Year 2 computing

Week 1: Revisions and Set-up

Creating a file and writing in it

```
f=open("file.txt", "w") #w means write
#stuff to find value
f.write(f"blah blah{value}\n")
f.close()
```

Reading a file

```
f=open("file.txt", "r") #r means read
Text = f.read()
f.close()
print(text)
```

Ploting 2-D function z=sin(x)sin(y-1) using imshow(). Use mgrid, set origin to the lower left corner.

```
y,x = mgrid[0:pi:0.1,0:pi:0.1]
z = sin(x)*sin(y -1)
imshow(z, origin='lower')
colorbar()
```

Saving a figure

```
savefig("2d_array.png")
```

Upload data from a link

```
! wget -q link.
```

Week 2: Functions

create function that converts angles in degrees to radians

```
def convert(angle):
  return (angle*np.pi)/180
```

Split a string/ data set

```
delimiter=','
XRD = np.loadtxt('XRD_data_Mo_anode.csv', delimiter=',')
x_full = XRD[:, 0] #to get the whole of the first column
cr_full = XRD[:,1] #to get the whole of the second column
```

Defining a function

```
def cube(x):
    """
    Return the cube of x
    >>> cube(3)
    27
    """
```

```
    return x * x * x

x = 3.1
print('Cube of', x, 'is', cube(x))
'''
Of course, we don't have to use the same variable name 'x'. This is
a 'dummy variable' in the function.
'''
y = 3.1
print('Cube of', y, 'is', cube(y))

print('Cube of', 1.3, 'is', cube(1.3))


# Define a simple function ...
def print_cube(x):
    """
    Prints the cube of x
    >>> print_cube(3)
    Cube of 3 is 27
    """
    print('Cube of', x, 'is', x * x * x)

print_cube(3)

# Define a less simple function ...
def linear(x, m, c):
    '''
    This returns a linear function of a variable x
    '''
    return m*x + c

x = 1.2
y = linear(x, 2, 1)
print(y)


def T(h, g):
    '''
    Returns fall time [s] of a mass released from a height
    h [m] above the Earth's surface. g in ms^-2.
    '''
    return sqrt(2*h/g)

height = 1  # metre
print ('Time from', height, 'metre', 'is' , T(height, 9.8),'s')
```

Linear fit graph with line of best fit

<u>Week 3: Uncertainties propagation</u>

We will use a new module uncertainties, which is not standard in the colab environment. You will have to first run:

```
! pip install -q uncertainties
```

To install the module before you can import it:

```
import uncertainties as uc
import uncertainties.umath as um # for maths functions
```

Example 1: If the length of a rectangle is and its breadth is , what is its area and the error in the area? The following code snippet solves this problem in a few lines.

```
L = uc.ufloat(1.24, 0.02)
W = uc.ufloat(0.61, 0.01)
  print ('Area is:', L*W, 'm^2') # Do remember to add the units when
  printing!
```

Instead, the error progation formula gives:

```
  np.abs(1.24*0.61)*np.sqrt((0.02/1.24)**2+(0.01/0.61)**2)
```

Example 2: A reference object is long, and makes a viewing angle of . How far is it?

```
L = uc.ufloat(10.0, 0.0001)
theta = uc.ufloat(0.62,0.02)

Distance = (L/2)/um.tan(theta/2)

print ('Distance is:', Distance.nominal_value, 'm, with an error of:',
Distance.std_dev)
```

```
 Distance is: 15.609024890896208 m, with an error of: 0.537283338762715
```

First number is the nominal value, second value is the standard deviation

Week 4: Curve fitting

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

Fitting with a straight line

```
# put some test data into arrays ...
x_data = np.array([ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 ])
y_data = np.array([ 2, 104, 212, 302, 398, 507, 606, 692 ])

# define a nice, self-contained fitting routine.
def linfit(x, y):
    '''
    Takes input arrays x an y and performs a linear least squares fit.
    Returns estimated slope, error is slope,
    intercept, error in intercept.
    '''
    # do LSF using method described in PX1224 week 4 ...
    p_coeff, residuals, _, _, _ = np.polyfit(x, y, 1, full=True)
    # Note: residuals is returned as an array with one element.
    # residuals[0] is the value of this element
    n = len(x)
    D = sum(x**2) - 1./n * sum(x)**2
    x_bar = np.mean(x)
    dm_squared = 1./(n-2)*residuals[0]/D
    dc_squared = 1./(n-2)*(D/n + x_bar**2)*residuals[0]/D
    dm = np.sqrt(dm_squared)
    dc = np.sqrt(dc_squared)
    return p_coeff[0], dm, p_coeff[1], dc

# Calling the fitting function:
m, dm, c, dc = linfit(x_data, y_data)
print ('slope: ', m, '+/-', dm)
print ('intercept: ', c, '+/-', dc)
print ()

# Now for plotting purposes, we define a linear function
def linear(x, m, c):
    return m*x + c

# alternative method to define the linear function
# (effectively, it's identical to the other method
# *p means just a list of values)
def linear(x, *p):
    return p[0]*x + p[1]

# plot data
plt.plot(x_data, y_data, 'r.',label='data')
# plot fit
x_plot = np.linspace(min(x_data),max(x_data),num=100)
plt.plot(x_plot, linear(x_plot, m, c), 'b-',label='fit')
plt.xlabel('x value')
plt.ylabel('y value')
plt.title('Plot of y versus x using linfit')
plt.legend()
plt.grid ()
slope:  99.0595238095238 +/- 0.9295121405070327
```

```
intercept:  6.166666666666632 +/- 3.8884282607017973
```

The general way the curve fitting function is used is like this:

```
popt, pcov = curve_fit(f=func, xdata=x, ydata=y, p0=init_guess,
bounds=([a_min, b_min], [a_max, b_max]))
```

Here, `func` is just the name of a function that you have defined, `x` and `y` are equal length data arrays, `init_guess` is an array of initial values (somewhere near to the final values you expect). (They don't have to be called func, x, y and init_guess since those names are up to you in your program. But the `f=` parts and so on must be written as above.) The term `bounds` is to constrain the range of the parameters fitted and can usually be left out.

Example:

```
def quadratic(x, *p):
    return p[0]*x**2 + p[1]*x + p[2]

x_plot = np.linspace(min(x_data),max(x_data),num=100)

# compute a guess curve by creating an array y_guess of data points.
p_guess = np.array([10.0, 1.0, 0.0]) # initial guess
y_guess = quadratic(x_plot, *p_guess) # values come from an educated guess looking at the data.

# plot data
plt.plot(x_data, y_data, 'r.',label='data')
# plot fit
plt.plot(x_plot, y_guess, 'b-',label='guess')
plt.xlabel('x value')
plt.ylabel('y value')
plt.title('Plot of y versus x using an initial guess of quadratic()')
plt.legend()
plt.grid ()

# now do curve fit. We can use the initial guess from before
# we must pass the #name# of the function ...
popt, pcov = curve_fit(quadratic, x_data, y_data, p_guess) # do fit
```

#the error in `popt[0]` can be found as `sqrt(pcov.diagonal()[0])`

printing out the fitted coefficients and the errors for example, or plotting the data and the fitted curve.

```
print ('a: ', popt[0], '+/-', np.sqrt(pcov.diagonal()[0]))
print ('b: ', popt[1], '+/-', np.sqrt(pcov.diagonal()[1]))
print ('c: ', popt[2], '+/-', np.sqrt(pcov.diagonal()[2]))
print ()

x_plot = np.linspace(min(x_data),max(x_data),num=100)

# find fitted curve by creating an array yfit of data points ...
```

```
yfit = quadratic(x_plot, *popt)

# plot data
plt.plot(x_data, y_data, 'r.',label='data')
# plot fit
plt.plot(x_plot, yfit, 'b-',label='fit')
plt.xlabel('x value')
plt.ylabel('y value')
plt.title('Plot of y versus x using curve_fit')
plt.legend()
plt.grid ()
```

```
a:  -0.5297619047848805 +/- 0.4506307555014284
b:  102.76785714307914 +/- 3.280641440439505
c:  2.4583333333387096 +/- 4.9158036871993405
```

`# quadratic(x, *popt)` is an alternative to `quadratic(x, popt[0], popt[1], popt[2])`

Importing a csv file
The data file `XRD_data_Mo_anode.csv`:

```
! wget -q
https://raw.githubusercontent.com/PX2134/data/master/week4/XRD_data_Mo_ano
de.csv
```

```
XRD = np.loadtxt('XRD_data_Mo_anode.csv', delimiter=',')

x_full = XRD[:, 0] # for the first row
cr_full = XRD[:,1] # for the second row
```

2. Fitting gaussian shape

Week 5: Further curve fitting

In this worksheet we'll have a further look at `curve_fit()`, in particular in the presence of noisy data.

3d functions

```
# Some imports for animating.
from matplotlib.animation import FuncAnimation
from IPython import display # This is specific to IPython, upon which jupyter notebooks are built.
```

Animating:

```
# Creating an empty figure
fig = plt.figure()
ax = plt.axes()
```

```
# Initialisation of the plot element `line` as empty:
line, = ax.plot([],label='data',color='red')
# We could in general have more than one plot element.

x=np.linspace(0,2*np.pi,100)

# Setting axes so they don't move from frame to frame
ax.set_xlim(0,2*np.pi)
ax.set_ylim(-1.1,1.1)

# The anination function, which gets run for each frame, with a
# different frame number. It has to accept as input a single
# frame number, and replace the data for each frame number
def animate(frame_num):

    # This function updated the data of the line element.
    # In this example we have a sine wave, but this could
    # be anything that depends on the frame number.
    line.set_data(x,np.sin(x+frame_num/100))
    return line

# Animation function, for `frames` number of frame, with `interval` ms
# between frames.
anim=FuncAnimation(fig,animate,frames=100, interval=20)

# And display
video = anim.to_html5_video()
html = display.HTML(video)
display.display(html)
plt.close()
```

Week 6: Monte Carlo integration

```
import numpy as np
from numpy.random import random, normal, seed
import matplotlib.pyplot as plt
```

Measuring execution speed

To measure the speed of a piece of code, if you are uring IPython/Jupyter (as Colab does), the IPython magic command `%%time` is very useful:

```
%%time

x,y=2*np.random.rand(2,N)-1
r=np.sqrt(x*x+y*y)
n=np.sum(r <= 1.0)
```

```
print(4*n/N)
```

Note that the similar-sounding `%%timeit` command will run the cell several time and calculate the average. In the absence of IPython, you can also use the `datetime` module:

```
from datetime import datetime
startTime = datetime.now()
x,y=2*np.random.rand(2,N)-1
r=np.sqrt(x*x+y*y)
n=np.sum(r <= 1.0)
print(4*n/N)
print(datetime.now() - startTime)
```

```
3.1410168
 0:00:00.445764
```

Integration in 1D using Monte Carlo

```
def f(x):
    return x**2

N = int(1e3)
s = 0 # s is the sum
for i in range (N):
    x = random() # get a random number in [0,1) ...
    s += f(x)
integral = s/N
print('Approximate result for {} points: {}'.format(N,integral))
```

Week 7: Random walks

We first look at a random walk in 1-D for a single particle. We start the particle off at the origin of the x-axis and assume that it moves unit distance in each time step (the molecular speed is thus 1 unit). At the end of each time step we set the direction to left or right with equal probability, mimicking the effect of collisions. You can easily write the python code, but for flexibility of later python coding, I show here how to define a function, called `walk(steps)`, which is passed a single number steps and returns a vector of that length, which contains all the positions of the random walk.

```
def walk(steps):
    '''
    steps is the length of the 1-D random walk vector returned
    '''
    x = np.zeros(steps, int)  # initialise the array of integers
    x[0] = 0  # start at the origin
    for i in range(1, len(x)):
```

```
        rnd = random()  # rnd selected from [0,1)
        # step left or right randomly ...
        if (rnd<0.5):
            x[i] = x[i-1] + 1  # steps right one unit
        else:
            x[i] = x[i-1] - 1 # steps left one unit
    return x
```

Vectorised function

```
def walk(steps):
    '''
    steps is the length of the 1-D random walk vector returned
    Vectorised method.
    '''
    x = np.random.choice([-1, 1], size=steps) # random -1 or 1
    x = np.cumsum(x) # cumulative sum
    return x
```

Week 8: Ordinary Differential Equations

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

Imputing data via a link

```
!wget -q
https://raw.githubusercontent.com/PX2134/data/master/week8/ex0.txt
time = np.loadtxt("ex0.txt", usecols=[0]) # data[:,0] means any row but
the first colum
undecayed_nuclei = np.loadtxt("ex0.txt", usecols=[1]) # data[:,0] means
any row but the second colum
```

The `scipy` library for Python includes many functions for scientific computing and data analysis. The function `odeint` will let us solve numerically one or more *first order* ODEs using various sophisticated algorithms. It is used to find a solution to a differential equation:

```
y = scipy.integrate.odeint(func, y0, t)
```

where `t` is the sequence of time points for which to solve for , and `y` is the returned solution array of the same length. `y0` is the initial value of `y`, and `func` is the name of a function which defines the derivative of `y`.

Example: dy/dt = -lambda*y, which has the analytic solution as verified by substitution. It describes radioactive decay for example, where represents the number of undecayed nuclei at a time t.

```
npts = 1000 # Number of points
tmax = 10 # Maximum time

# setting up the time array:
t = np.linspace(0.0, tmax, npts)

# And the initial conditions:
t[0] = 0.0
y0 = 1e3
```

So, in this case using the `odeint()` function would look like this:

```
lam = 1.0
def f(y, t): # return derivative(s) of the array y
    return -lam*y

y_odeint = odeint(f, y0, t)

plt.plot(t, y_odeint,'.',label='odeint')
plt.xlabel('time')
plt.ylabel('nuclei')
plt.title('radioactive decay')
plt.legend()
```

Odient() with the analytical solution method:

```
def solution(t): # return the analytic solution
    return y0*np.exp(-lam*t)

plt.plot(t, y_odeint,'.',label='odeint')
plt.plot(t, solution(t),label='solution')
plt.xlabel('time')
plt.ylabel('nuclei')
plt.title('radioactive decay')
plt.legend()
plt.grid()
```

Euler method:

```
# use Euler's method to integrate equation for radioactive decay
# y_euler is the array for the solution y:
y_euler = np.zeros_like(t)
y_euler[0] = y0

dt = tmax/npts
for i in range(npts -1):
    y_euler[i+1] = y_euler[i] - lam*y_euler[i]*dt
```

```
plt.plot(t, y_odeint,'.',label='odeint')
plt.plot(t, y_euler, label='Euler')
plt.xlabel('time')
plt.ylabel('nuclei')
plt.title('radioactive decay')
plt.legend()
plt.grid()
```

Improving the function definition

In order to be able to use a different value of without modifying the function itself each time (that would be a very bad idea) you should modify the function definition and the way it is called like this:

```
npts = 1000 # Number of points
tmax = 10 # Maximum time

# setting up the time array:
t = np.linspace(0.0, tmax, npts)

def f(y, t, lam): # return derivative(s) of the array y
    return -lam*y

yinit = 1e3 # initial value
y_odeint = odeint(f, yinit, t, args=(2.0,))
```

The extra argument to the function needs to be passed to odeint as a “tuple” of values, i.e. a series of values in brackets separated by commas. So is equal to 2.0 in this example.

Solving several ODEs at once - look at worksheet week 8

Week 9: Further ODE solvers

Basic numerical approach

One way to solve the second order equations of motion is to break it down into two coupled first-order equations. This can be done as follows:

Given some initial positions , and initial velocity components , then we integrate the first equation to get an update on the position, and the second equation to get an update on the velocity, at the next time step later. We then repeat this for subsequent time steps. Let’s see three methods by which this can be done:

I Euler’s method

The simplest integration method is Euler’s method, which you used in Week 9 of PX1224 last year. The update scheme is (*ensure that you understand this*):

```
 r(t) = r[i], r(t + change in t) = r[1+1]



def orbit_solve (r0, v0, tmax, N, method = 'euler'):

    # set-up the time-step using tmax and N:

    N = 10000

    tmax = 10

    dt = tmax/N

    # set up an array for the output times (using tmax, time-step and
N):

    t = np.linspace(0, tmax, N)

    # set up arrays for the positions and velocities (see below)

    r = np.zeros((N,2)) # N rows (time step), 2 columns (x- and
y-components)

    v = np.zeros((N,2)) # same shape for v

    # include the initial conditions in the arrays

    r[0] = r0

    v[0] = v0

    # then do the integration for all the integration points

    for i in range (0, N-1):

        dist = np.linalg.norm(r[i])

        a = -G*M * r[i] / dist**3

        r[i+1] = r[i] + v[i]*dt

        v[i+1] = v[i] + a*dt
```

```
    # output the solution (arrays for positions and velocities) versus
time

    return t, r, v
```

II Euler-Cromer method

In the Euler-Cromer method, we update the velocity first, and then use the new velocity to update the position. i.e.,

An even better method is …

```
def orbit_solve(r0, v0, tmax, N):

    dt = tmax / N

    t = np.linspace(0, tmax, N)



    r = np.zeros((N, 2))

    v = np.zeros((N, 2))



    r[0] = r0

    v[0] = v0



    G = 1

    M = 1



    for i in range(N - 1):

        dist = np.linalg.norm(r[i])

        a = -G*M * r[i] / dist**3   # gravitational acceleration
```

```
        r[i+1] = r[i] + v[i] * dt

        v[i+1] = v[i] + a * dt



    return t, r, v
```

III Verlet method

This was popularised by Loup Verlet in 1967 (although first discovered two centuries before) for molecular dynamics simulations. It is very important nowadays for calculating such things as the interaction of large molecules (e.g. for developing theraputic drugs).

The method can be written as:

```
#Verlet method

    elif method == "verlet":

        a = acceleration(r[0])

        for i in range(N - 1):

            # position update

            r[i+1] = r[i] + v[i] * dt + 0.5 * a * dt**2



            # compute acceleration at new position

            a_new = acceleration(r[i+1])



            # velocity update

            v[i+1] = v[i] + 0.5 * (a + a_new) * dt



            # prepare for next iteration

            a = a_new
```

Strictly speaking, this is called the 'velocity Verlet' method. (It is closely related to the so-called 'leapfrog' method.)

So you need to evaluate the acceleration at the current point (time) and the next point (time). Since depends only on position in our case of orbits, this is straightforward because the next position is calculated in the first line.

The clever thing about the integrator is that it is time-reversible, and this means that the energy of the system is in principle exactly (not just approximately) conserved. This leads to huge improvements in accuracy.

Exercise 3 - Modify your function to include both the Euler-Cromer method and the Verlet method. The method keyword in the function should be used to select the integration method used in the function (using an if statement).

```
def orbit_solve(r0, v0, tmax, N, method="euler"):

    dt = tmax / N
    t = np.linspace(0, tmax, N)

    r = np.zeros((N, 2))
    v = np.zeros((N, 2))

    r[0] = r0
    v[0] = v0

    GM = 1.0

    # gravitational acceleration
    def acceleration(pos):
        dist = np.linalg.norm(pos)
        return -GM * pos / dist**3
    #Euler method
    if method == "euler":
        for i in range(N - 1):
            a = acceleration(r[i])
            r[i+1] = r[i] + v[i] * dt
            v[i+1] = v[i] + a * dt

    #Euler-cromer method
    elif method == "euler-cromer":
        for i in range(N - 1):
            a = acceleration(r[i])
            v[i+1] = v[i] + a * dt      # Update v first (symplectic)
            r[i+1] = r[i] + v[i+1] * dt

    #Verlet method
    elif method == "verlet":
        a = acceleration(r[0])
```

```
        for i in range(N - 1):
            # position update
            r[i+1] = r[i] + v[i] * dt + 0.5 * a * dt**2

            # compute acceleration at new position
            a_new = acceleration(r[i+1])

            # velocity update
            v[i+1] = v[i] + 0.5 * (a + a_new) * dt

            # prepare for next iteration
            a = a_new

    else:
        raise ValueError("Unknown method. Choose 'euler',
'euler-cromer', or 'verlet'.")

    return t, r, v
```

```
[ ]
```

Exercise 4 - Compare the Euler, Euler-Cromer and Verlet method by plotting a graph for a given number of points. Again, choose just a few orbits.

```
#initila conditions
r0 = np.array([10.0, 0.0])
v0 = np.array([0.0, np.sqrt(1/10)])

# one orbital period
T = 2 * np.pi * np.sqrt(10**3)

# simulate for two orbits
tmax = 2 * T

# number of time steps
N = 400

t_e, r_e, v_e = orbit_solve(r0, v0, tmax, N, method="euler")
t_ec, r_ec, v_ec = orbit_solve(r0, v0, tmax, N, method="euler-cromer")
t_v, r_v, v_v = orbit_solve(r0, v0, tmax, N, method="verlet")

plt.figure(figsize=(8,8))
```

```
plt.plot(r_e[:,0], r_e[:,1], color="red", label="Euler")
plt.plot(r_ec[:,0], r_ec[:,1], color="blue", label="Euler-Cromer")
plt.plot(r_v[:,0], r_v[:,1], color="green", label="Verlet")
plt.scatter([0], [0], s=40, color="black") # middle marker
plt.xlabel("x")
plt.ylabel("y")
plt.title("Orbit Comparison: Euler, Euler-Cromer, Verlet")
plt.grid()
plt.legend()
```

Week 10: Fourier Transform

```
import matplotlib.pyplot as plt
import numpy as np
from numpy.fft import fft, ifft
```

FFT

There are several FFT algorithms which are simple to use. We'll just use fft and ifft. First set up a data array to test out the method:

```
# Examine a small number of points to see what happens more easily
N=16 # the fft algorithm is fastest for N a power of 2
tau = 10.0 # time length
t = np.linspace(0, tau, N, endpoint=False) # creates N points to cover tau seconds
# Choose endpoint False so that signal would fit exactly periodically within tau
print(t)
#
f = 0.2 # frequency in Hz
x = np.cos(2*np.pi*f*t) # create x array, amplitude 1.0
plt.plot(x)
plt.plot(x, 'o') # show as points and line
# Note how the plot is exactly periodic

plt.title('Single cosine wave sampled at {} points'.format(N))
plt.xlabel('point number')
plt.ylabel('x(N)')
plt.grid()
```

```
[0.    0.625 1.25  1.875 2.5   3.125 3.75  4.375 5.    5.625 6.25  6.875
  7.5   8.125 8.75  9.375]
```