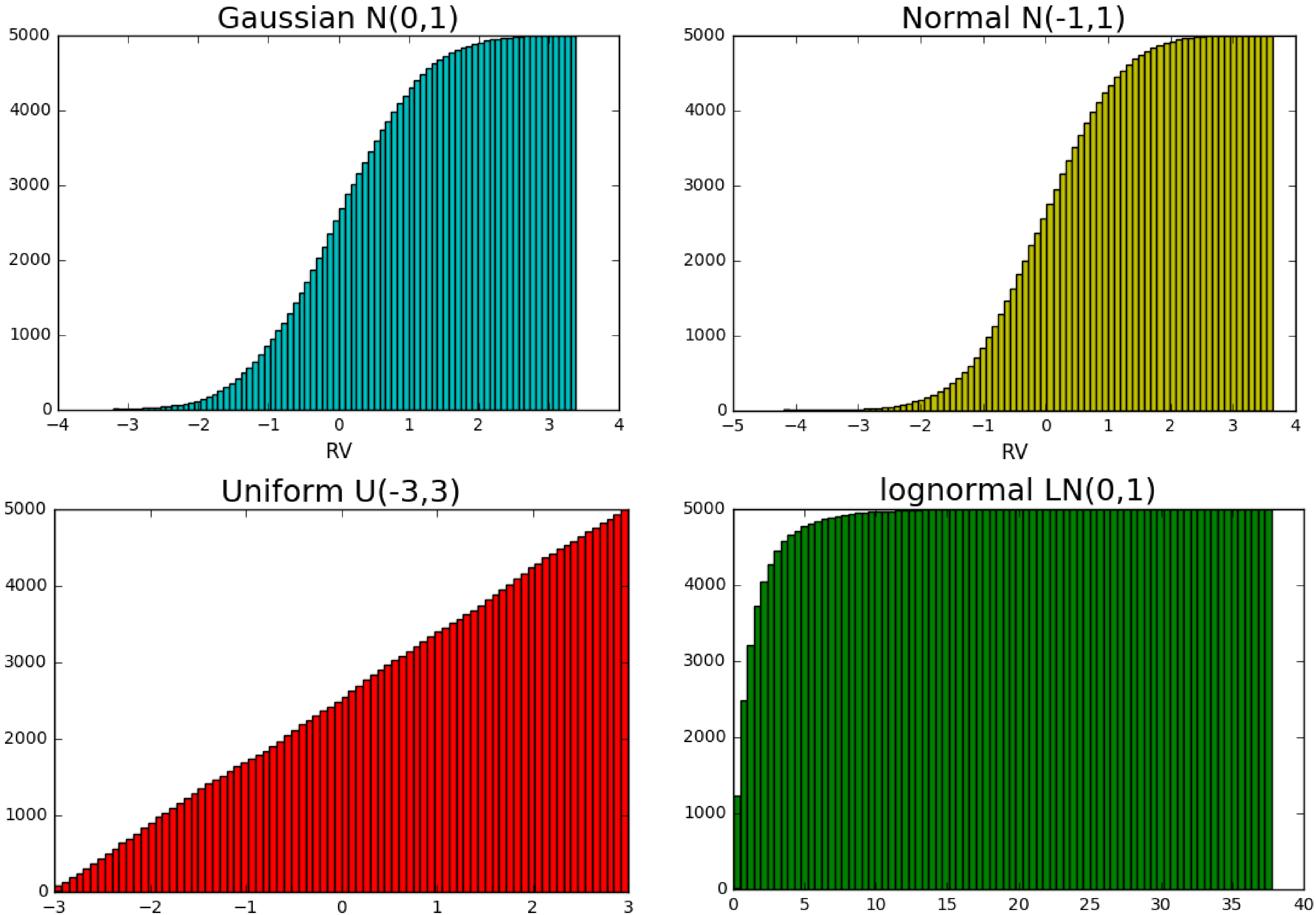
Plotting histogram

By changing to True → CDF

With FALSE → pdf


pdf
Standard Normals
300
250
200
150
100
50
0
−4
−3
−2
−1
0
1
2
3
4
RV
Normal N(-1,1)
300
250
200
150
100
50
0
−5
−4
−3
−2
−1
0
1
2
3
RV
Uniform U(-3,3)
140
120
100
80
60
40
20
0
−3
−2
−1
0
1
2
3
RV
lognormal LN(0,1)
2500
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
40
45
RV

CDF


Gaussian N(0,1)
Normal N(-1,1)
Uniform U(-3,3)
lognormal LN(0,1)
RV
RV
0
1000
2000
3000
4000
5000
−4
−3
−2
−1
0
1
2
3
4
−5
5
10
15
20
25
30
35
40

Exercise

rand(N) $\rightarrow$ N lots of $U(0,1)$

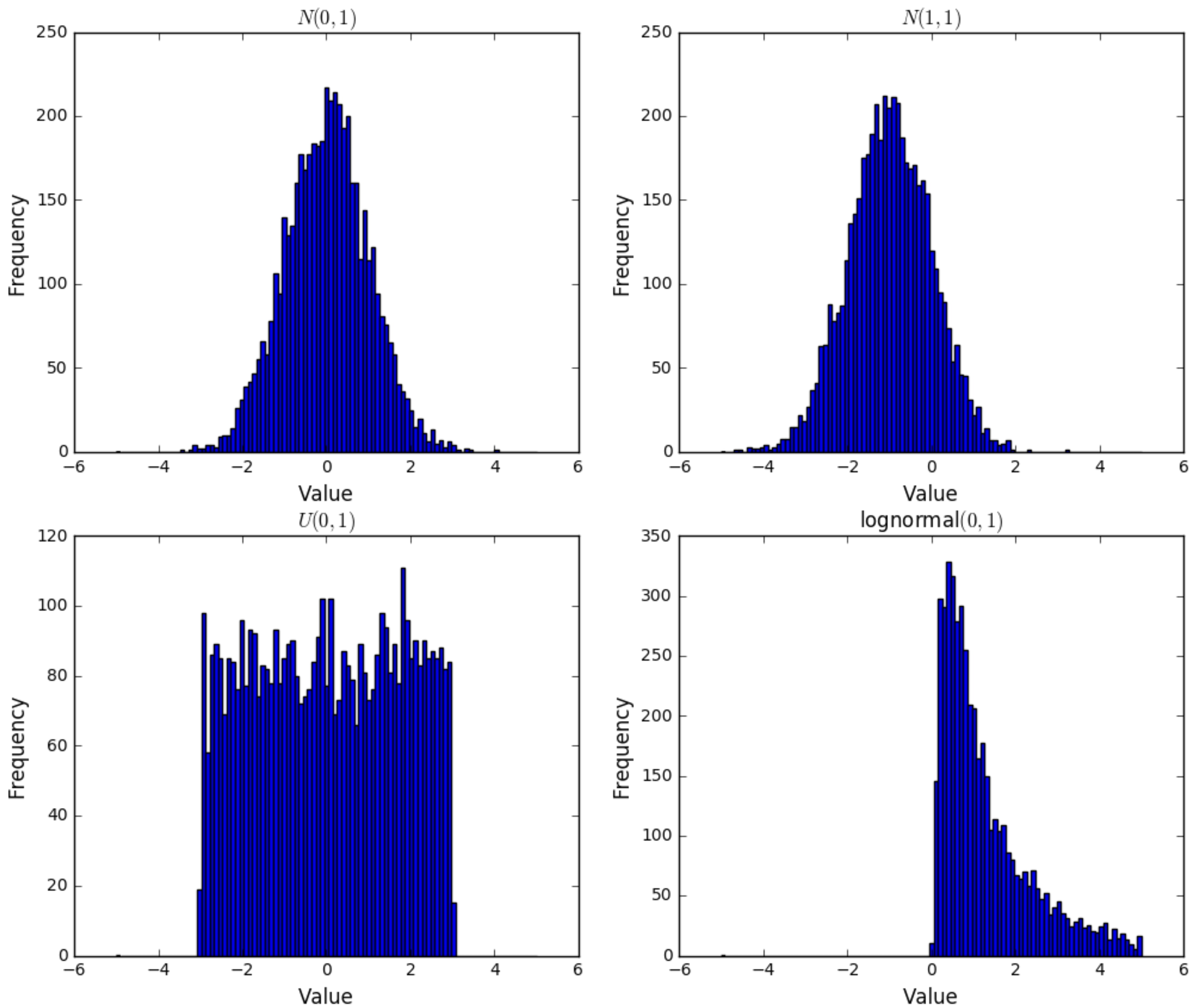
Use the earlier relationship between the CDF for the Normal distribution and the error function to generate a number of $U(0,1)$. Check the resulting mean and variance

$$N(x) = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

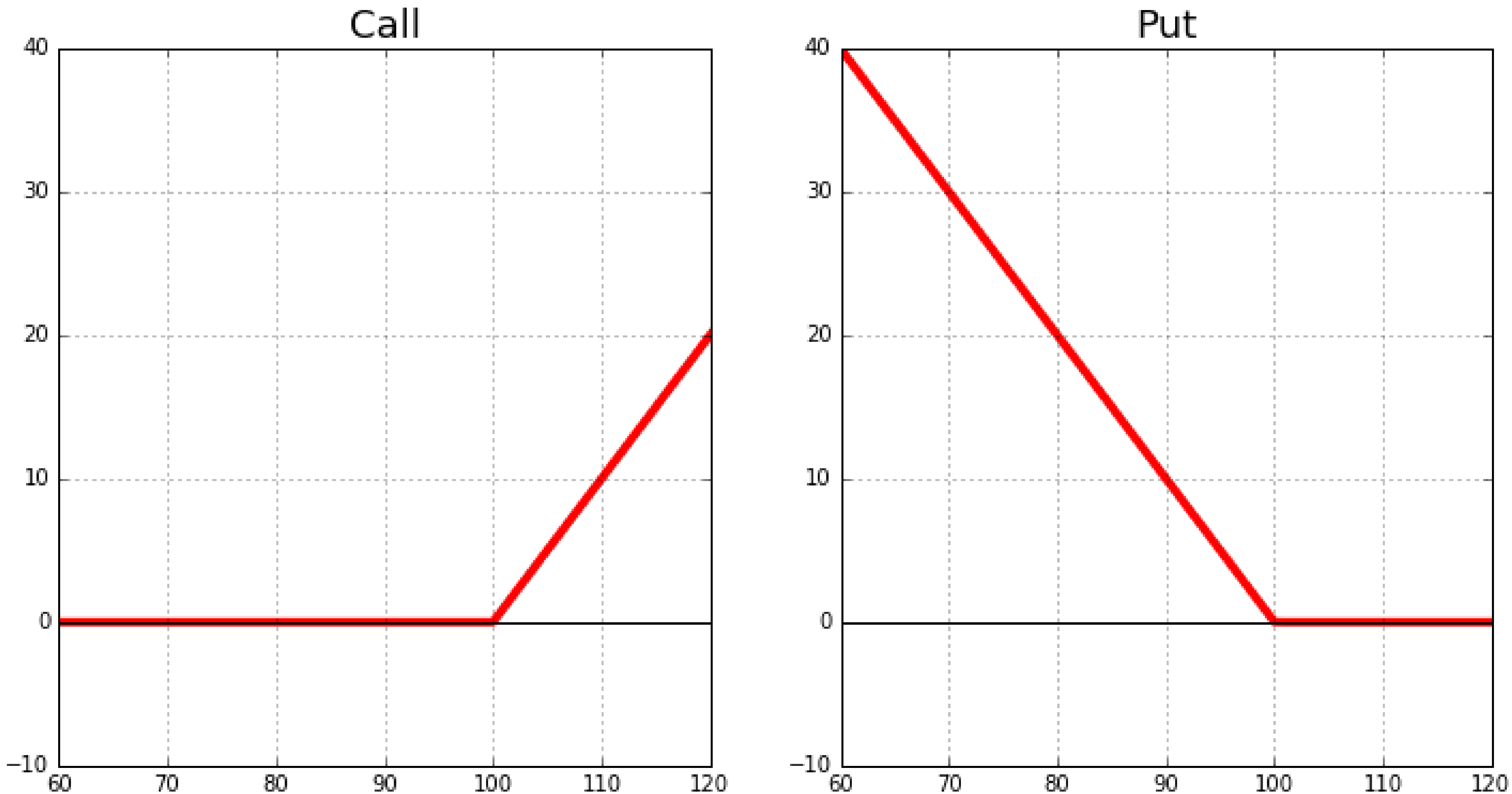
$$\sqrt{2}\,\mathrm{erfinv}\left[(2y-1)\right] \sim N(0,1)$$

$\hookrightarrow U(0,1)$

Subplots


$N(0,1)$
$N(1,1)$
$U(0,1)$
lognormal$(0,1)$
Frequency
Value
250
200
150
100
50
0
120
100
80
60
40
20
350
300
−6
−4
−2
0
2
4
6

```
bins = np.linspace(-5, 5, 100)

fig = plt.figure(figsize = (12, 10))
sub1 = fig.add_subplot(221)
plt.hist(X, bins)
plt.title("$N(0,1)$", fontsize = 12)
plt.xlabel("Value", fontsize = 12)
plt.ylabel("Frequency", fontsize = 12)

sub2 = fig.add_subplot(222)
plt.hist(Y, bins)
plt.title("$N(1,1)$", fontsize = 12)
plt.xlabel("Value", fontsize = 12)
plt.ylabel("Frequency", fontsize = 12)

sub3 = fig.add_subplot(223)
plt.hist(Z, bins)
plt.title("$U(0,1)$", fontsize = 12)
plt.xlabel("Value", fontsize = 12)
plt.ylabel("Frequency", fontsize = 12)

sub4 = fig.add_subplot(224)
plt.hist(W, bins)
plt.title("lognormal$(0,1)$", fontsize = 12)
plt.xlabel("Value", fontsize = 12)
plt.ylabel("Frequency", fontsize = 12)

fig = plt.gcf()
show()
```


Call
Put
40
30
20
10
0
−10
60
70
80
90
100
110
120

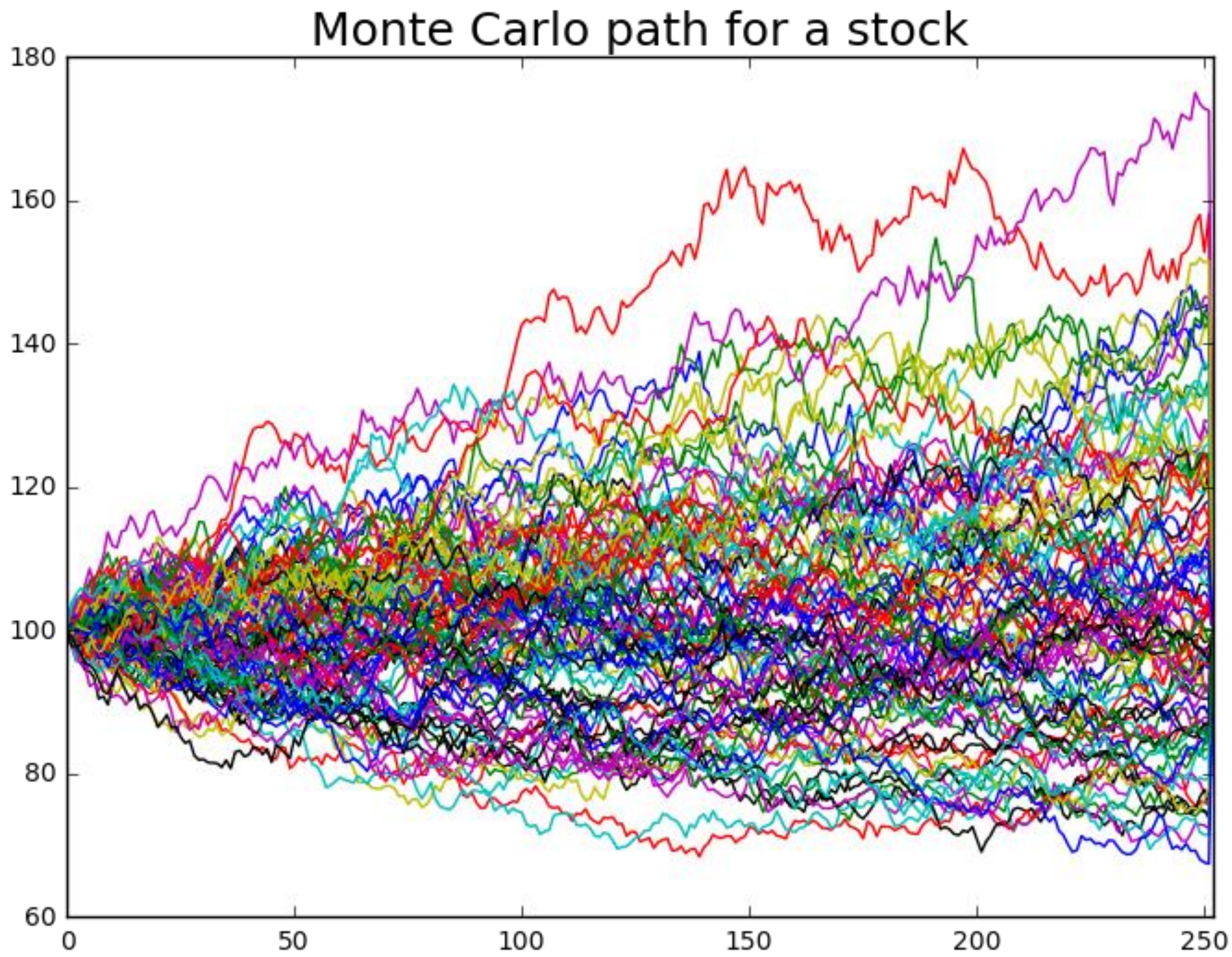
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import *

def pffcall(S, K):
    return np.maximum(S - K, 0.0)
def pffput(S, K):
    return np.maximum(K - S, 0.0)
S = np.linspace(50, 151, 100)
fig = plt.figure(figsize=(12, 6))

sub1 = fig.add_subplot(121) # col, row, num
sub1.set_title('Call', fontsize = 18)
plt.plot(S, pffcall(S, 100), 'r-', lw = 4)
plt.plot(S, np.zeros_like(S), 'black',lw = 1)
sub1.grid(True)
sub1.set_xlim([60, 120])
sub1.set_ylim([-10, 40])

sub2 = fig.add_subplot(122)
sub2.set_title('Put', fontsize = 18)
plt.plot(S, pffput(S, 100), 'r-', lw = 4)
plt.plot(S, np.zeros_like(S), 'black',lw = 1)
sub2.grid(True)
sub2.set_xlim([60, 120])
sub2.set_ylim([-10, 40])
```

MC Simulations

```
import numpy as np
import numpy.random as npr
import matplotlib.pyplot as plt

S = 100
T = 1.0
r = 0.05
vol = 0.2

I = 100 # MC paths
N = 252
dt=T/N
Z = npr.standard_normal((N+1, I))
St = S * np.ones((N+1, I))
rnd = np.exp((r - 0.5 * vol**2) * dt + vol * np.sqrt(dt) * Z)
for i in range(1, N):
    St[i] = St[i-1] * rnd[i-1]

plt.figure(figsize = (8, 6))
for k in range(I):
    plt.plot(np.arange(N+1), St[:, k])
plt.xlim(0,252)
plt.title('Monte Carlo path for a stock', fontsize = 18)
```

vary

$\delta t = \frac{1}{N}$


Monte Carlo path for a stock
180
160
140
120
100
80
60
0
50
100
150
200
250

Exercise 1

V: $dr = \gamma(r - \bar{r})dt + \sqrt{\beta}\,\phi\sqrt{dt}$

CIR: $dr = -\gamma(\bar{r} - r)dt \sqrt{\beta r}\,\phi\sqrt{dt}$

Simulate a series of sample paths for both Vasicek and CIR models using the input parameters $\bar{r} = 0.055$; $\gamma = 3.0$; $\sqrt{\beta} = 0.015$; spot rate is 0.05. The same random numbers should be used for both models.

$\bar{r} = \eta/\gamma$

V: $r_{i+1} = r_i + \gamma(r_i - \bar{r})dt + \sqrt{\beta}\,\phi\sqrt{dt}$

CIR: $r_{i+1} = r_i + \gamma(r_i - \bar{r})dt + \sqrt{\beta r_i}\,\phi\sqrt{dt}$

Exercise 2

OU P. ps

Consider the process

$$x_t = x_0 e^{-\theta t} + \mu\left(1 - e^{-\theta t}\right) + \sigma\sqrt{\frac{1-e^{-\theta t}}{2\theta}}\,\emptyset\,;$$

$\emptyset \sim N(0,1)$. Using the following parameters x_0=1, $\theta = 1, \mu = 1, \sigma = 0.5$

Generate 10000 MC paths for $t = 10$. Obtain the numerical mean of the paths and compare with the mean given by $x_0 e^{-\theta t} + \mu\left(1 - e^{-\theta t}\right)$.