COMPUTATIONAL PHYSICS – PH 354

HOMEWORK DUE ON 20TH JAN 2020

Write functions wherever possible to make your codes clean and easily readable/modifiable.

Exercise 1: Altitude of a satellite

A satellite is to be launched into a circular orbit around the Earth so that it orbits the planet once every T seconds.

a) Write a program that asks the user to enter the desired value of T and then calculates and prints out the correct altitude of the satellite above the Earth's surface in meters.

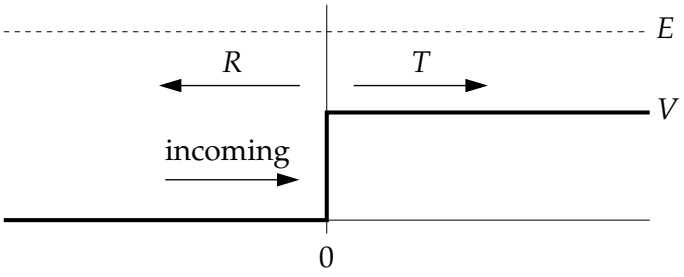
b) Use your program to calculate the altitudes of satellites that orbit the Earth once a day (so-called "geosynchronous" orbit), once every 90 minutes, and once every 45 minutes. What do you conclude from the last of these calculations?

c) Technically a geosynchronous satellite is one that orbits the Earth once per *sidereal day*, which is 23.93 hours, not 24 hours. Why is this? And how much difference will it make to the altitude of the satellite?

Exercise 2: Write a program to ask the user for the Cartesian coordinates x, y of a point in two-dimensional space, and calculate and print the corresponding polar coordinates, with the angle θ given in degrees. The calculation should be done via a function (and not directly in the main program).

Exercise 3: Quantum potential step

A well-known quantum mechanics problem involves a particle of mass m that encounters a one-dimensional potential step, like this:



The particle with initial kinetic energy E and wavevector $k_1 = \sqrt{2mE}/\hbar$ enters from the left and encounters a sudden jump in potential energy of height V at position $x = 0$. By solving the Schrödinger equation, one can show that when $E > V$ the particle may either (a) pass the step, in which case it has a lower kinetic energy of $E - V$ on the other side and a correspondingly smaller wavevector of $k_2 = \sqrt{2m(E-V)}/\hbar$, or (b) it may be reflected, keeping all of its kinetic energy and an unchanged wavevector but moving in the opposite direction. The probabilities T and R for transmission and reflection are given by

$$T = \frac{4k_1k_2}{(k_1+k_2)^2}, \qquad R = \left(\frac{k_1-k_2}{k_1+k_2}\right)^2.$$

Suppose we have a particle with mass equal to the electron mass $m = 9.11 \times 10^{-31}$ kg and energy 10 eV encountering a potential step of height 9 eV. Write a Python program to compute and print out the transmission and reflection probabilities using the formulas above. (Note: pay attention to the units and possible underflows).

Exercise 4: Planetary orbits

The orbit in space of one body around another, such as a planet around the Sun, need not be circular. In general it takes the form of an ellipse, with the body sometimes closer in and sometimes further out. If you are given the distance ℓ_1 of closest approach that a planet makes to the Sun, also called its *perihelion*, and its linear velocity v_1 at perihelion, then any other property of the orbit can be calculated from these two. Kepler's second law tells us that the distance ℓ_2 and velocity v_2 of the planet at its most distant point, or *aphelion*, satisfy $\ell_2 v_2 = \ell_1 v_1$. Write a program that asks the user to enter the distance to the Sun and velocity at perihelion, then calculates and prints the quantities ℓ_2, v_2, Orbital period T, and Orbital eccentricity e.

Test your program by having it calculate the properties of the orbits of the Earth (for which $\ell_1 = 1.4710 \times 10^{11}\,\mathrm{m}$ and $v_1 = 3.0287 \times 10^4\,\mathrm{m\,s^{-1}}$) and Halley's comet ($\ell_1 = 8.7830 \times 10^{10}\,\mathrm{m}$ and $v_1 = 5.4529 \times 10^4\,\mathrm{m\,s^{-1}}$). Among other things, you should find that the orbital period of the Earth is one year and that of Halley's comet is about 76 years.

Exercise 5: Catalan numbers

The Catalan numbers C_n are a sequence of integers 1, 1, 2, 5, 14, 42, 132... that play an important role in quantum mechanics and the theory of disordered systems. (They were central to Eugene Wigner's proof of the so-called semicircle law.) They are given by

$$C_0 = 1, \qquad C_{n+1} = \frac{4n+2}{n+2} C_n.$$

Write a program to calculate C_{100} and print the time taken to run the program.

Exercise 6: The Madelung constant

In condensed matter physics the Madelung constant gives the total electric potential felt by an atom in a solid. It depends on the charges on the other atoms nearby and their locations. Consider for instance solid sodium chloride—table salt. The sodium chloride crystal has atoms arranged on a cubic lattice, but with alternating sodium and chlorine atoms, the sodium ones having a single positive charge $+e$ and the chlorine ones a single negative charge $-e$, where e is the charge on the electron. If we label each position on the lattice by three integer coordinates (i, j, k), then the sodium atoms fall at positions where $i + j + k$ is even, and the chlorine atoms at positions where $i + j + k$ is odd.

Consider a sodium atom at the origin, $i = j = k = 0$, and let us calculate the Madelung constant. If the spacing of atoms on the lattice is a, then the distance from the origin to the atom at position (i, j, k) is

$$\sqrt{(ia)^2 + (ja)^2 + (ka)^2} = a\sqrt{i^2 + j^2 + k^2},$$

and the potential at the origin created by such an atom is

$$V(i, j, k) = \pm \frac{e}{4\pi\epsilon_0 a \sqrt{i^2 + j^2 + k^2}},$$

with ϵ_0 being the permittivity of the vacuum and the sign of the expression depending on whether $i+j+k$ is even or odd. The total potential felt by the sodium atom is then the sum of this quantity over all other atoms. Let us assume a cubic box around the sodium at the origin, with L atoms in all directions. Then

$$V_{\text{total}} = \sum_{\substack{i,j,k=-L \\ \text{not } i=j=k=0}}^{L} V(i,j,k) = \frac{e}{4\pi\epsilon_0 a} M,$$

where M is the Madelung constant, at least approximately—technically the Madelung constant is the value of M when $L \to \infty$, but one can get a good approximation just by using a large value of L.

Write a program to calculate and print the Madelung constant for sodium chloride. Use as large a value of L as you can, while still having your program run in reasonable time—say in a minute or less.

Exercise 7: The semi-empirical mass formula

In nuclear physics, the semi-empirical mass formula is a formula for calculating the approximate nuclear binding energy B of an atomic nucleus with atomic number Z and mass number A:

$$B = a_1 A - a_2 A^{2/3} - a_3 \frac{Z^2}{A^{1/3}} - a_4 \frac{(A-2Z)^2}{A} + \frac{a_5}{A^{1/2}},$$

where, in units of millions of electron volts, the constants are $a_1 = 15.67$, $a_2 = 17.23$, $a_3 = 0.75$, $a_4 = 93.2$, and

$$a_5 = \begin{cases} 0 & \text{if } A \text{ is odd,} \\ 12.0 & \text{if } A \text{ and } Z \text{ are both even,} \\ -12.0 & \text{if } A \text{ is even and } Z \text{ is odd.} \end{cases}$$

a) Write a program that takes as its input the values of A and Z, and prints out the binding energy for the corresponding atom. Use your program to find the binding energy of an atom with $A = 58$ and $Z = 28$. (Hint: The correct answer is around 490 MeV.)

b) Modify your program to print out not the total binding energy B, but the binding energy per nucleon, which is B/A.

c) Now modify your program so that it takes as input just a single value of the atomic number Z and then goes through all values of A from $A = Z$

to $A = 3Z$, to find the one that has the largest binding energy per nucleon. This is the most stable nucleus with the given atomic number. Have your program print out the value of A for this most stable nucleus and the value of the binding energy per nucleon.

d) Modify your program again so that, instead of taking Z as input, it runs through all values of Z from 1 to 100 and prints out the most stable value of A for each one. At what value of Z does the maximum binding energy per nucleon occur?

Exercise 8: 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 $k = 0$.

a) 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 $k = 0$.

b) Using your function write a program to print out the first 20 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

c) 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 (a) the total probability that a coin tossed 100 times comes up heads exactly 60 times, and (b) the probability that it comes up heads 60 or more times.

Exercise 9: Prime numbers

Write an efficient program to calculate prime numbers using the following observations:

a) 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.

b) 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.

c) 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.

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 10 000 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. You can print out the list all in one go at the end of the program, or you can print out the individual numbers as you find them.

Exercise 10: 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.

a) Catalan numbers C_n are defined by the expression given below:

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

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

b) Euclid showed that the greatest common divisor $g(m,n)$ of two nonnegative integers m and n satisfies

$$g(m,n) = \begin{cases} m & \text{if } n = 0, \\ g(n, m \bmod n) & \text{if } n > 0. \end{cases}$$

Write a Python function `g(m,n)` that employs recursion to calculate the greatest common divisor of m and n using this formula. Use your function to calculate and print the greatest common divisor of 108 and 192.

Comparing the calculation of the Catalan numbers in part (a) above with that in a previous exercise, 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.

Exercise 11: 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.

a) Write a program that reads in the data and makes a graph of sunspots as a function of time.

b) Modify your program to display only the first 1000 data points on the graph.

c) Modify your program further to calculate and plot the running average of the data, defined by

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

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 12: 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.

a) 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 $0 \le \theta < 2\pi$. 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.

b) 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 $0 \le \theta \le 10\pi$.

c) 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 $0 \le \theta \le 24\pi$.

Exercise 13: There is a file in the on-line resources called `stm.txt`, which contains a grid of values from scanning tunneling microscope measurements of the (111) surface of silicon. A scanning tunneling microscope (STM) is a device that measures the shape of a surface at the atomic level by tracking a sharp tip over the surface and measuring quantum tunneling current as a function of position. The end result is a grid of values that represent the height of the surface and the file `stm.txt` contains just such a grid of values. Write a program that reads the data contained in the file and makes a density plot of the values. Use the various options and variants you have learned about to make a picture that shows the structure of the silicon surface clearly.

Exercise 14: 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 a *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're not. They're 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's 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's 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. You can either use the `plot` function with the options `"ko"` or `"k."` to draw a graph with dots, one for each point, of you can use the `scatter` function to draw a scatter plot (which always uses dots). Repeat the whole calculation for values of r from 1 to 4 in steps of 0.01, plotting the dots for all values of r on the same figure and then finally using the function `show` once to display the complete figure.

Your program should generate a distinctive plot that looks like a tree bent over onto its side. This famous picture is called the *Feigenbaum plot*, after its discoverer Mitchell Feigenbaum, or sometimes the *figtree plot*, a play on the fact that it looks like a tree and Feigenbaum means "figtree" in German.

Give answers to the following questions:

a) For a given value of r what would a fixed point look like on the Feigenbaum plot? How about a limit cycle? And what would chaos look like?

b) Based on your plot, at what value of r does the system move from orderly behavior (fixed points or limit cycles) to chaotic behavior? This point is sometimes called the "edge of chaos."

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.)

Comment: There is another approach for computing the Feigenbaum plot, which is neater and faster, making use of Python's ability to perform arithmetic with entire arrays. You could create an array `r` with one element containing each distinct value of r you want to investigate: `[1.0, 1.01, 1.02, ... ]`. Then create another array `x` of the same size to hold the corresponding values of x, which should all be initially set to 0.5. Then an iteration of the logistic map can be performed for all values of r at once with a statement of the form `x = r*x*(1-x)`. Because of the speed with which Python can perform calculations on arrays, this method should be significantly faster than the more basic method above.

Exercise 15: The Mandelbrot set

The Mandelbrot set, named after its discoverer, the French mathematician Benoît Mandelbrot, is a *fractal*, an infinitely ramified mathematical object that contains structure within structure within structure, as deep as we care to look. The definition of the Mandelbrot set is in terms of complex numbers as follows.

Consider the equation

$$z' = z^2 + c,$$

where z is a complex number and c is a complex constant. For any given value of c this equation turns an input number z into an output number z'. The definition of the Mandelbrot set involves the repeated iteration of this equation: we take an initial starting value of z and feed it into the equation to get a new value z'. Then we take that value and feed it in again to get another value, and so forth. The Mandelbrot set is the set of points in the complex plane that satisfies the following definition:

> *For a given complex value of c, start with $z = 0$ and iterate repeatedly. If the magnitude $|z|$ of the resulting value is ever greater than 2, then the point in the complex plane at position c is* not *in the Mandelbrot set, otherwise it is in the set.*

In order to use this definition one would, in principle, have to iterate infinitely many times to prove that a point is in the Mandelbrot set, since a point is in the set only if the iteration never passes $|z| = 2$ ever. In practice, however, one usually just performs some large number of iterations, say 100, and if $|z|$ hasn't exceeded 2 by that point then we call that good enough.

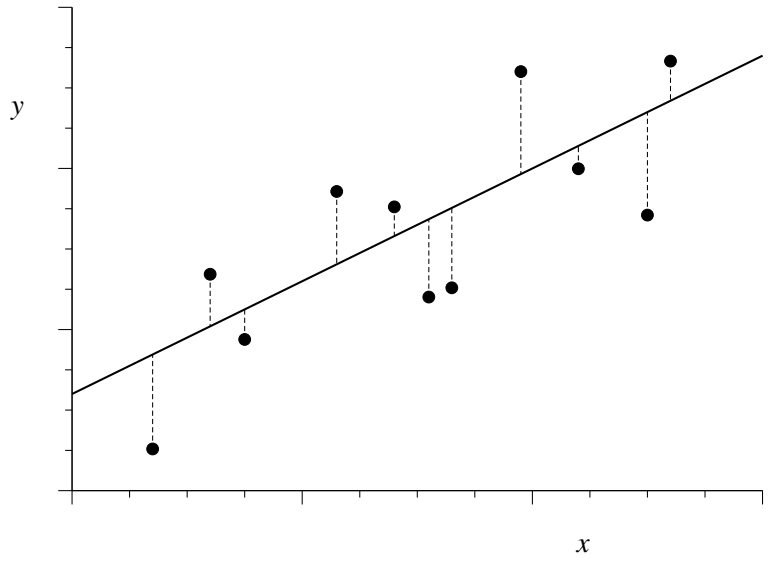
Write a program to make an image of the Mandelbrot set by performing the iteration for all values of $c = x + iy$ on an $N \times N$ grid spanning the region where $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. Make a density plot in which grid points inside the Mandelbrot set are colored black and those outside are colored white. The Mandelbrot set has a very distinctive shape that looks something like a beetle with a long snout—you'll know it when you see it.

Hint: You will probably find it useful to start off with quite a coarse grid, i.e., with a small value of N—perhaps $N = 100$—so that your program runs quickly while you are testing it. Once you are sure it is working correctly, increase the value of N to produce a final high-quality image of the shape of the set.

If you are feeling enthusiastic, here is another variant of the same exercise that can produce amazing looking pictures. Instead of coloring points just black or white, color points according to the number of iterations of the equation before $|z|$ becomes greater than 2 (or the maximum number of iterations if $|z|$ never becomes greater than 2). If you use one of the more colorful color schemes Python provides for density plots, such as the "`hot`" or "`jet`" schemes, you can make some spectacular images this way. Another interesting variant is to color according to the logarithm of the number of iterations, which helps reveal some of the finer structure outside the set.

Exercise 16: 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.

a) 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.

b) 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.

c) 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.

d) 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.