Computational Methods in Research

Assignment 2

Exercise 1 (2.11 from book)

Binomial coefficients

The binomial coefficient $\binom{n}{k}$ is an integer equal to

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{1 \times 2 \times \ldots \times k}$$

when $k \geq 1$, or $\binom{n}{0} = 1$ when $k = 0$.

1. Using this form for the binomial coefficient, write a user-defined function `binomial(n,k)` that calculates the binomial coefficient for given n and k. Make sure your function returns the answer in the form of an integer (not a float) and gives the correct value of 1 for the case where $k = 0$. Use your function to compute $C(6,0)$ and $C(6,2)$.

2. Using your function write a program to print out the first 7 lines of "Pascal's triangle." The nth line of Pascal's triangle contains $n+1$ numbers, which are the coefficients $\binom{n}{0}$, $\binom{n}{1}$, and so on up to $\binom{n}{n}$. Thus the first few lines are

   1 1
   1 2 1
   1 3 3 1
   1 4 6 4 1

3. The probability that an unbiased coin, tossed n times, will come up heads k times is $\binom{n}{k}/2^n$. Write a program to calculate

   (i) the total probability that a coin tossed 100 times comes up heads exactly 60 times, and

   (ii) the probability that it comes up heads 60 or more times.

   (iii) the probability that it comes up heads 59 or less times. If we add the value in (ii) and the value in (iii), do you get 1?

Exercise 2 (2.12 from book)

Prime numbers

You do not need to define any function to solve this problem.

The program in Example 2.8 is not a very efficient way of calculating prime numbers: it checks each number to see if it is divisible by any number less than it. We can develop a much faster program for prime numbers by making use of the following observations:

1. A number n is prime if it has no prime factors less than n. Hence we only need to check if it is divisible by other primes.

2. If a number n is non-prime, having a factor r, then $n = rs$, where s is also a factor. If $r \geq \sqrt{n}$ then $n = rs \geq \sqrt{n}s$, which implies that $s \leq \sqrt{n}$. In other words, any non-prime must have factors, and hence also prime factors, less than or equal to $\sqrt{n}$. Thus to determine if a number is prime we have to check its prime factors only up to and including $\sqrt{n}$—if there are none then the number is prime.
3. If we find even a single prime factor less than $\sqrt{n}$ then we know that the number is non-prime, and hence there is no need to check any further—we can abandon this number and move on to something else.

Because of the facts above, we can follow this recipe to write a Python program that finds all the primes up to ten thousand.

*) Create a list to store the primes, which starts out with just the one prime number 2 in it.

*) Then for each number n from 3 to 1500 check whether the number is divisible by any of the primes in the list up to and including $\sqrt{n}$. As soon as you find a single prime factor you can stop checking the rest of them—you know n is not a prime. If you find no prime factors $\sqrt{n}$ or less then n is prime and you should add it to the list.

*) Print out the list at the end of the program.

Exercise 3 (2.13 from book)

Recursion

A useful feature of user-defined functions is *recursion*, the ability of a function to call itself. For example, consider the following definition of the factorial $n!$ of a positive integer n:

$$n! = \begin{cases} 1 & \text{if } n = 1, \\ n \times (n-1)! & \text{if } n > 1. \end{cases}$$

This constitutes a complete definition of the factorial which allows us to calculate the value of $n!$ for any positive integer. We can employ this definition directly to create a Python function for factorials, like this:

```
def factorial(n):
    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

Note how, if n is not equal to 1, the function calls itself to calculate the factorial of $n-1$. This is recursion. If we now say "`print(factorial(5))`" the computer will correctly print the answer 120.

We encountered the Catalan numbers C_n previously in Exercise 2.7 on page 46. With just a little rearrangement, the definition given there can be rewritten in the form

$$C_n = \begin{cases} 1 & \text{if } n = 0, \\ \dfrac{4n-2}{n+1} C_{n-1} & \text{if } n > 0. \end{cases}$$

Write a Python function, using recursion, that calculates C_n. Use your function to calculate and print C_{100}.

Comparing the calculation of the Catalan numbers in part (a) above with that of Exercise 2.7, we see that it's possible to do the calculation two ways, either directly or using recursion. In most cases, if a quantity can be calculated *without* recursion, then it will be faster to do so, and we normally recommend taking this route if possible. There are some calculations, however, that are essentially impossible (or at least much more difficult) without recursion. We will see some examples later in this book.

Exercise 4

For all plots below: use axes labels (font=14), tic labels (font=12), and a legend.

(a) Using different lines, plot simultaneously the functions $y_1 = -x^2$ (black dashed), $y_2 = x^2$ (blue dot-dashed), and $y_3 = x^2 \sin(1/x)$ (solid red) on the interval [-0.02,0.02]. Make sure you have enough points, so that the plot for y_3 shows a smooth curve. Start the x-tics at -0.02 and show them in increments of 0.02. Start the y-tics at -0.0004 and show them in increments of 0.0002.

(b) The function $y_1 = x^2$ is the inverse of $y_2 = \sqrt{x}$. Inverse functions are symmetric with respect to the line $y = x$. Plot y_1 and y_2 with solid lines and $y = x$ with a dashed line. The plot should show the x- and y-axes from 0 to 4. Observe the symmetry. Use different colors for each curve.

(c) Write a code to find the first ten primes, the first ten Fibonacci numbers, and the first ten perfect squares. Using different symbols for the points, plot together the set of points corresponding to the first ten primes (red circles), the first ten Fibonacci numbers (blue triangles), and the first ten perfect squares (black squares). The values of x should go from 1 to 10.

Exercise 5

(a) Plot three different functions: $y_1(t,x) = \cos(t+x)$, $y_2(t,x) = \cos(t-x)$ and $y_3(t,x) = y_1(t,x) + y_2(t,x)$. Fix the value of $x = 1.47 * \pi$ and use t from 0 to 4π. Use increments of t equal or smaller than 0.1. The positions and titles for the plots should be

- y_1 in [0.2, 0.7, 0.2, 0.2], with title 'Left'. Use facecolor='y'. Range: x-axis should go from 0 to 4π and y-axis from -2 to 2. Color of the curve = red.
- y_2 in [0.5, 0.7, 0.2, 0.2], with title 'Right'. Use facecolor='y'. Range: x-axis should go from 0 to 4π and y-axis from -2 to 2. Color of the curve = blue.
- y_3 in [0.1, 0.1, 0.7, 0.5], with title 'Interference'. y-axis label='sum'; x-axis label='time (s)'. Range: x-axis should go from 0 to 4π and y-axis from -2 to 2. Color of the curve = black.

(b) Keep everything as above, but remove "facecolor". Make an animation, using x from -4π to 4π. Use increments of x equal or smaller 0.1. Use interval=200.

(c) Make another animation, this time with just two panels, The top one should have po-

sition [0.35, 0.72, 0.2, 0.2], should contain animations for both curves y_1 and y_2, and should be called 'Separated Waves'. The bottom one should have position [0.1, 0.1, 0.7, 0.5], should contain y_3, and be called 'Interference'.

Exercise 6

Plotting experimental data

In the on-line resources you will find a file called `sunspots.txt`, which contains the observed number of sunspots on the Sun for each month since January 1749. The file contains two columns of numbers, the first being the month and the second being the sunspot number.

1. Write a program that reads in the data and makes a graph of sunspots as a function of time.
2. Modify your program to display only the first 1000 data points on the graph.
3. Modify your program further to calculate and plot the running average of the data, defined by

$$Y_k = \frac{1}{(2r+1)} \sum_{m=-r}^{r} y_{k+m} ,$$

where $r = 5$ in this case (and the y_k are the sunspot numbers). Have the program plot both the original data and the running average on the same graph, again over the range covered by the first 1000 data points.

Exercise 7

Curve plotting

Although the `plot` function is designed primarily for plotting standard xy graphs, it can be adapted for other kinds of plotting as well.

1. Make a plot of the so-called *deltoid* curve, which is defined parametrically by the equations

$$x = 2\cos\theta + \cos 2\theta, \qquad y = 2\sin\theta - \sin 2\theta,$$

where $0 \leq \theta < 2\pi$. Take a set of values of θ between zero and 2π and calculate x and y for each from the equations above, then plot y as a function of x.

2. Taking this approach a step further, one can make a polar plot $r = f(\theta)$ for some function f by calculating r for a range of values of θ and then converting r and θ to Cartesian coordinates using the standard equations $x = r\cos\theta$, $y = r\sin\theta$. Use this method to make a plot of the Galilean spiral $r = \theta^2$ for $0 \leq \theta \leq 10\pi$.

[The idea here is simply to make a plot of x vs y, where $x = r\cos(\theta)$, $y = r\sin(\theta)$ and $r = \theta^2$, with $\theta \in (0, 10\pi)$]

3. Using the same method, make a polar plot of "Fey's function"

$$r = e^{\cos\theta} - 2\cos 4\theta + \sin^5 \frac{\theta}{12}$$

in the range $0 \leq \theta \leq 24\pi$.

Exercise 8

Histogram

Generate 500 real random numbers from a Gaussian distribution with mean zero and variance 1. Use this data to make two histograms on the same figure. Each histogram should be on a different panel (axes).

The histogram on the left should have binwidth 0.2. It should be red and have black solid lines edging the bins.

The histogram on the right should have binwidth 0.5. It should be blue and have black solid lines edging the bins.

Exercise 9

Deterministic chaos and the Feigenbaum plot

One of the most famous examples of the phenomenon of chaos is the *logistic map*, defined by the equation

$$x' = rx(1-x). \tag{1}$$

For a given value of the constant r you take a value of x, say $x = \frac{1}{2}$, and you feed it into the right-hand side of this equation, which gives you a value of x'. Then you take that value and feed it back in on the right-hand side again, which gives you another value, and so forth. This is an *iterative map*. You keep doing the same operation over and over on your value of x, and one of three things happens:

1. The value settles down to a fixed number and stays there. This is called a *fixed point*. For instance, $x = 0$ is always a fixed point of the logistic map. (You put $x = 0$ on the right-hand side and you get $x' = 0$ on the left.)
2. It doesn't settle down to a single value, but it settles down into a periodic pattern, rotating around a set of values, such as say four values, repeating them in sequence over and over. This is called a *limit cycle*.
3. It goes crazy. It generates a seemingly random sequence of numbers that appear to have no rhyme or reason to them at all. This is *deterministic chaos*. "Chaos" because it really does look chaotic, and "deterministic" because even though the values look random, they are not. They are clearly entirely predictable, because they are given to you by one simple equation. The behavior is *determined*, although it may not look like it.

Write a program that calculates and displays the behavior of the logistic map. Here is what you need to do. For a given value of r, start with $x = \frac{1}{2}$, and iterate the logistic map equation a thousand times. That will give it a chance to settle down to a fixed point or limit cycle if it is going to. Then run for another thousand iterations and plot the points (r, x) on a graph where the horizontal axis is r and the vertical axis is x. Repeat the whole calculation for values of r from 2.8 to 4 in steps of 0.01, plotting the dots for all values of r on the same figure.

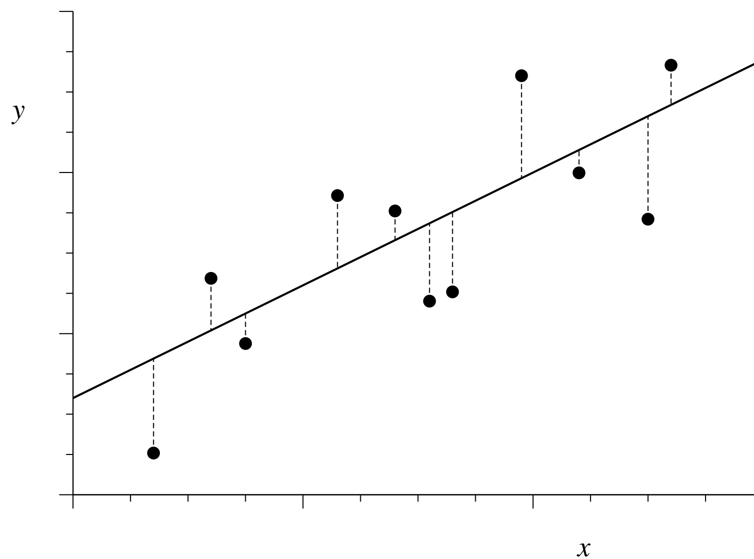
Comment: The logistic map is a very simple mathematical system, but deterministic chaos is seen in many more complex physical systems also, including especially fluid dynamics and

the weather. Because of its apparently random nature, the behavior of chaotic systems is difficult to predict and strongly affected by small perturbations in outside conditions. You've probably heard of the classic exemplar of chaos in weather systems, the *butterfly effect*, which was popularized by physicist Edward Lorenz in 1972 when he gave a lecture to the American Association for the Advancement of Science entitled, "Does the flap of a butterfly's wings in Brazil set off a tornado in Texas?" (Although arguably the first person to suggest the butterfly effect was not a physicist at all, but the science fiction writer Ray Bradbury in his famous 1952 short story *A Sound of Thunder*, in which a time traveler's careless destruction of a butterfly during a tourist trip to the Jurassic era changes the course of history.)

Exercise 10

Least-squares fitting and the photoelectric effect

It's a common situation in physics that an experiment produces data that lies roughly on a straight line, like the dots in this figure:



The solid line here represents the underlying straight-line form, which we usually don't know, and the points representing the measured data lie roughly along the line but don't fall exactly on it, typically because of measurement error.

The straight line can be represented in the familiar form $y = mx + c$ and a frequent question is what the appropriate values of the slope m and intercept c are that correspond to the measured data. Since the data don't fall perfectly on a straight line, there is no perfect answer to such a question, but we can find the straight line that gives the best compromise fit to the data. The standard technique for doing this is the *method of least squares*.

Suppose we make some guess about the parameters m and c for the straight line. We then calculate the vertical distances between the data points and that line, as represented by the short vertical lines in the figure, then we calculate the sum of the squares of those distances, which we denote χ^2. If we have N data points with coordinates (x_i, y_i), then χ^2 is given by

$$\chi^2 = \sum_{i=1}^{N} (mx_i + c - y_i)^2.$$

The least-squares fit of the straight line to the data is the straight line that minimizes this total squared distance from data to line. We find the minimum by differentiating with respect to both m and c and setting the derivatives to zero, which gives

$$m\sum_{i=1}^{N} x_i^2 + c\sum_{i=1}^{N} x_i - \sum_{i=1}^{N} x_i y_i = 0,$$
$$m\sum_{i=1}^{N} x_i + cN - \sum_{i=1}^{N} y_i = 0.$$

For convenience, let us define the following quantities:

$$E_x = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad E_y = \frac{1}{N}\sum_{i=1}^{N} y_i, \qquad E_{xx} = \frac{1}{N}\sum_{i=1}^{N} x_i^2, \qquad E_{xy} = \frac{1}{N}\sum_{i=1}^{N} x_i y_i,$$

in terms of which our equations can be written

$$mE_{xx} + cE_x = E_{xy}\,,$$
$$mE_x + c = E_y\,.$$

Solving these equations simultaneously for m and c now gives

$$m = \frac{E_{xy} - E_x E_y}{E_{xx} - E_x^2}, \qquad c = \frac{E_{xx}E_y - E_x E_{xy}}{E_{xx} - E_x^2}.$$

These are the equations for the least-squares fit of a straight line to N data points. They tell you the values of m and c for the line that best fits the given data.

1. In the on-line resources you will find a file called `millikan.txt`. The file contains two columns of numbers, giving the x and y coordinates of a set of data points. Write a program to read these data points and make a graph with one dot or circle for each point.
2. Add code to your program, before the part that makes the graph, to calculate the quantities E_x, E_y, E_{xx}, and E_{xy} defined above, and from them calculate and print out the slope m and intercept c of the best-fit line.
3. Now write code that goes through each of the data points in turn and evaluates the quantity $mx_i + c$ using the values of m and c that you calculated. Store these values in a new array or list, and then graph this new array, as a solid line, on the same plot as the original data. You should end up with a plot of the data points plus a straight line that runs through them.
4. The data in the file `millikan.txt` are taken from a historic experiment by Robert Millikan that measured the *photoelectric effect*. When light of an appropriate wavelength is shone on the surface of a metal, the photons in the light can strike conduction electrons in the metal and, sometimes, eject them from the surface into the free space above. The energy of an ejected electron is equal to the energy of the photon that struck it minus a small amount ϕ called the *work function* of the surface, which represents the energy needed to remove an electron from the surface. The energy of a photon is $h\nu$, where h is Planck's

constant and ν is the frequency of the light, and we can measure the energy of an ejected electron by measuring the voltage V that is just sufficient to stop the electron moving. Then the voltage, frequency, and work function are related by the equation

$$V = \frac{h}{e}\nu - \phi,$$

where e is the charge on the electron. This equation was first given by Albert Einstein in 1905.

The data in the file `millikan.txt` represent frequencies ν in hertz (first column) and voltages V in volts (second column) from photoelectric measurements of this kind. Using the equation above and the program you wrote, and given that the charge on the electron is 1.602×10^{-19} C, calculate from Millikan's experimental data a value for Planck's constant. Compare your value with the accepted value of the constant, which you can find in books or on-line. You should get a result within a couple of percent of the accepted value.

This calculation is essentially the same as the one that Millikan himself used to determine of the value of Planck's constant, although, lacking a computer, he fitted his straight line to the data by eye. In part for this work, Millikan was awarded the Nobel prize in physics in 1923.