MATH40006: An Introduction To Computation
Module notes, Section 8

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

8 `random`, `numpy.random`, `matplotlib` and `matplotlib.pyplot`

8.1 The `random` module and NumPy's `random` submodule

There are two significant resources for statistical tasks (in the broad sense) in Python: the "Core Python" module `random`, and the submodule of the same name that forms part of `numpy`.

The `random` module has one basic function, also called `random`, which takes no arguments, and returns a randomly chosen float (a Core Python float) between 0 and 1:

```
import random as rnd
print(rnd.random())
print(rnd.random())
print(rnd.random())
```

```
0.24853329350505304
0.9876234408428399
0.34253757101753357
```

There are then several subsidiary functions that piggyback on `random`. There's `randint`, which returns a random integer between two values.

```
print(rnd.randint(4,10))
print(rnd.randint(1,1000))
print(rnd.randint(4,4))
```

```
8
708
4
```

Notice that `randint(a, b)` returns a random int n such that $a \leq n \leq b$; this is arguably a slight deviation from the usual Python convention of "less than or equal at one end, strictly less than at the other".

A bit more generally, there's then `randrange`, which returns a randomly chosen int from the object `range(start, stop, step)` (although it doesn't actually create that object).

```
print(rnd.randrange(4,5))
print(rnd.randrange(1,1000,2))
print(rnd.randrange(11))
```

```
4
479
6
```

Notice that this *does* obey the convention "less than or equal at one end, strictly less than at the other"!

More generally still, there's `choice`, which chooses randomly from a given sequence:

```
print(rnd.choice([1,1.0,1+0j,'one']))
print(rnd.choice(range(1,1000,2)))
print(rnd.choice('abcdefg'))
```

```
1.0
217
d
```

Then there's `choices`, which generates lists of such randomly picked members:

```
print(rnd.choices([1,1.0,1+0j,'one'], k=10))
print(rnd.choices(range(1,1000,2), k=20))
print(rnd.choices('abcdefg', k=5))
```

```
['one', 1.0, 'one', 1, 'one', 1, 1.0, 1.0, (1+0j), 'one']
[783, 673, 975, 253, 431, 797, 617, 247, 279, 227, 791,
139, 43, 243, 759, 285, 967, 817, 931, 391]
['a', 'e', 'f', 'e', 'g']
```

The `choices` function allows the user to specify weights, which cause options to be picked with different probabilities, proportional to the weights:

```
print(rnd.choices([1,1.0,1+0j,'one'], weights=[1,1,1,4], k=10))
print(rnd.choices('abcdefg', weights=[10,1,1,1,1,1,1], k=10))
```

```
[1.0, 1, 1.0, 'one', (1+0j), 1, 'one', 'one', 'one', 'one']
['a', 'a', 'a', 'e', 'a', 'b', 'a', 'a', 'e', 'a']
```

Related are the functions `sample`, which draws from a sequence without replacement, and `shuffle`, which shuffles the elements of a sequence in place. Note that `sample` generates output, whereas `shuffle` returns `None` and performs its shuffle **in place** as a side-effect.

```
print(rnd.sample([1,1.0,1+0j,'one'], 3))
print(rnd.sample('abcdefg', 4))
all_ones = [1,1.0,1+0j,'one']; rnd.shuffle(all_ones); print(all_ones)
```

```
[1.0, (1+0j), 1]
['g', 'd', 'f', 'a']
[(1+0j), 1.0, 'one', 1]
```

Because `shuffle` shuffles in place, you can't use it on immutable data structures like strings or tuples, and must instead make creative use of `sample`:

```
alphabet = 'abcdefg'
alphabet = ''.join(rnd.sample(alphabet,len(alphabet)))
print(alphabet)
```

```
cagdbef
```

Finally, there's a slightly limited range of **probability distributions** implemented in `random`, which you can sample:

```
print(rnd.uniform(5,6))
print(rnd.normalvariate(100,15))
```

```
5.128437075573844
110.6353290359451
```

Note that the parameters for the uniform distribution are the lower and upper bounds of the range, and those for the Normal the mean and standard deviation (*not* the variance).

The NumPy version of all this stuff is (a) array-compatible and (b) a bit turbocharged, as you'd predict. The function called `random` in this submodule works a little differently from the non-NumPy one; yes, you can get a single (NumPy) float between 0 and 1, but you can also get an array of them, of any dimension you like:

```
import numpy.random as nrnd
print(nrnd.random())
print(nrnd.random(5))
print(nrnd.random([3,2]))
print(nrnd.random(size=[3,2]))
```

```
0.971264235704511
[0.90662282 0.78890151 0.11711386 0.33967627 0.56968404]
[[0.73228846 0.04429268]
 [0.72977398 0.72109446]
 [0.02274657 0.09315491]]
```

```
[[0.64902511 0.11407875]
 [0.99828324 0.10626859]
 [0.18550121 0.46765575]]
```

NumPy's version of `randint` obeys the Python convention for inequalities; it too is capable of returning arrays:

```
print(nrnd.randint(1,5))
print(nrnd.randint(1,5,[4,4]))
```

```
4
[[1 1 2 1]
 [2 2 1 4]
 [3 2 3 4]
 [4 1 2 2]]
```

Note that the returned values are **NumPy** ints, which means they're not allowed to be of arbitrary size. So the first of these works but the second generates an error message:

```
print(rnd.randint(10**100,10**101))
print(nrnd.randint(10**100,10**101))
```

```
4858978821039943135232623881271712182360322482929729694665516009137896474700246950953246316624913569 5
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-41-f266798a9735> in <module>()
      1 print(rnd.randint(10**100,10**101))
----> 2 print(nrnd.randint(10**100,10**101))

mtrand.pyx in mtrand.RandomState.randint()

ValueError: high is out of bounds for int32
```

In the core `random` module, the `choices` function does repeated selection with replacement, and the `sample` function does it without. In NumPy's `random` submodule, both are handled by the function called `choice`:

```
import numpy as np
print(nrnd.choice(np.array([2, 3, 5, 7, 11])))
print(nrnd.choice(np.array([2, 3, 5, 7, 11]),5))
print(nrnd.choice(np.array([2, 3, 5, 7, 11]),5,replace=False))
```

```
3
[3 7 2 2 2]
[ 3  7  5  2 11]
```

There's a `permutation` function for generating a shuffled version of an array (as output), and a `shuffle` function for shuffling in place (as a side-effect):

```
print(nrnd.permutation(np.array([2, 3, 5, 7, 11])))

primes = np.array([2, 3, 5, 7, 11]); nrnd.shuffle(primes)
print(primes)
```

```
[ 7  2 11  5  3]
[ 3  7 11  5  2]
```

Finally, there's much more **probability distribution** functionality:

```
print(nrnd.normal(100,15))
print(nrnd.normal(100,15,5))
print(nrnd.binomial(10,0.3,5))
print(nrnd.poisson(3,[2,3]))
```

```
100.74505758683581
[112.45045275 128.03507447  82.97648018 116.62469513 109.40454701]
[4 5 3 3 3]
[[3 2 1]
 [5 7 6]]
```

And in fact loads, loads more; we've barely scratched the surface here!

8.2 `matplotlib` and `pyplot`

You're already somewhat familiar with the "workhorse" plotting function from the `pyplot` submodule of `matplotlib`, which is simply called `plot`. It's used for generating line plots, and point plots, of data in the form of x and y coordinates (the data can come in the form of 1D arrays, or lists, or tuples, or ranges). Let's make use of what we know about random numbers to do something a bit ambitious:

Challenge 1: write a function called `three_points_step` which takes as its input a 1D array of length 2 representing a pair of coordinates (x, y). It should then choose a random integer 0, 1 or 2. If it's equal to 0, it should return $(x/2, y/2)$; if 1, $(x/2 + 1/2, y/2)$; if 2, $(x/2 + 1/4, y/2 + \sqrt{3}/4)$.

Starting with $x = y = 0.5$, iterate this function 100 times, discarding the iterates.

Then iterate it a further 10000 times, keeping the iterates in an array. Finally, create a dot plot of these iterates you've retained.

This starts fairly straightforwardly:

```
def three_points_step(coords):
    from random import randint
    from numpy import sqrt, array
    r = randint(0,2)
    if r==0:
        return coords/2
    elif r==1:
        return coords/2 + array([1/2, 0])
    else:
        return coords/2 + array([1/4, sqrt(3)/4])

import numpy as np
coords = np.array([0.5, 0.5])

# iterate and discard
for n in range(100):
    coords = three_points_step(coords)
```

Then we want to set up an array with (at first) one row, consisting of our last pair of coordinates

```
# initialize array of coordinates
coords_array = np.array([coords])
```

Now, the next bit takes a little thought, because we need to work out how to append a row to an array. Neither of the following quite works, for example:

```
np.append(coords_array, coords)
np.append(coords_array, [coords])
```

In order to get NumPy's append function correctly, we need to specify dimension 0:

```
np.append(coords_array, [coords], 0)
```

(notice that NumPy's append generates output, rather than appending in place using a side-effect).

An alternative would be:

```
np.concatenate([coords_array, [coords]])
```

(note the extra pair of square brackets).

So:

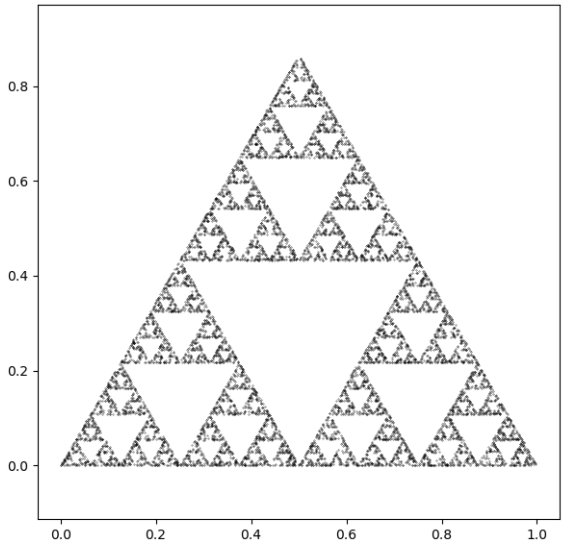

Figure 1: The Sierpinski triangle

```
# iterate and retain
for n in range(10000):
    coords = three_points_step(coords)
    coords_array = np.append(coords_array,[coords],0)
```

Then finally

```
import matplotlib.pyplot as plt
plt.figure(figsize=[7,7])
plt.axis('equal')
plt.plot(coords_array[:,0],coords_array[:,1],'k.',markersize=0.5)
```

The image we get forms Figure 1. Things to notice on the plotting side of things include:

- the way we've set the size of the figure;
- how to make the scales equal on both axes;
- the way we've picked out columns 0 and 1 of the coordinate array for plotting using the `:` notation (actually, this is sort of a NumPy observation too);
- the use of the optional keyword-only argument `markersize`.

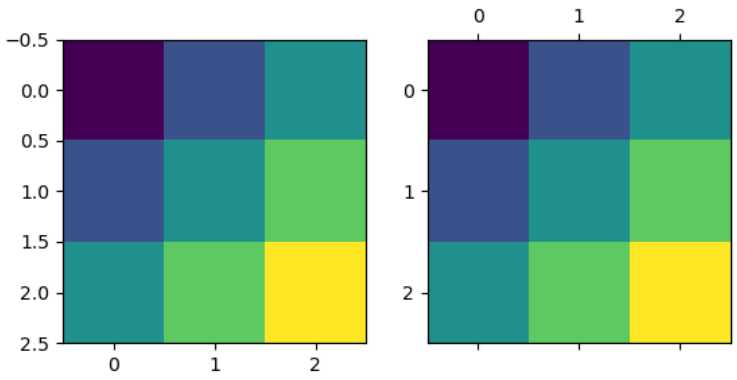

Figure 2: Visualisation of a (3 by 3) matrix using `imshow` (left) and `matshow` (right)

The **plot** function is very useful, but it's not the only thing out there. For example, there are also `matshow` and `imshow`, each of which take as its argument an array of values, and visualises that array using coloured squares:

```
mat1 = np.array([[0,1,2],[1,2,3],[2,3,4]])
fig, (ax0, ax1) = plt.subplots(1,2)
ax0.imshow(mat1)
ax1.matshow(mat1)
```

The image is shown as Figure 2. As you can see, the two functions differ somewhat in the conventions they use for annotating the axes. Observe, in passing, how we set up two subplots in the same grid; in this case, a 2 by 1 grid.

Both `imshow` and `matshow` use the convention that the positive vertical direction is *downwards*. This fits with the (row, column) convention for matrices, but is at odds with how we usually do things in the xy-plane. It can be overridden by setting the value of the keyword argument `origin` to `'lower'`:

```
mat1 = np.array([[0,1,2],[1,2,3],[2,3,4]])
fig, (ax0, ax1) = plt.subplots(1,2)
ax0.imshow(mat1,origin='lower')
ax1.matshow(mat1,origin='lower')
```

The image is shown as Figure 3.

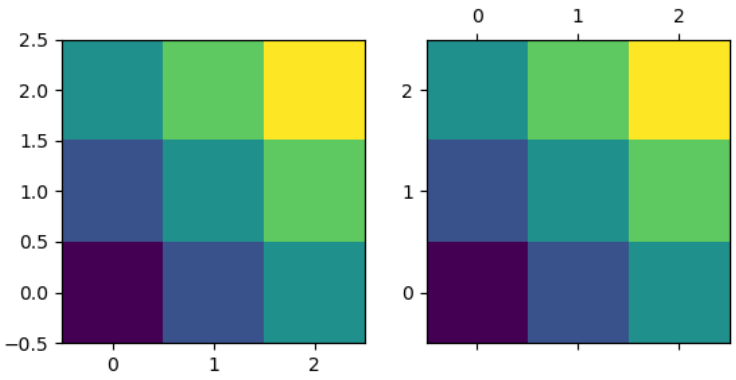

Figure 3: Visualisation of a (3 by 3) matrix using `imshow` (left) and `matshow` (right); ascending vertical axis

The `imshow` function gives us one way of visualizing a function of two variables. The tactic is to create values of $u(x, y)$ for values of x and y lying on a lattice. The most flexible way to do that uses a NumPy function called `meshgrid`. For example, to create the matrix we used in the last example, we could do this:

```
x_values = np.arange(3)
y_values = np.arange(3)
x, y = np.meshgrid(x_values, y_values)
mat1 = x+y
print(x)
print(y)
print(mat1)
```

```
[[0 1 2]
 [0 1 2]
 [0 1 2]]
[[0 0 0]
 [1 1 1]
 [2 2 2]]
[[0 1 2]
 [1 2 3]
```

```
[2 3 4]]
```

Look what `meshgrid` does; it allows us to create arrays `x` and `y`, standing for the x- and y-coordinates respectively of nine points on a lattice. The matrix then represents the values on that lattice of $u(x, y) = x + y$, and the diagram visualises that function on that lattice.

(In this case, of course, we could also have used `fromfunction`, but going via `meshgrid` allows us to use non-integer ranges, so it's much more flexible.)

Challenge 2: use `imshow` to create a visualisation of the function

$$u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$$

for 101 equally-spaced values of x and y, each between -2.0 and 2.0

```
x_values = np.linspace(-2.0,2.0,101)
y_values = np.linspace(-2.0,2.0,101)
x, y = np.meshgrid(x_values, y_values)

u = (x**2 - 2*y**2)*np.exp(-x**2-y**2)
plt.imshow(u,origin='lower',extent=[-2.0,2.0,-2.0,2.0])
```

The image is shown as Figure 4. Note the use of `origin='lower'`, and also the use of the keyword argument `extent` to set appropriate data ranges on the axes.

There are alternative ways to visualise a function of two variables. In two dimensions, the main one is a **contour plot**. By default, `matplotlib`'s `contour` function shows contours in a variety of colours; we can override this using the `colors` option.

```
fig, (ax1, ax2) = plt.subplots(1,2)
ax1.axis('equal')
ax2.axis('equal')
ax1.contour(x, y, u)
ax2.contour(x, y, u, colors='red')
```

The image is shown as Figure 5.

There are also three important 3D visualisation tools for functions of two variables: `plot_wireframe`, `plot_surface` and the 3D version of `contour`. The way we use them, though, is a little different from in 2D:

```
from mpl_toolkits.mplot3d import Axes3D
ax = plt.axes(projection='3d')
ax.plot_wireframe(x, y, u)
```

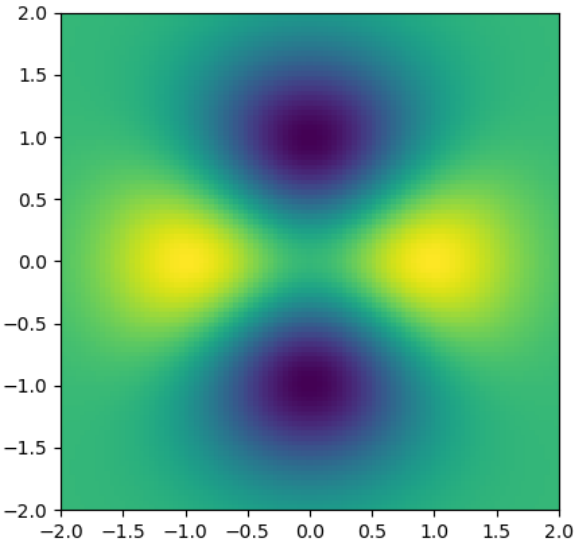

Figure 4: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using `imshow`

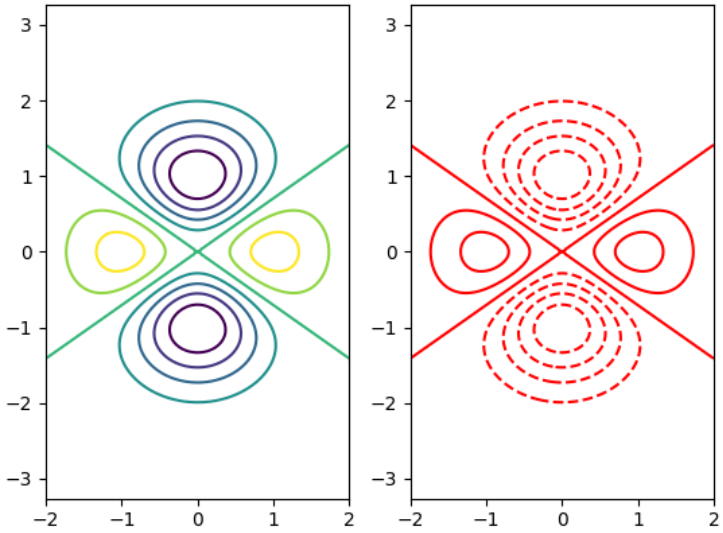

Figure 5: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using `contour`

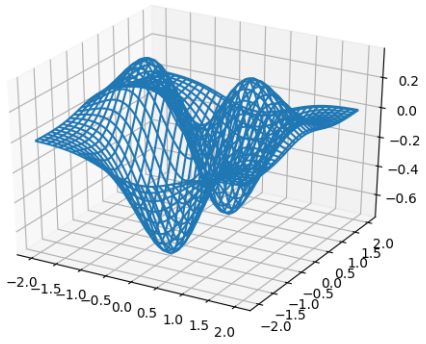

Figure 6: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using a wireframe

```
ax = plt.axes(projection='3d')
ax.plot_surface(x, y, u)
```

```
ax = plt.axes(projection='3d')
ax.contour(x, y, u, colors='red')
```

Another important 2D diagram is the **vector field plot**, sometimes called the **quiver plot**; this consists of a field of arrows representing a vector field $\mathbf{f}(x, y)$, in which each pair of coordinates is associated with a vector. The `matplotlib` function we use for this is called `quiver`; it takes four arguments: values of x and y specifying the positions of the arrows, and the two components of $\mathbf{f}(x, y)$, specifying the corresponding vectors. Here, for example, is a plot of the two partial derivatives of our function

$$u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$$

with respect to x and y respectively We've used a 21 by 21 lattice of points (in general, you should use a coarser mesh for quiver plots than for, say, contour plots).

```
X_values = np.linspace(-2.0,2.0,21)
Y_values = np.linspace(-2.0,2.0,21)
X, Y = np.meshgrid(X_values, Y_values)

dudx = -2*X*(-1+X**2-2*Y**2)*np.exp(-X**2-Y**2)
dudy = -2*Y*(2+X**2-2*Y**2)*np.exp(-X**2-Y**2)
plt.axis('equal')
plt.quiver(X, Y, dudx, dudy)
```

The image appears as Figure 9. The quiver plot looks interesting superimposed on a contour

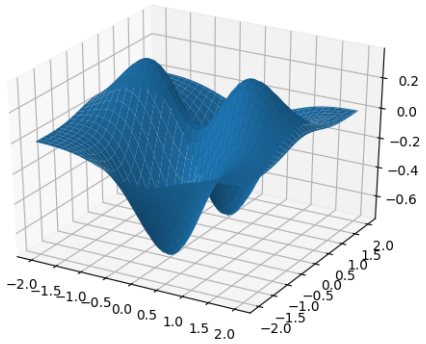

Figure 7: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using a surface plot

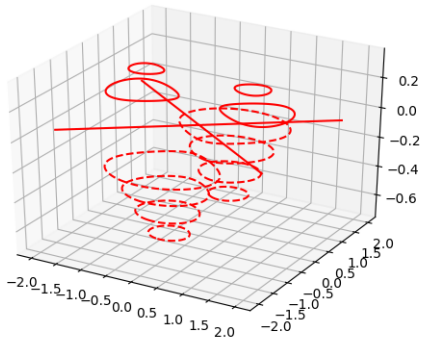

Figure 8: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using a 3D contour plot

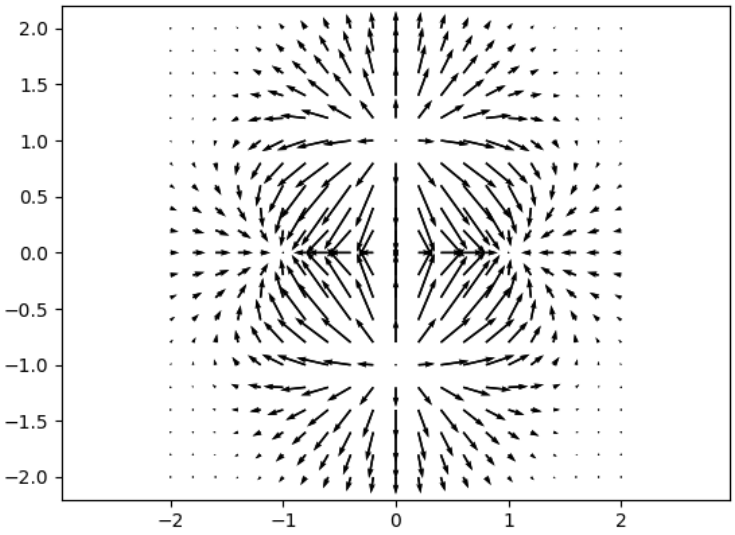

Figure 9: Quiver plot of the partial derivatives of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$

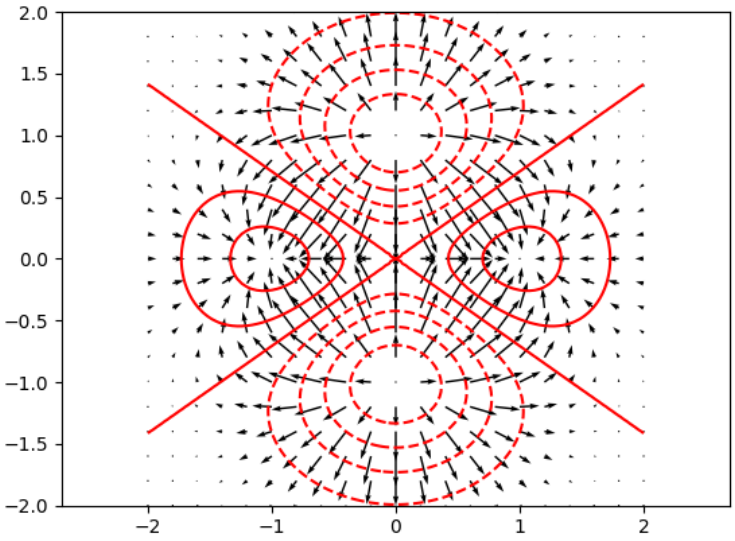

Figure 10: Quiver plot of the partial derivatives of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$, superimposed on contour plot of u

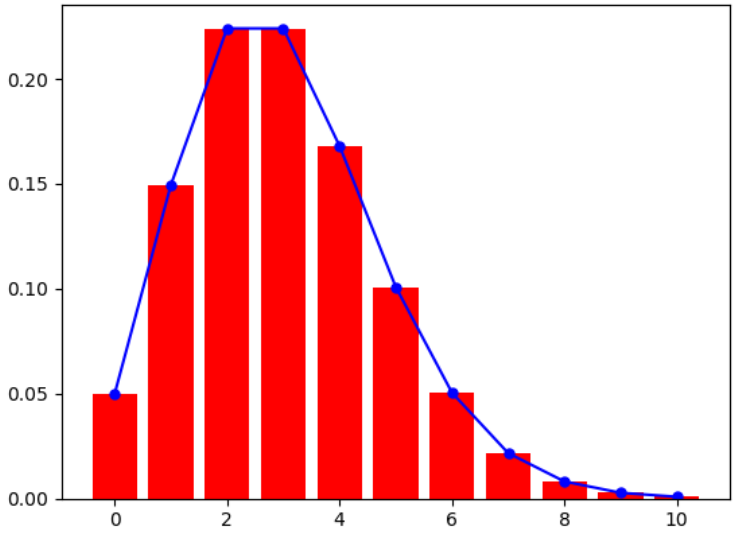

Figure 11: Visualisation of a Poisson distribution

plot of the original function; this is shown in Figure 10.

Finally, `pyplot` supports a variety of **statistical diagrams**: bar charts, histograms, pie charts, box and whisker plots, scatter diagrams and many others.

Challenge 3: calculate, as a 1D array, the values of $r!$ for r from 0 to 10, and hence the values of the pdf for those values of a random variable that is Poisson distributed with parameter 3.0. Show these, on the same pair of axes, as a line plot, a point plot and a bar chart.

```
r = np.arange(11)
rfac = np.concatenate((np.array([1]),np.cumprod(r[1:len(r)])))
lamb = 3.0

poissprob = np.exp(-lamb)*lamb**r/rfac

plt.plot(r, poissprob, 'b')
plt.plot(r, poissprob, '.b', markersize=10)
plt.bar(r, poissprob, color='red')
```

The image is shown as Figure 11. Note the use of NumPy's `cumprod` function.

By default, **histograms** (as opposed to bar charts) are scaled so that the total area is

the total size of the data set. However, they can, by setting `density` to `True`, be scaled so that the total area is 1, meaning that the height is the **relative frequency density**. Bin boundaries are by default set automatically, but you can override that.

Challenge 4: sample the Poisson distribution with parameter 3.0 (a) 100 times and (b) 1000 times, and show your results on density histograms on which line and point plots of the underlying distribution also appear.

```
import numpy.random as nrnd

ndata1 = 100
data1 = nrnd.poisson(3.0, ndata1)
ndata2 = 1000
data2 = nrnd.poisson(3.0, ndata2)

fig, (ax1, ax2) = plt.subplots(1,2)
ax1.plot(r, poissprob, 'b')
ax1.plot(r, poissprob, '.b', markersize=10)
ax1.hist(data1, bins=np.arange(-0.5,12.5),color='red',density=True);
ax2.plot(r, poissprob, 'b')
ax2.plot(r, poissprob, '.b', markersize=10)
ax2.hist(data2, bins=np.arange(-0.5,12.5),color='red',density=True);
```

The image is shown as Figure 12.

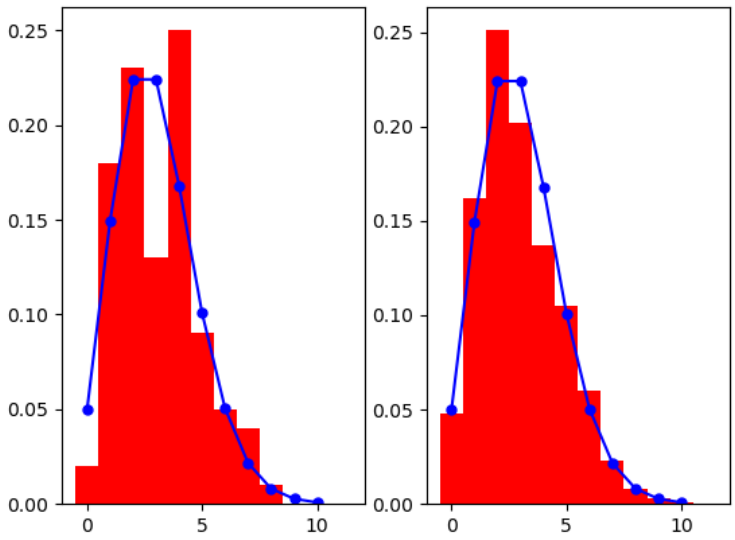

Figure 12: Density histograms of samples of a Poisson distribution; sample sizes 100 and 1000