Homework 5

In [621…

```
import astropy.units as u
import astropy.constants as c
from astropy.coordinates import SkyCoord
from astropy.time import Time
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import glob
%matplotlib inline
```

In [622…

```
plt.rcParams['figure.figsize'] = (10, 10)
plt.rc('axes', labelsize=14)
plt.rc('axes', labelweight='bold')
plt.rc('axes', titlesize=16)
plt.rc('axes', titleweight='bold')
plt.rc('font', family='sans-serif')
```

Problem 1

Problem 1: The Lotka–Volterra equations The Lotka–Volterra equations are a mathematical model of predator–prey interactions between biological species. Let two variables x and y be proportional to the size of the populations of two species, traditionally called "rabbits" (the prey) and "foxes" (the predators). You could think of x and y as being the population in thousands, say, so that x = 2 means there are 2000 rabbits. Strictly the only allowed values of x and y would then be multiples of 0.001, since you can only have whole numbers of rabbits or foxes. But 0.001 is a pretty close spacing of values, so it's a decent approximation to treat x and y as continuous real numbers so long as neither gets very close to zero. In the Lotka–Volterra model the rabbits reproduce at a rate proportional to their popula- tion, but are eaten by the foxes at a rate proportional to both their own population and the population of foxes: dx dt = αx − βxy, where α and β are constants. At the same time the foxes reproduce at a rate proportional the rate at which they eat rabbits—because they need food to grow and reproduce—but also die of old age at a rate proportional to their own population: dy dt = γxy − δy, where γ and δ are also constants.

a) Write a program to solve these equations using the fourth-order Runge–Kutta method for the case α = 1, β = γ = 0.5, and δ = 2, starting from the initial condition x = y = 2. Have the program make a graph showing both x and y as a function of time on the same axes from t = 0 to t = 30. (Hint: Notice that the differential equations in this case do not depend explicitly on time t—in vector notation, the right-hand side of each equation is a function f (r) with no t dependence. You may nonetheless find it convenient to define a Python function f(r,t) including the time variable, so that your program takes the same form as other programs given in Newman's Chapter 8. You don't have to do it that way, but it can avoid some confusion. Several of the following exercises have a similar lack of explici

In [3]:

```
#Function from class:
def rk4(function,a,b,N):
    h = (b-a)/N
    tpoints = np.arange(a,b,h)
    xpoints = []
    x = 0.0
    for t in tpoints:
        xpoints.append(x)
        k1 = h*f(x,t)
        k2 = h*f(x+0.5*k1,t+0.5*h)
        k3 = h*f(x+0.5*k2,t+0.5*h)
        k4 = h*f(x+k3,t+h)
        x += (k1+2*k2+2*k3+k4)/6
    return xpoints, tpoints
```

But we have a system of equations. We need to adapt them.

for:

$\frac{dx}{dt} = f_x(x, y, t)$

$\frac{dy}{dt} = f_y(x, y, t)$

we express:

$\frac{dr}{dt} = f(r, t)$

$r(t+h) = r(t) + h\frac{dr}{dt} = r(t) + hf(r, t)$

In [4]:

```
#Defining our functions to differentiate
A = 1
B = G = 0.5
D = 2

def fx(x,t,y):
    return ((A*x)-(B*x*y))

def fy(x, t, y):
    return (G*x*y - D*y)
```

In [5]:

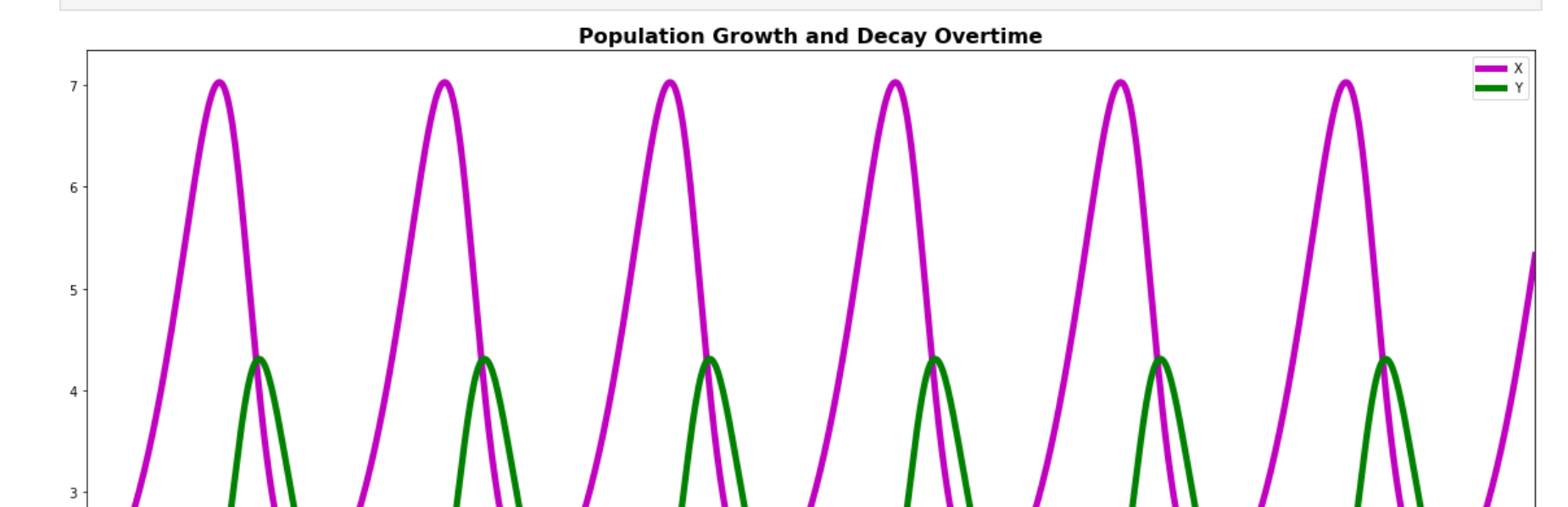
```
#Defininf the vector for the runge kata
def f(r,t):
    x = r[0]
    y = r[1]

    Fx = fx(x,t,y)
    Fy = fy(x,t,y)

    return np.array([Fx,Fy], float)
```

In [6]:

```
#initialization parameters
N=1000
a = 0
b = 30
h = (b-a)/N

tpoints = np.arange(a, b, h)
xpoints = []
ypoints = []

r = np.array([2.0,2.0], float)

for t in tpoints:
    #Add the previos solution to the list, caclulate the runge kata coeficients and compute the new r
    xpoints.append(r[0])
    ypoints.append(r[1])

    k1 = h*f(r,t)
    k2 = h*f(r+0.5*k1, t+0.5*h)
    k3 = h*f(r+0.5*k2, t+0.5*h)
    k4 = h*f(r+k3, t+h)

    r+= (1/6)*(k1+ 2*k2 + 2*k3 + k4)
```

In [7]:

```
plt.rcParams['figure.figsize'] = (20, 10)
plt.plot(tpoints, xpoints, label = 'X', c = 'm', lw = 5)
plt.plot(tpoints, ypoints, label = 'Y', c = 'g', lw = 5)
plt.xlabel('Time')
plt.title('Population Growth and Decay Overtime')
plt.xlim(0,29.95)
plt.legend();
```



b) Describe

This is a population of predators and prey animals. When there are too many predators, the prey population shrinks and this lack of food then shrinks the predator population. With less predators the prey population can increase, and the predator population then increases due to the larger food supply. The cycle repeats and we can see this hapening in the graph.

Problem 2

Problem 2: Vibration in a one-dimensional system Let's study the motion of a system of N identical masses (in zero gravity) joined by identical linear springs like this: 1 As shown in Newman's Example 6.2, the horizontal displacements ξi of masses i = 1 . . . N satisfy equations of motion m d2ξ1 dt2 = k(ξ2 − ξ1) + F1, m d2ξi dt2 = k(ξi+1 − ξi) + k(ξi−1 − ξi) + Fi , m d2ξN dt2 = k(ξN−1 − ξN) + FN. where m is the mass, k is the spring constant, and Fi is the external force on mass i. Example 6.2 shows how these equations could be solved by guessing a form for the solution and using a matrix method. Here we'll solve them more directly. Write a program to solve for the motion of the masses using the fourth-order Runge–Kutta method, where m = 1 and k = 6, and the driving forces are all zero except for F1 = cos ωt with ω = 2. Plot your solutions for the displacements ξi of all the masses as a function of time from t = 0 to t = 20 on the same plot. Write your program to work with general N, but test it out for small values—N = 5 is a reasonable choice. You will need first of all to convert the N second-order equations of motion into 2N first- order equations. Then combine all of the dependent variables in those equations into a single large vector r to which you can apply the Runge–Kutta method in the standard

I dont like the squiglies so I'm using xs:

$m\frac{d^2x_1}{dt^2} = k(x_2 - x_1) + F_1$

$m\frac{d^2x_i}{dt^2} = k(x_{i+1} - x_i) + k(x_{i-1} - x_i) + F_i$

$m\frac{d^2x_N}{dt^2} = k(x_{N-1} - x_N) + F_N$

We need to convert the two second order equations each to two seperate first order equations to solve.

for the ith mass:

$m\frac{d^2x_i}{dt^2} = k(x_{i+1} - x_i) + k(x_{i-1} - x_i) + F_i$

$\frac{d^2x_i}{dt^2} = \frac{1}{m}(k(x_{i+1} - x_i) + k(x_{i-1} - x_i) + F_i)$

Generally, we have the equation form:

$\frac{d^2x_i}{dt^2} = f(x_i, \frac{dx_i}{dt}, t)$

but our equation instead needs to be adapted to the form:

$\frac{d^2x_i}{dt^2} = f(x_i, x_{i+1}, x_{i-1}, t)$

We can try following example 8.6:

let $\frac{dx_i}{dt} = \omega$

New system of equations:

$\frac{dx_i}{dt} = \omega$

$\frac{d\omega}{dt} = \frac{1}{m}(k(x_{i+1} - x_i) + k(x_{i-1} - x_i) + F_i)$

Similarly for the Nth particle:

$\frac{dx_N}{dt} = \omega$

$\frac{d\omega}{dt} = \frac{1}{m}(k(x_{N-1} - x_N) + F_N)$

In [14]:

```
#Create function for the F coefficients
def Fi(i, t):
    if i == 1:
        return np.cos(w*t)
    else:
        return 0
# compute the function for any mass i for a number of masses Nmass
def f(r,t, i, Nmass, xnext, xlast):
    xi = r[0]
    omega = r[1]
    ftheta = omega

    F = Fi(i, t)
    if i == Nmass:
        fomega = (1/m)*(k*(xlast-xi)+F)
    elif i ==1:
        fomega = (1/m)*(k*(xnext-xi)+F)
    else:
        fomega = (1/m)*(k*(xnext-xi)+k*(xlast-xi)+F)

    return np.array([ftheta, fomega], float)
#Compute the runge kata approximation
def rk(a, b, h, f, i, Nmass, xnext, xlast):
    tpoints = np.arange(a, b, h)
    xpoints = []
    ypoints = []
    for t in tpoints:
        xpoints.append(r[0])
        ypoints.append(r[1])

        k1 = h*f(r,t, i, Nmass, xnext, xlast)
        k2 = h*f(r+0.5*k1, t+0.5*h, i, Nmass, xnext, xlast)
        k3 = h*f(r+0.5*k2, t+0.5*h, i, Nmass, xnext, xlast)
        k4 = h*f(r+k3, t+h, i, Nmass, xnext, xlast)

        r+= (1/6)*(k1+ 2*k2 + 2*k3 + k4)
    return xpoints
```

This problem is not clear at all about what it expects for the initial positions of the masses. There are 2 possibilities.

1. The masses all begin at the origin (all have an initial position of 0)

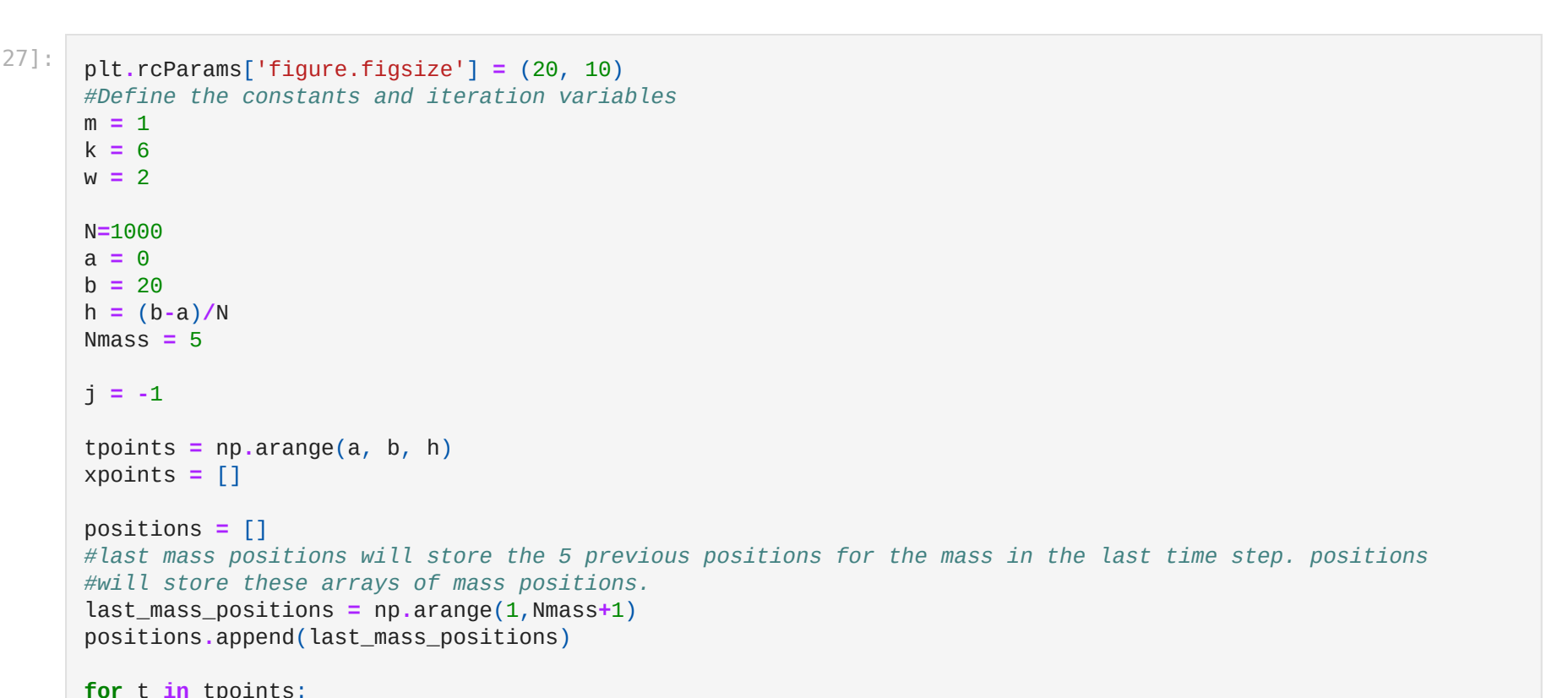
This seems to be what the problem is leaning towards, but it is not physically possible and not very intuitive. I will first plot this situation.

1. All the masses start at some arbitrary distance from eachother.

This is a case I think makes much more sense. I will plot this second.

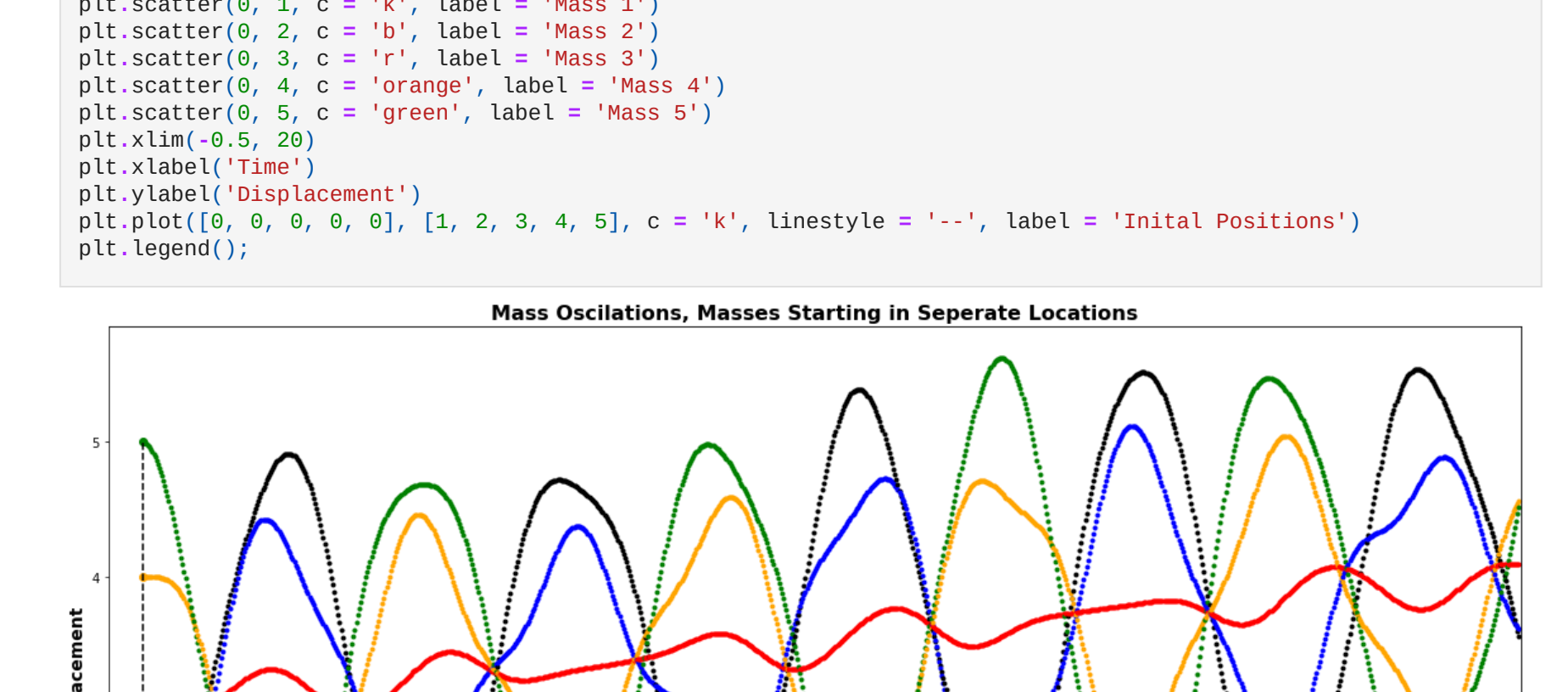
In [33]:

```
plt.rcParams['figure.figsize'] = (20, 10)
#Define the constants and iteration variables
m = 1
k = 6
w = 2

N=1000
a = 0
b = 20
h = (b-a)/N
Nmass = 5


j = -1

tpoints = np.arange(a, b, h)

xpoints = []
positions = []
#last mass positions will store the 5 previous positions for the mass in the last time step. positions
#will store these arrays of mass positions.
last_mass_positions = np.zeros(5)
positions.append(last_mass_positions)

for t in tpoints:
    j+=1
    mass_pos = []
    for i in range(1, Nmass+1):
        #for the first iteration:
        if j ==0:
            if i == 1:
                xlast = 0
                xnext = last_mass_positions[i]
            elif i == Nmass:
                xlast = last_mass_positions[i-2]
                xnext = 0
            else:
                xlast = last_mass_positions[i]
                xnext = last_mass_positions[i-2]

            r = np.array([last_mass_positions[i-1],0.0], float)
        else:
            #for all other iterations
            if i == 1:
                xlast = 0
                xnext = last_mass_positions[i][0]
            elif i == Nmass:
                xlast = last_mass_positions[i-2][0]
                xnext = 0
            else:
                xlast = last_mass_positions[i][0]
                xnext = last_mass_positions[i-2][0]

            r = last_mass_positions[i-1]

        #add the mass positions for the current mass, compute the runge kata coeffiecients to get the next r va
        mass_pos.append(r)
        k1 = h*f(r,t, i, Nmass, xnext, xlast)
        k2 = h*f(r+0.5*k1, t+0.5*h, i, Nmass, xnext, xlast)
        k3 = h*f(r+0.5*k2, t+0.5*h, i, Nmass, xnext, xlast)
        k4 = h*f(r+k3, t+h, i, Nmass, xnext, xlast)

        r+= (1/6)*(k1+ 2*k2 + 2*k3 + k4)

    #Grab the 5 mass positions and plot them
    last_mass_positions = mass_pos
    positions.append(mass_pos)

    b1 = mass_pos[0]
    b2 = mass_pos[1]
    b3 = mass_pos[2]
    b4 = mass_pos[3]
    b5 = mass_pos[4]

    plt.plot(t, b1[0], marker = '.', c = 'k')
    plt.plot(t, b2[0], marker = '.', c = 'b')
    plt.plot(t, b3[0], marker = '.', c = 'r')
    plt.plot(t, b4[0], marker = '.', c = 'orange')
    plt.plot(t, b5[0], marker = '.', c = 'green')

plt.title('Mass Oscilations, All Masses Starting at Origin')
plt.scatter(0, 0, c = 'k', label = 'Mass 1')
plt.scatter(0, 0, c = 'b', label = 'Mass 2')
plt.scatter(0, 0, c = 'r', label = 'Mass 3')
plt.scatter(0, 0, c = 'orange', label = 'Mass 4')
plt.scatter(0, 0, c = 'green', label = 'Mass 5')
plt.xlim(-0.5, 20)
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.scatter( 0, 0, c = 'c', label = 'Inital Positions', s = 200)
plt.legend();
```



In [27]:

```
plt.rcParams['figure.figsize'] = (20, 10)
#Define the constants and iteration variables
m = 1
k = 6
w = 2

N=1000
a = 0
b = 20
h = (b-a)/N
Nmass = 5

j = -1

tpoints = np.arange(a, b, h)
xpoints = []

positions = []
#last mass positions will store the 5 previous positions for the mass in the last time step. positions
#will store these arrays of mass positions.
last_mass_positions = np.arange(1,Nmass+1)
positions.append(last_mass_positions)

for t in tpoints:
    j+=1
    mass_pos = []
    for i in range(1, Nmass+1):
        if j ==0:
            #for the first iteration:
            if i == 1:
                xlast = 0
                xnext = last_mass_positions[i]
            elif i == Nmass:
                xlast = last_mass_positions[i-2]
                xnext = 0
            else:
                xlast = last_mass_positions[i]
                xnext = last_mass_positions[i-2]

            r = np.array([last_mass_positions[i-1],0.0], float)
        else:
            #for all other iterations
            if i == 1:
                xlast = 0
                xnext = last_mass_positions[i][0]
            elif i == Nmass:
                xlast = last_mass_positions[i-2][0]
                xnext = 0
            else:
                xlast = last_mass_positions[i][0]
                xnext = last_mass_positions[i-2][0]

            r = last_mass_positions[i-1]

        mass_pos.append(r)

        k1 = h*f(r,t, i, Nmass, xnext, xlast)
        k2 = h*f(r+0.5*k1, t+0.5*h, i, Nmass, xnext, xlast)
        k3 = h*f(r+0.5*k2, t+0.5*h, i, Nmass, xnext, xlast)
        k4 = h*f(r+k3, t+h, i, Nmass, xnext, xlast)

        r+= (1/6)*(k1+ 2*k2 + 2*k3 + k4)
    #Grab the 5 mass positions and plot them
    last_mass_positions = mass_pos
    positions.append(mass_pos)

    #Wow the rest of this code is structured so strangely. Why did Kelcey write it like this?
    #Excellent question. For some reason if I were to prit the object b1 here, it would print the
    #correct value. However, if I were to append it to a list defined outside of the time loop I would
    #only get the very last value for b1 and no other values. I spent hours messing with this in an attempt
    #to fix the problem and ultimately gave up and plotted it like this and now here we are. If you were to pri
    # out the variable "positions" you would see what I mean.

    #I also know that TECHNICALLY plotting it like this makes it non-general for N masses but the problem
    #only specifies that the code that ends here at line 61 be generalized so I would argue that this is fine.

    b1 = mass_pos[0]
    b2 = mass_pos[1]
    b3 = mass_pos[2]
    b4 = mass_pos[3]
    b5 = mass_pos[4]

    plt.plot(t, b1[0], marker = '.', c = 'k')
    plt.plot(t, b2[0], marker = '.', c = 'b')
    plt.plot(t, b3[0], marker = '.', c = 'r')
    plt.plot(t, b4[0], marker = '.', c = 'orange')
    plt.plot(t, b5[0], marker = '.', c = 'green')

plt.title('Mass Oscilations, Masses Starting in Seperate Locations')
plt.scatter(0, 1, c = 'k', label = 'Mass 1')
plt.scatter(0, 2, c = 'b', label = 'Mass 2')
plt.scatter(0, 3, c = 'r', label = 'Mass 3')
plt.scatter(0, 4, c = 'orange', label = 'Mass 4')
plt.scatter(0, 5, c = 'green', label = 'Mass 5')
plt.xlim(-0.5, 20)
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.plot([0, 0, 0, 0, 0], [1, 2, 3, 4, 5], c = 'k', linestyle = '--', label = 'Inital Positions')
plt.legend();
```



Problem 3

Problem 3: Oscillating chemical reactions The Belousov–Zhabotinsky reaction is a chemical oscillator, a cocktail of chemicals which, when heated, undergoes a series of reactions that cause the chemical concentrations in the mixture to oscillate between two extremes. You can add an indicator dye to the reaction which changes color depending on the concentrations and watch the mixture switch back and forth between two different colors for as long as you go on heating the mixture. Physicist Ilya Prigogine formulated a mathematical model of this type of chemical oscilla- tor, which he called the "Brusselator" after his home town of Brussels. The equations for the Brusselator are dx dt = 1 − (b + 1)x + ax2y, dy dt = bx − ax2y. Here x and y represent concentrations of chemicals and a and b are positive constants. Write a program to solve these equations for the case a = 1, b = 3 with initial conditions x = y = 0, to an accuracy of at least δ = 10−10 per unit time in both x and y, using the adaptive Bulirsch–Stoer method described in Newman's Section 8.5.6. Calculate a solution from t = 0 to t = 20, initially using a single time interval of size H = 20. Allow a maximum of n = 8 modified midpoint steps in an interval before you divide in half and try again. Make a plot of your solutions for x and y as a function of time, both on the same graph, and have your program add dots to the curves to show where the boundaries of the time intervals lie. You should find that the points are significantly closer together in parts of the solution where the variables are changing rapidly. 2 Hint: The simplest way to do this calculation is to make use of recursion, the ability of a Python function to call itself. Write a user-defined function called, say, step(r,t,H) that takes as arguments the vector r = (x, y) at a starting time t and an interval length H, and returns the new value of r at time t + H. This function should perform the modified midpoint/Richardson extrapolation calculation described in Newman's Section 8.5.5 until either the calculation con- verges to the required accuracy or you reach the maximum number n = 8 of modified midpoint steps. If it fails to converge in eight steps, have your function call itself, twice, to calculate sep- arately the solution for the first then the second half of the interval from t to t + H, something like this: r1 = step(r,t,H/2) r2 = step(r1,t+H/2,H/2) (Then these functions can call themselves, and so forth, subdividing the interval as many times as necessary to reach the required accuracy

$\frac{dx}{dt} = 1 - (B+1)x + Ax^2y$

$\frac{dy}{dt} = Bx - Ax^2y$

Just a check: the runge-kata method

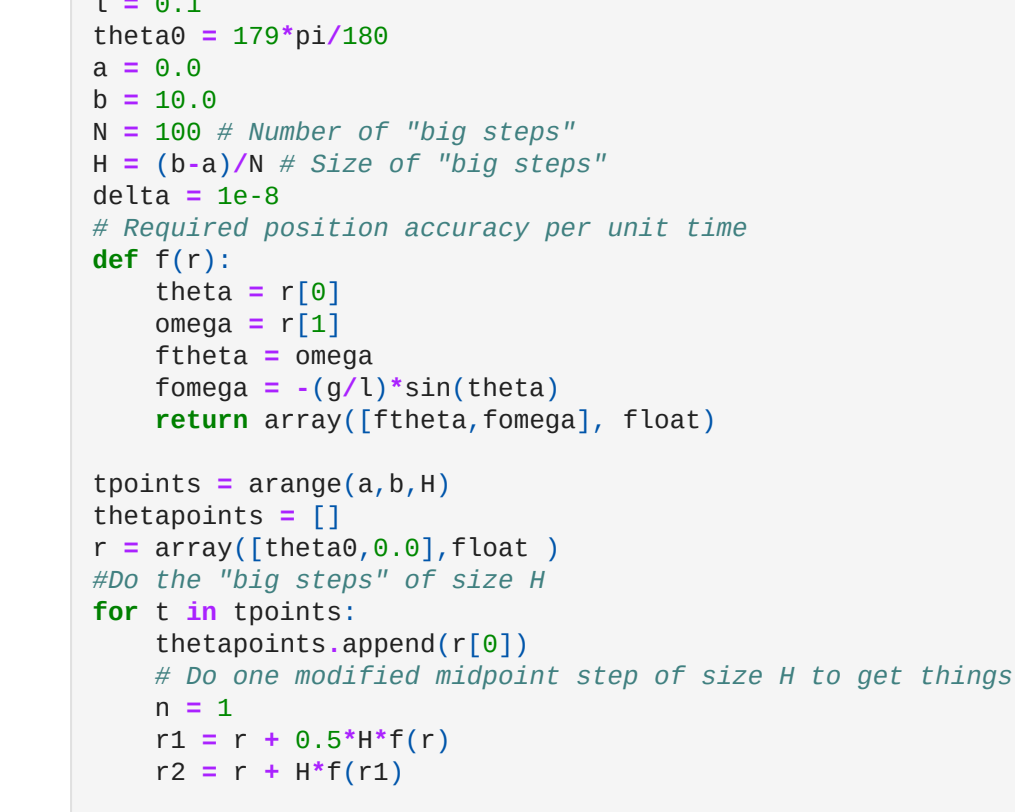
- we can get a quick look at about what we want our solution to look like by quickly applying a runge-kata:

In [347…

```
#just modifying what we had for 1 we can make this a quick and simple application:
A = 1
B = 3

def fx(x,t,y):
    return (1 - (B+1)*x + A*(x**2)*y)

def fy(x, t, y):
    return (B*x-A*(x**2)*y)

def f(r,t):
    x = r[0]
    y = r[1]

    Fx = fx(x,t,y)
    Fy = fy(x,t,y)

    return np.array([Fx,Fy], float)

N=1000
a = 0
b = 20
h = (b-a)/N

x = y = 0.0

tpoints = np.arange(a, b, h)
xpoints = []
ypoints = []

r = np.array([x,y], float)

for t in tpoints:
    xpoints.append(r[0])
    ypoints.append(r[1])

    k1 = h*f(r,t)
    k2 = h*f(r+0.5*k1, t+0.5*h)
    k3 = h*f(r+0.5*k2, t+0.5*h)
    k4 = h*f(r+k3, t+h)

    r+= (1/6)*(k1+ 2*k2 + 2*k3 + k4)
```

In [348…

```
plt.rcParams['figure.figsize'] = (10, 5)
plt.plot(tpoints, xpoints, label = 'X')
plt.plot(tpoints, ypoints, label = 'Y')
plt.title('Solutions Derrived Using RK Method: \n Solution Check')
plt.legend(fontsize = 'xx-large')
plt.xlabel('Time');
```

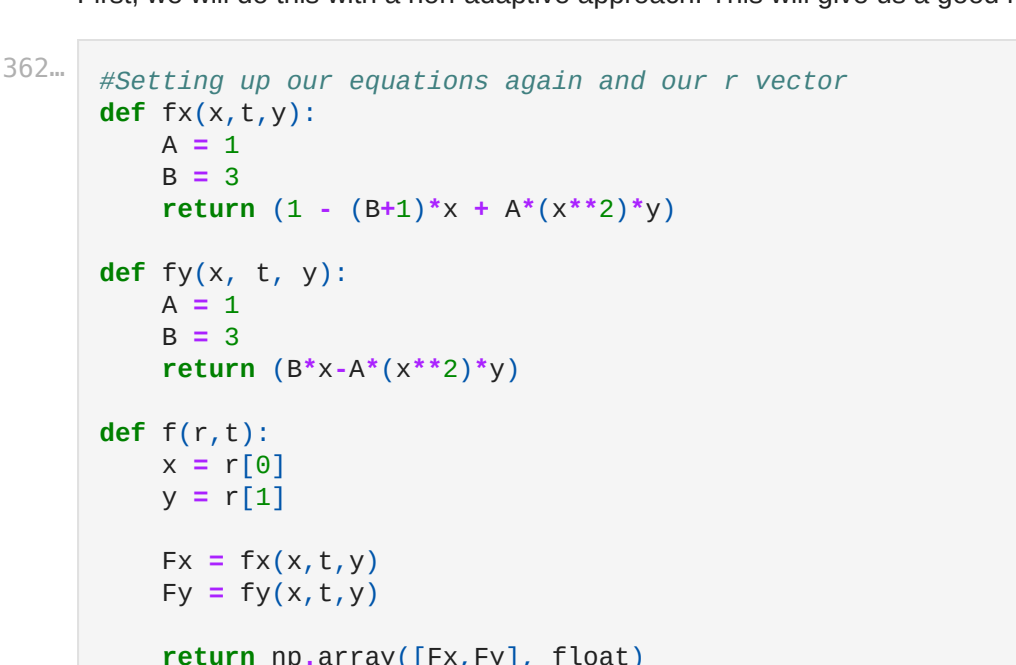


Next: the Bulrich-Stoer method:

- we have an example in the book we can reference. Example 8.7 for the non-linear pendulum reads:

In [255…

```
from math import sin,pi
from numpy import empty,array,arange
from pylab import plot,show


g = 9.81
l = 0.1
theta0 = 179*pi/180
a = 0.0
b = 10.0
N = 100 # Number of "big steps"
H = (b-a)/N # Size of "big steps"
delta = 1e-8
# Required position accuracy per unit time
def f(r):
    theta = r[0]
    omega = r[1]
    ftheta = omega
    fomega = -(g/l)*sin(theta)
    return array([ftheta,fomega],float)

tpoints = arange(a,b,H)
thetapoints = []
r = array([theta0,0.0],float )
#Do the "big steps" of size H
for t in tpoints:
    thetapoints.append(r[0])
    # Do one modified midpoint step of size H to get things started
    n = 1
    r1 = r + 0.5*H*f(r)
    r2 = r + H*f(r1)

    #The array R1 stores the first row of the
    #extrapolation table, which contains only the single
    # modified midpoint estimate of the solution at the
    # end of the interval
    R1 = empty([1,2] ,float)
    R1[0] = 0.5*(r1 + r2 + 0.5*H*f(r2))

    # Now increase n until the required accuracy is reached
    error = 2*H*delta
    while error>H*delta:
        n += 1
        h = H/n
        # Modified midpoint method
        r1 = r + 0.5*h*f(r)
        r2 = r + h*f(r1)
        for i in range(n-1):
            r1 += h*f(r2)
            r2 += h*f (r1)
        # Calculate extrapolation estimates. Arrays R1 and R2
        # hold the two most recent lines of the table
        R2 = R1
        R1 = empty([n,2] ,float)
        R1[0] = 0.5*(r1 + r2 + 0.5*h*f(r2))
        for m in range(1,n):
            epsilon= (R1[m-1]-R2[m-1])/((n/(n-1))**(2*m)-1)
            R1[m] = R1[m-1] + epsilon
        error= abs(epsilon[0])
    #Set r equal to the most accurate estimate we have
    # before moving on to the next big step
    r = R1 [n-1]
#Plot the results
plot(tpoints,thetapoints)
plot(tpoints,thetapoints,"b.")
show()
```



We are essentially accomplishing the same thing, but with an addaptive H and different sets of equations.

First, we will do this with a non-adaptive approach. This will give us a good handle on the structure of the code befoer we adapt it.

In [362…

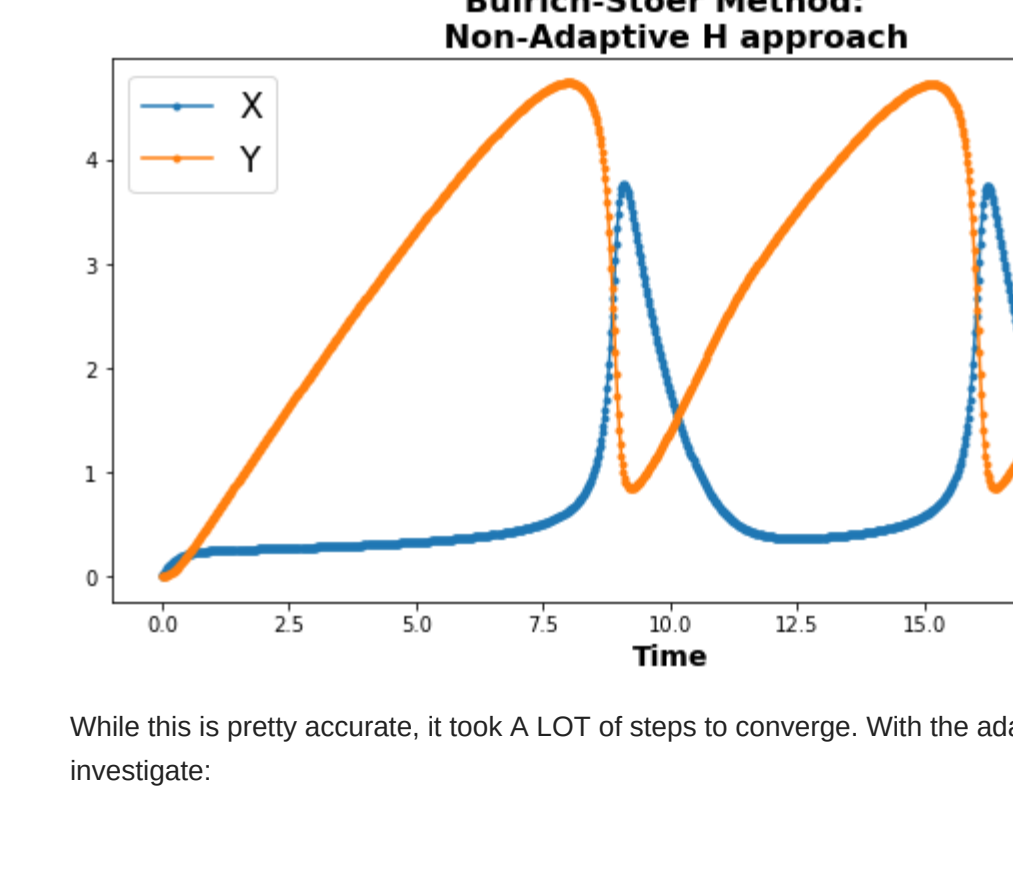
```
#Setting up our equations again and our r vector
def fx(x,y):
    A = 1
    B = 3
    return (1 - (B+1)*x + A*(x**2)*y)

def fy(x, y):
    A = 1
    B = 3
    return (B*x-A*(x**2)*y)

def f(r,t):
    x = r[0]
    y = r[1]

    Fx = fx(x,y)
    Fy = fy(x,y)

    return np.array([Fx,Fy], float)
```

In [385…

```
def adaptive_bul_st(f):

    N = 1000
    a = 0
    b = 20
    H = (b-a)/N

    x = y = 0.0

    tpoints = np.arange(a, b, H)
    xpoints = []
    ypoints = []

    r = np.array([x,y], float)

    for t in tpoints:
        xpoints.append(r[0])
        ypoints.append(r[1])
        #from example
        n = 1
        r1 = r + 0.5*H*f(r, t)
        r2 = r + H*f(r1, t)

        R1 = empty([1,2] ,float)
        R1[0] = 0.5*(r1 + r2 + 0.5*H*f(r2, t))

        error = 2*H*delta
        while error>H*delta:
            n += 1
            h = H/n

            r1 = r + 0.5*h*f(r, t)
            r2 = r + h*f(r1, t)
            for i in range(n-1):
                r1 += h*f(r2, t)
                r2 += h*f (r1, t)

            R2 = R1
            R1 = empty([n,2] ,float)
            R1[0] = 0.5*(r1 + r2 + 0.5*h*f(r2, t))

            for m in range(1,n):
                epsilon= (R1[m-1]-R2[m-1])/((n/(n-1))**(2*m)-1)
                R1[m] = R1[m-1] + epsilon
            error= abs(epsilon[0])

        r = R1 [n-1]

    plt.plot(tpoints,xpoints, marker = '.', label = 'X')
    plt.plot(tpoints,ypoints, marker = '.', label = 'Y')
    plt.xlabel('Time')
    plt.legend(fontsize = 'xx-large')
    plt.title('Bulrich-Stoer Method: \n Non-Adaptive H approach');
```

In [386…

```
adaptive_bul_st(f)
```



While this is pretty accurate, it took A LOT of steps to converge. With the adaptive H, will we see an improved computation time? Let us investigate:

In [692…

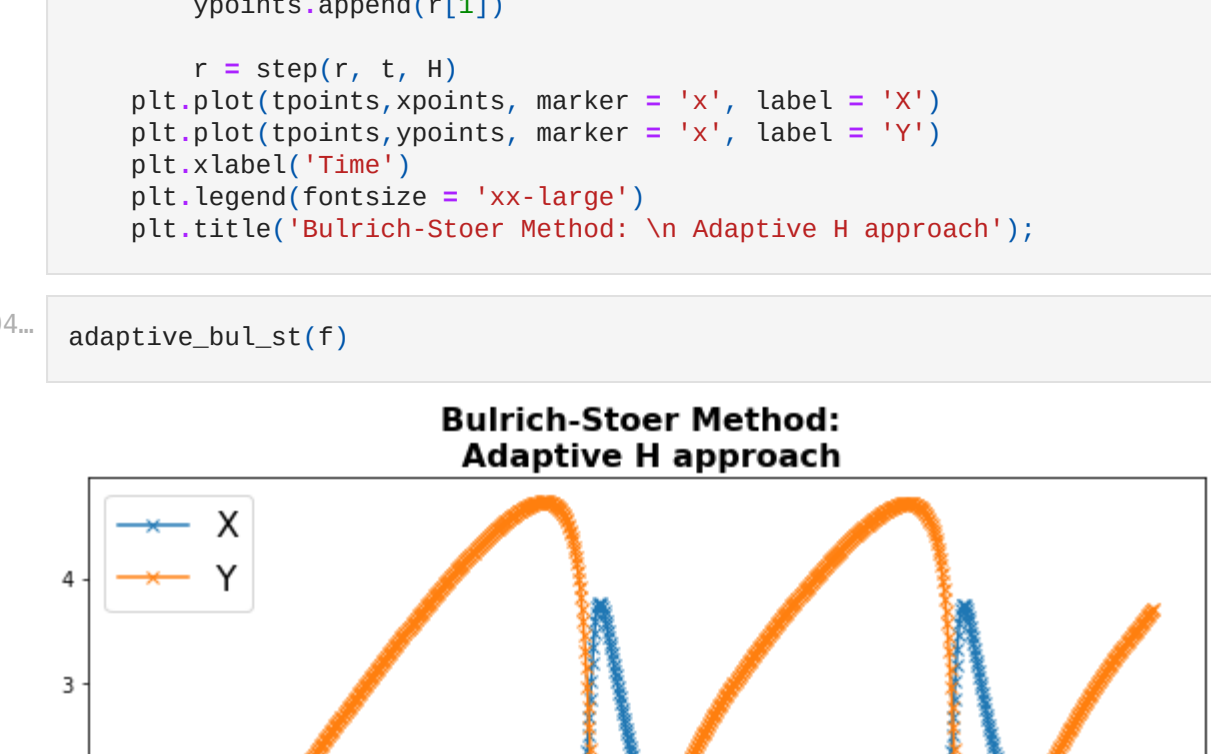
```
def step(r,t,H):
    H0 = .020

    delta = 1e-10
    nmax = 8

    n = 1

    h = H0/n

    r1 = r + 0.5*h*f(r, t)
    r2 = r + h*f(r1, t)

    R1 = empty([1,2] ,float)
    R1[0] = 0.5*(r1 + r2 + 0.5*h*f(r2, t))

    #error = 2*H*delta
    error = 1
    while error>H*delta:

        n+=1
        h = H0/n

        while n<nmax:
            r1 = r + 0.5*h*f(r, t)
            r2 = r + h*f(r1, t)

            for i in range(n-1):
                r1 += h*f(r2, t)
                r2 += h*f(r1, t)

            rvals = [r1[0], r1[1], r2[0], r2[1]]
            #add a check to get rid of blown up values
            if np.max(rvals) > 1e4:
                break

            R2 = R1
            R1 = empty([n,2] ,float)
            R1[0] = 0.5*(r1 + r2 + 0.5*h*f(r2, t))
            for m in range(1,n):
                epsilon= (R1[m-1]-R2[m-1])/((n/(n-1))**(2*m)-1)
                R1[m] = R1[m-1] + epsilon
            error= abs(epsilon[0])
            if error<delta:
                return R1[n-1]
    #recursion if while loop breaks
    r1 = step(r, t, H0/2)
    r2 = step(r1, t+H0/2, H0/2)
    return r2
```

In [693…

```
def adaptive_bul_st(f):
    #define initial parameters
    N = 1000
    a = 0
    b = 20
    h = (b-a)/N

    x = y = 0.0

    tpoints = np.arange(a, b, h)

    xpoints = []
    ypoints = []

    r = np.array([x,y], float)

    for t in tpoints:
        xpoints.append(r[0])
        ypoints.append(r[1])

        r = step(r, t, H)
    plt.plot(tpoints,xpoints, marker = 'x', label = 'X')
    plt.plot(tpoints,ypoints, marker = 'x', label = 'Y')
    plt.xlabel('Time')
    plt.legend(fontsize = 'xx-large')
    plt.title('Bulirsch-Stoer Method: \n Adaptive H approach');
```

In [694…

```
adaptive_bul_st(f)
```



Problem 4

Problem 4: Consider the following simple model of an electronic capacitor, consisting of two flat metal plates enclosed in a square metal box:10 cm 6 cm +1 V −1 V 0 V 2 cm 6 cm 2 cm For simplicity let us model the system in two dimensions. Using any of the methods we have studied, write a program to solve Laplace's equation and calculate the electrostatic potential in the box on a grid of 100 × 100 points, where the walls of the box are at voltage zero and the two plates (which are of negligible thickness) are at voltages ±1 V as shown. Have your program calculate the value of the potential at each grid point to a precision of 10−6 volts and then make a density plot of the result. Hint: Notice that the capacitor plates are at fixed voltage, so they are part of the boundary condition in this case: the capacitor plates behave the same way as the walls of the box, with potentials that are fixed at a certain value and cannot change

Laplace's equations:

$$E = -\nabla\phi$$

$$\nabla^2\phi = 0$$

- square is 10cm x 10cm = 0.1m x 0.1m
- +1 charge at 2cm, 6cm long
- -1 charge at 8cm, 6cm long
- At all sides, $V = \phi = 0$

We need to:

- solve the laplace equations
- calculate the electrostatic potention on a 100x100 grid
- find potential at each grid point to 10^{-6} precision
- make density plot

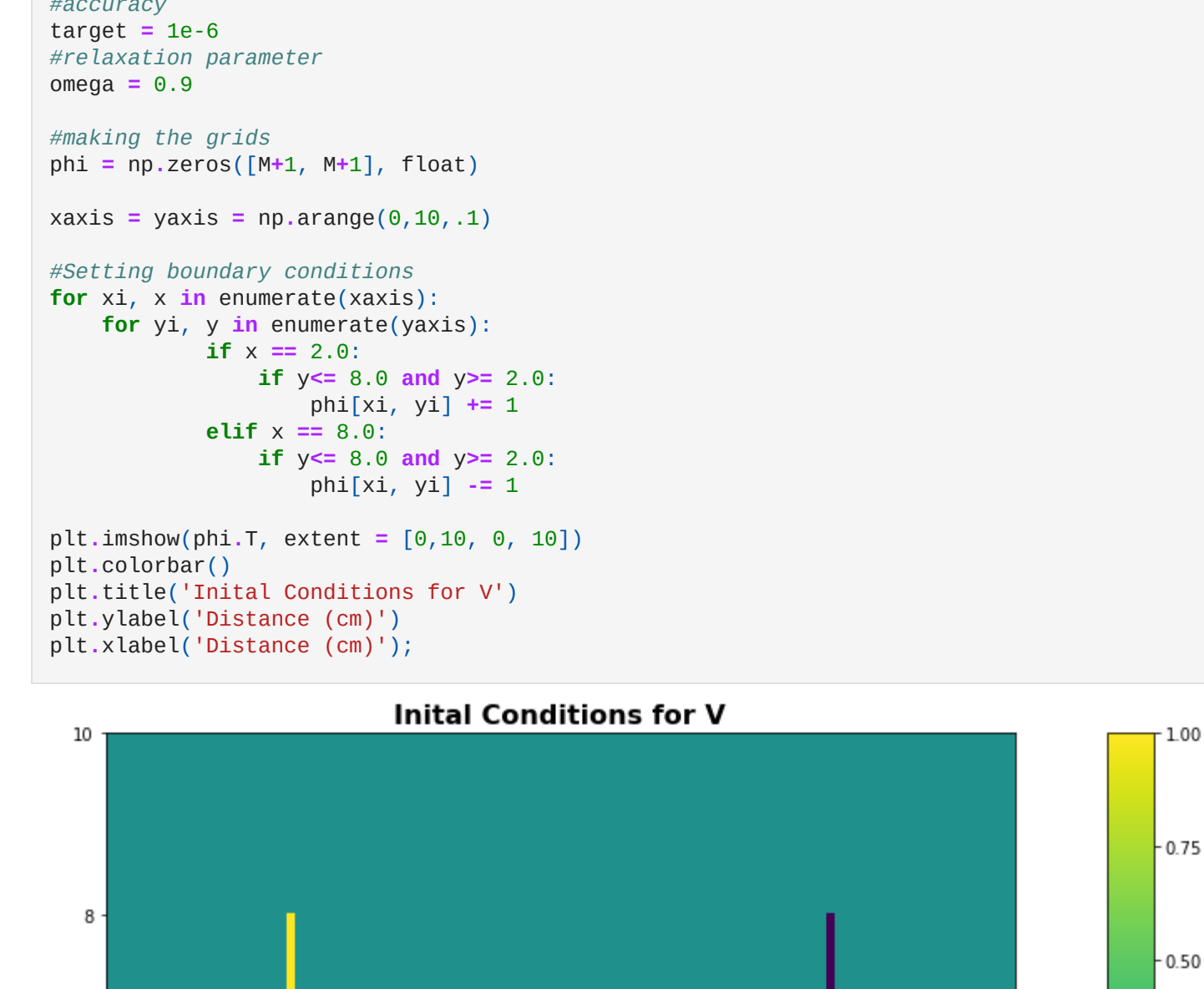
2-D laplace eqns:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0$$

Using relaxation method, overrelaxed:

In [111…

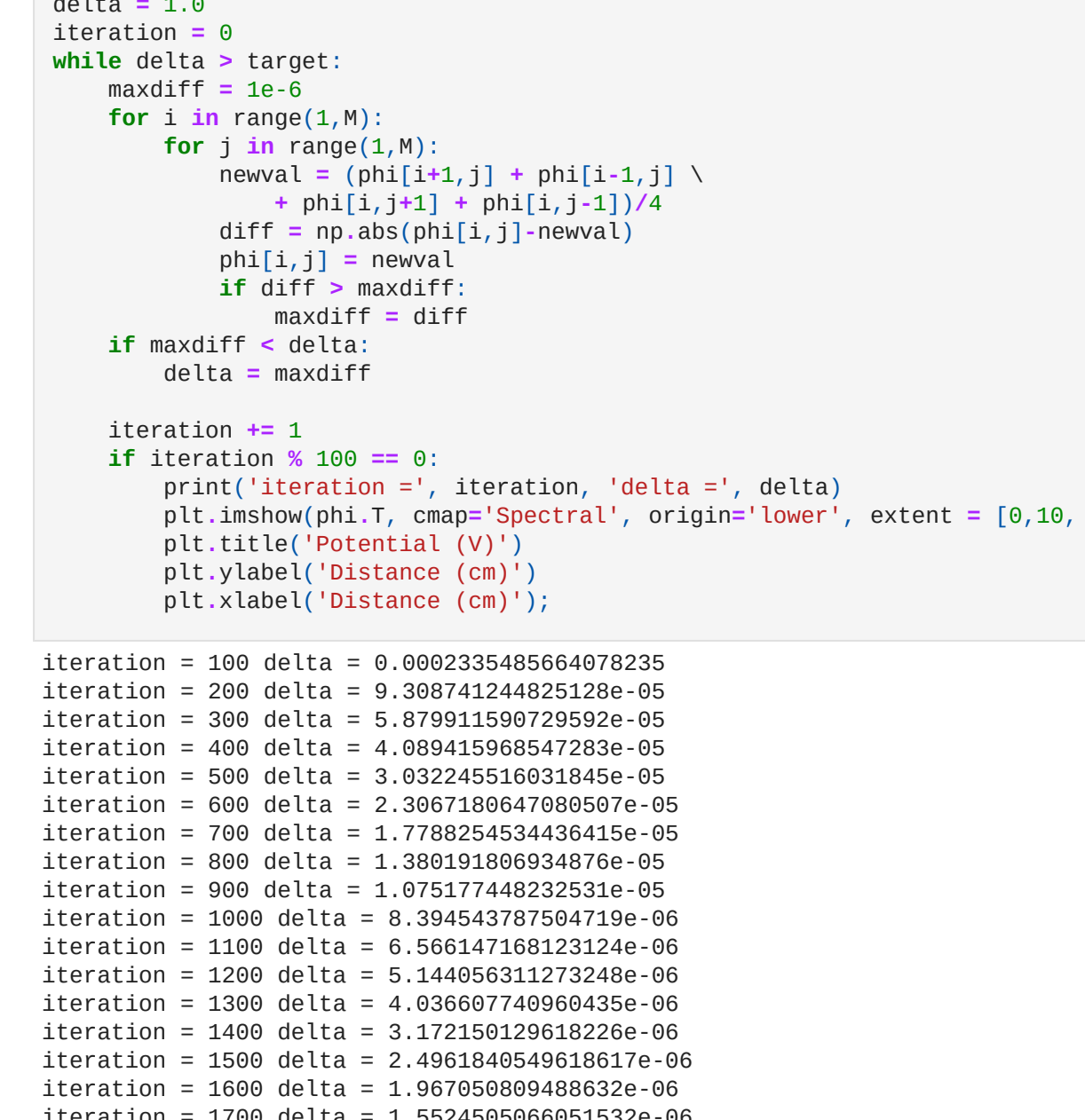
```
#grid square
M = 100
#voltage at all wals
V = 0.0
#accuracy
target = 1e-6
#relaxation parameter
omega = 0.9

#making the grids
phi = np.zeros([M+1, M+1], float)

xaxis = yaxis = np.arange(0,10,.1)

#Setting boundary conditions
for xi, x in enumerate(xaxis):
    for yi, y in enumerate(yaxis):
        if x == 2.0:
            if y<= 8.0 and y>= 2.0:
                phi[xi, yi] += 1
        elif x == 8.0:
            if y<= 8.0 and y>= 2.0:
                phi[xi, yi] -= 1

plt.imshow(phi.T, extent = [0,10, 0, 10])
plt.colorbar()
plt.title('Inital Conditions for V')
plt.ylabel('Distance (cm)')
plt.xlabel('Distance (cm)');
```



In [114…

```
#Following the code we used in lecture 17:
delta = 1.0
iteration = 0
while delta > target:
    maxdiff = 1e-6
    for i in range(1,M):
        for j in range(1,M):
            newval = (phi[i+1,j] + phi[i-1,j] \
                + phi[i,j+1] + phi[i,j-1])/4
            diff = np.abs(phi[i,j]-newval)
            phi[i,j] = newval
            if diff > maxdiff:
                maxdiff = diff
    if maxdiff < delta:
        delta = maxdiff

    iteration += 1
    if iteration % 100 == 0:
        print('iteration =', iteration, 'delta =', delta)
        plt.imshow(phi.T, cmap='Spectral', origin='lower', extent = [0,10, 0, 10])
        plt.title('Potential (V)')
        plt.ylabel('Distance (cm)')
        plt.xlabel('Distance (cm)');
```

```
iteration = 100 delta = 0.0002335485664078235
iteration = 200 delta = 9.308741244825128e-05
iteration = 300 delta = 5.879911590729592e-05
iteration = 400 delta = 4.089415968547283e-05
iteration = 500 delta = 3.032245516031845e-05
iteration = 600 delta = 2.3067180647080507e-05
iteration = 700 delta = 1.7788254534436415e-05
iteration = 800 delta = 1.380191806934876e-05
iteration = 900 delta = 1.075177448232531e-05
iteration = 1000 delta = 8.394543787504719e-06
iteration = 1100 delta = 6.566147168123124e-06
iteration = 1200 delta = 5.144056311273248e-06
iteration = 1300 delta = 4.036607740960435e-06
iteration = 1400 delta = 3.172150129618226e-06
iteration = 1500 delta = 2.4961840549618617e-06
iteration = 1600 delta = 1.967050809488632e-06
iteration = 1700 delta = 1.5524505066051532e-06
iteration = 1800 delta = 1.2272839945886085e-06
```



Problem 5

Problem 5: Thermal diffusion in the Earth's crust

A classic example of a diffusion problem with a time-varying boundary condition is the diffu- sion of heat into the crust of the Earth, as surface temperature varies with the seasons. Suppose 3 the mean daily temperature at a particular point on the surface varies as: T0(t) = A + B sin 2πt τ , where τ = 365 days, A = 10◦C and B = 12◦C. At a depth of 20 m

below the surface almost all annual temperature variation is ironed out and the temperature is, to a good approximation, a constant 11◦C

The thermal diffusivity of the Earth's crust varies somewhat from place to place, but for our purposes we will treat it as constant with value

D = 0.1 m2 day−1.

Write a program, or modify one of the ones given in this chapter, to

calculate the temperature profile of the crust as a function of depth up to 20 m and time up to 10 years.

Start with temperature everywhere equal to 10◦C, except at the surface and the deepest point,

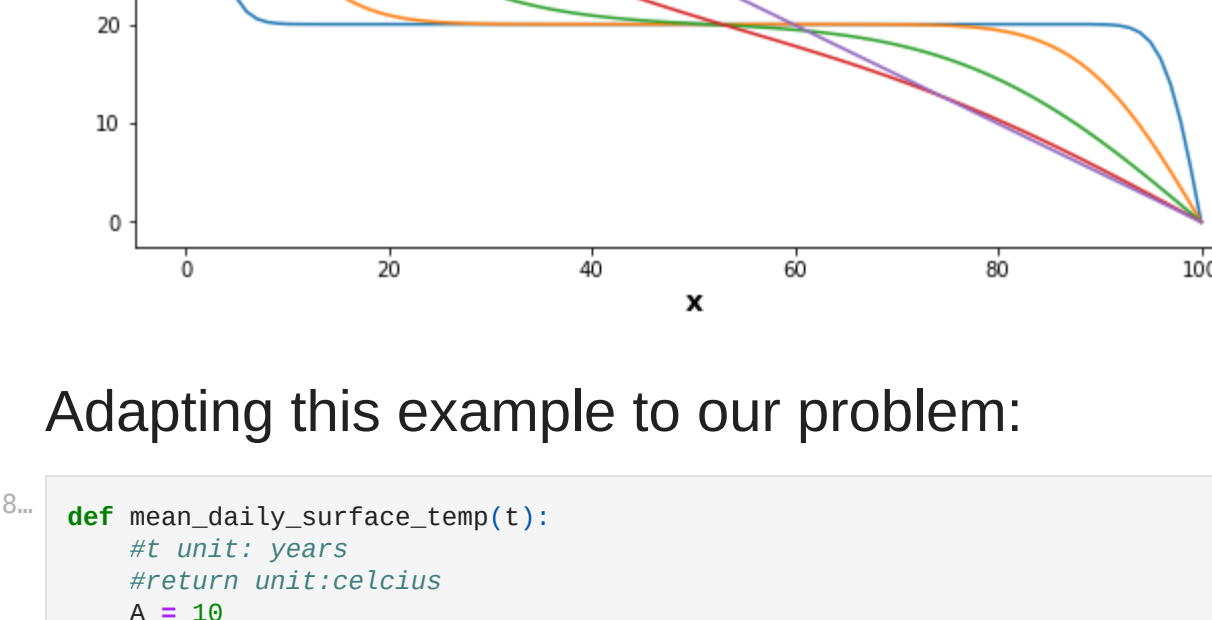
choose values for the number of grid points and the time-step h,

then run your program for the first nine simulated years, to allow it to settle down into whatever pattern it reaches. Then for the tenth and final year plot four temperature profiles taken at 3-month intervals on a single graph to illustrate how the temperature changes as a function of depth

- I'll be adapting from example 9.3 in the book:

In [411…

```
from numpy import empty
from pylab import plot,xlabel,ylabel,show

#Constants
L = 0.01
# Thickness of steel in meters
D = 4.25e-6
# Thermal diffusivity
N = 100
# Number of divisions in gri&
a = L/N
# Grid spacing
h = 1e-4
# Time-step
epsilon = h/1000

Tlo = 0.0
Tmid = 20.0
Thi = 50.0

t1 = 0.01
t2 = 0.1
t3 = 0.4
t4 = 1.0
t5 = 10.0
tend = t5 + epsilon

# Create arrays
T = empty(N+1,float)
T[0] = Thi
T[N] = Tlo
T[1:N] = Tmid
Tp = empty(N+1,float)
Tp[0] = Thi
Tp[N] = Tlo
# Main loop
t = 0.0
c = h*D/(a*a)
while t<tend:
    # Calculate the new values of T
    for i in range(1,N):
        Tp[i] = T[i] + c*(T[i+1]+T[i-1]-2*T[i])
    T,Tp = Tp,T
    t += h
    # Make plots at the given times
    if abs(t-t1)<epsilon:
        plot(T)
    if abs(t-t2)<epsilon:
        plot(T)
    if abs(t-t3)<epsilon:
        plot(T)
    if abs(t-t4)<epsilon:
        plot(T)
    if abs(t-t5)<epsilon:
        plot(T)
xlabel ("x")
ylabel("T")
show()
```



Adapting this example to our problem:

In [538…

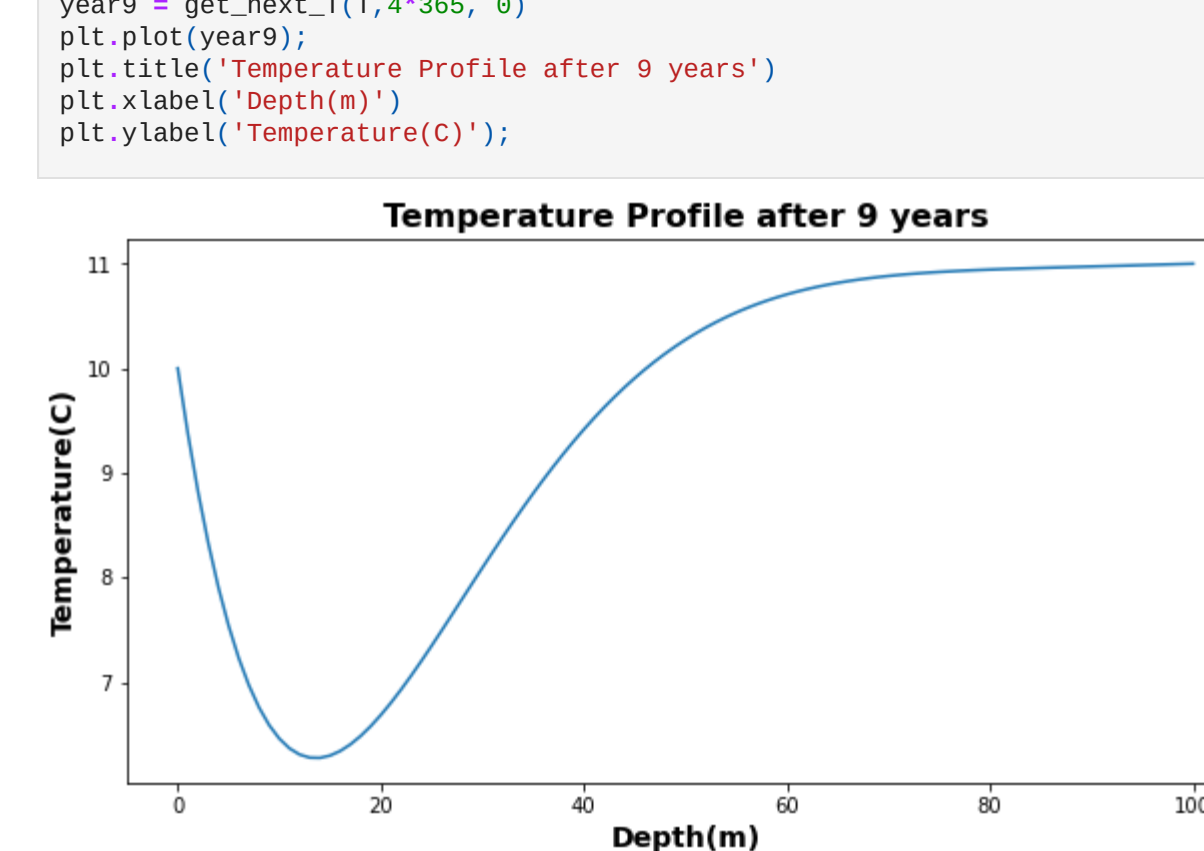
```
def mean_daily_surface_temp(t):
    #t unit: years
    #return unit:celsius
    A = 10
    B = 12
    #Tao is one year. I know this is not an entirely accurate value but for the purposes of this problem
    #taking a year to be 365 days simplifies things.
    tao = 365
    return A+B*np.sin(2*np.pi*t/tao)
```

In [539…

```
def get_next_T(T, tend, tstart):
    t = tstart
    while t<tend:
        #All the middle portions of the ocean: simplified python indexing from book pg 423
        T[1:N] =+c*(T[0:N-1]+T[2:N+1]-2*T[1:N])
        T[0] = mean_daily_surface_temp(t)
        T[N] = 11.0
        t+=h
    return T
```

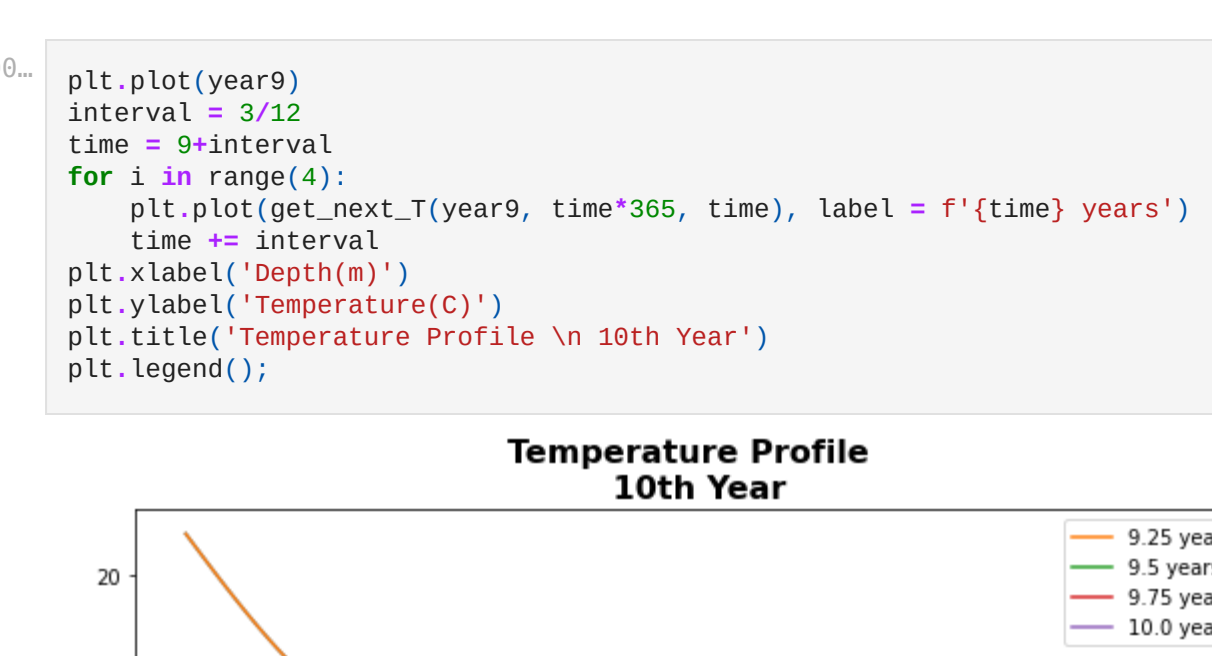
In [697…

```
#Defining the constants
N = 100
a = 0.001
h = 0.1
epsilon = h/1000
Tall = 11
D = 0.1/36565
c = h*D/(a**2)
#Initialize the ocean temperature
T = np.repeat(10.0, N+1)
T[0]=0
T[-1] = 11
#Call and plot our function
year9 = get_next_T(T,4*365, 0)
plt.plot(year9)
plt.title('Temperature Profile after 9 years')
plt.xlabel('Depth(m)')
plt.ylabel('Temperature(C)')
```



In [700…

```
plt.plot(year9)
interval = 3/12
time = 9+interval
for i in range(4):
    plt.plot(get_next_T(year9,time*365, time), label = f'{time} years')
    time += interval
plt.xlabel('Depth(m)')
plt.ylabel('Temperature(C)')
plt.title('Temperature Profile \n 10th Year')
plt.legend();
```



Problem 6

a

In [635…

```
from dcst import *
plt.rcParams['figure.figsize'] = (10, 5)
```

In [638…

```
#grid space
N=1000
#box length
L=1e-8
#constants
x0 = L/2
sigma = 1e-10
k = 5e10
```

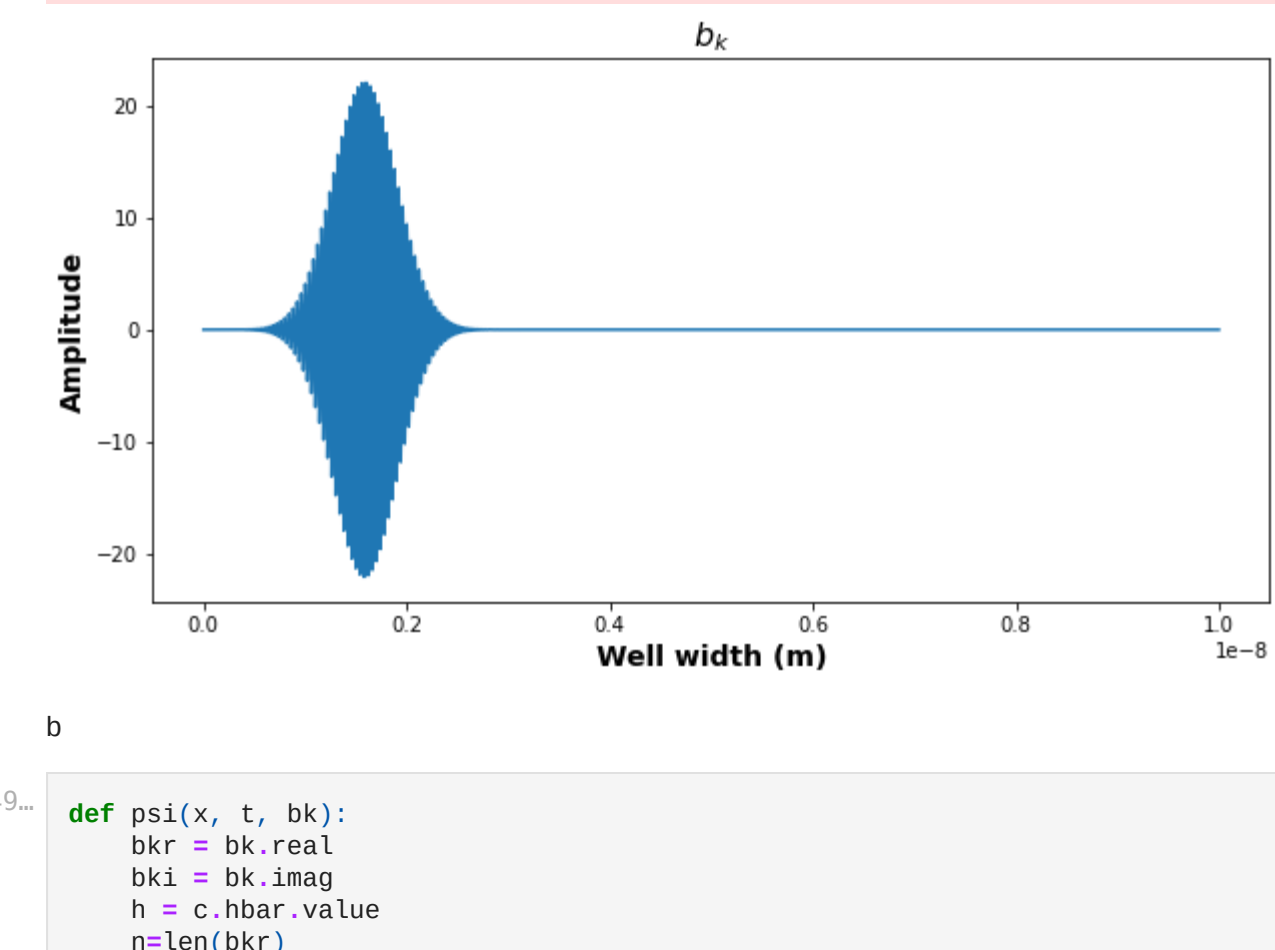
In [637…

```
def psi(x):
    return (np.exp(-1*((x-x0)**2)/(2*(sigma**2))))*np.exp(1j*k*x))
```

In [638…

```
bks = np.zeros(N+1, complex)

xaxis = np.linspace(0,L,N, complex)
psival = psi(xaxis)
psival[0]=psival[-1]=0
plt.plot(xaxis,psival)
plt.xlabel('Well width (m)')
plt.ylabel('Psi')
plt.title('Wave Function');
```

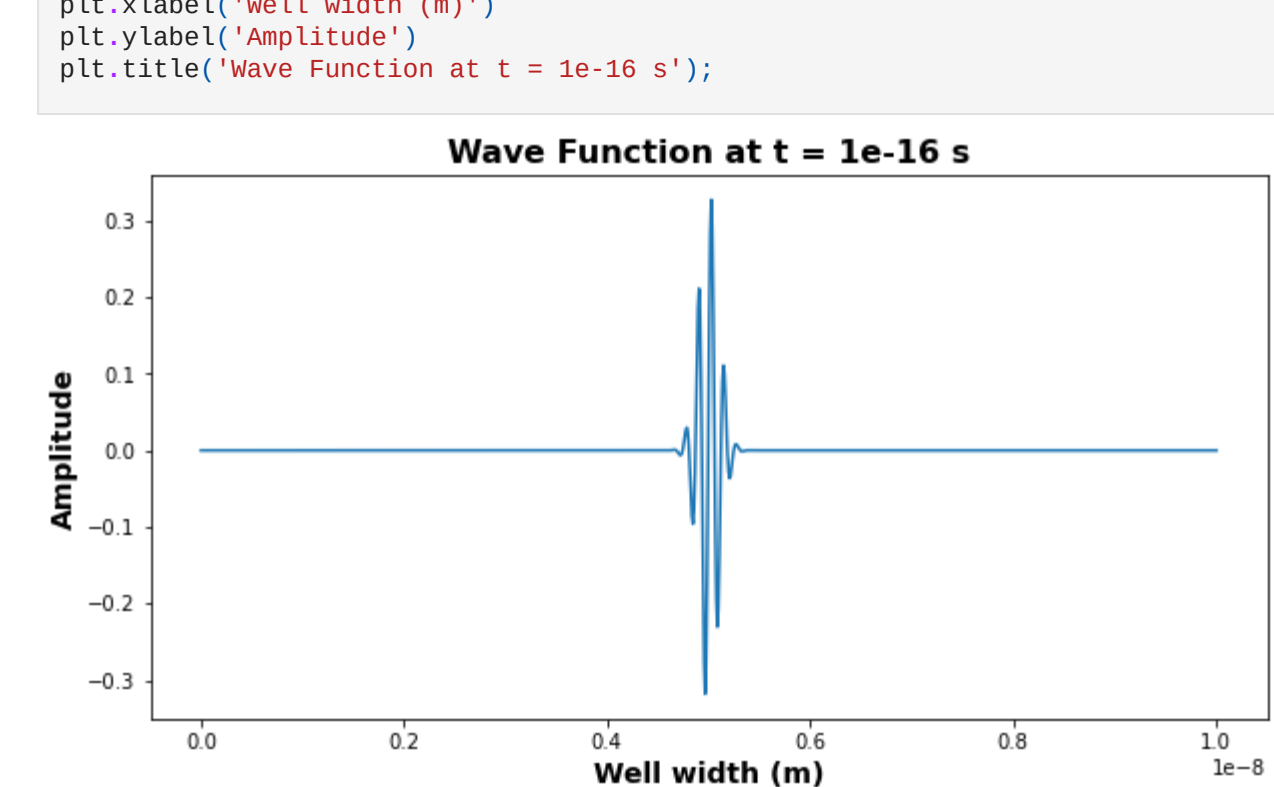
```
/home/kelcey/anaconda3/envs/hera/lib/python3.9/site-packages/matplotlib/cbook/__init__.py:1333: ComplexWarning:
Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```



In [642…

```
bk = dst(psival)
plt.plot(xaxis,bk)
plt.xlabel('Well width (m)')
plt.ylabel('Amplitude')
plt.title('$b_k$');
```

```
/home/kelcey/computational_phys/Homework5/dcst.py:99: ComplexWarning: Casting complex values to real discards t
he imaginary part
  y2[1:N] = y[1:]
/home/kelcey/computational_phys/Homework5/dcst.py:100: ComplexWarning: Casting complex values to real discards
the imaginary part
  y2[:N:-1] = -y[1:]
```



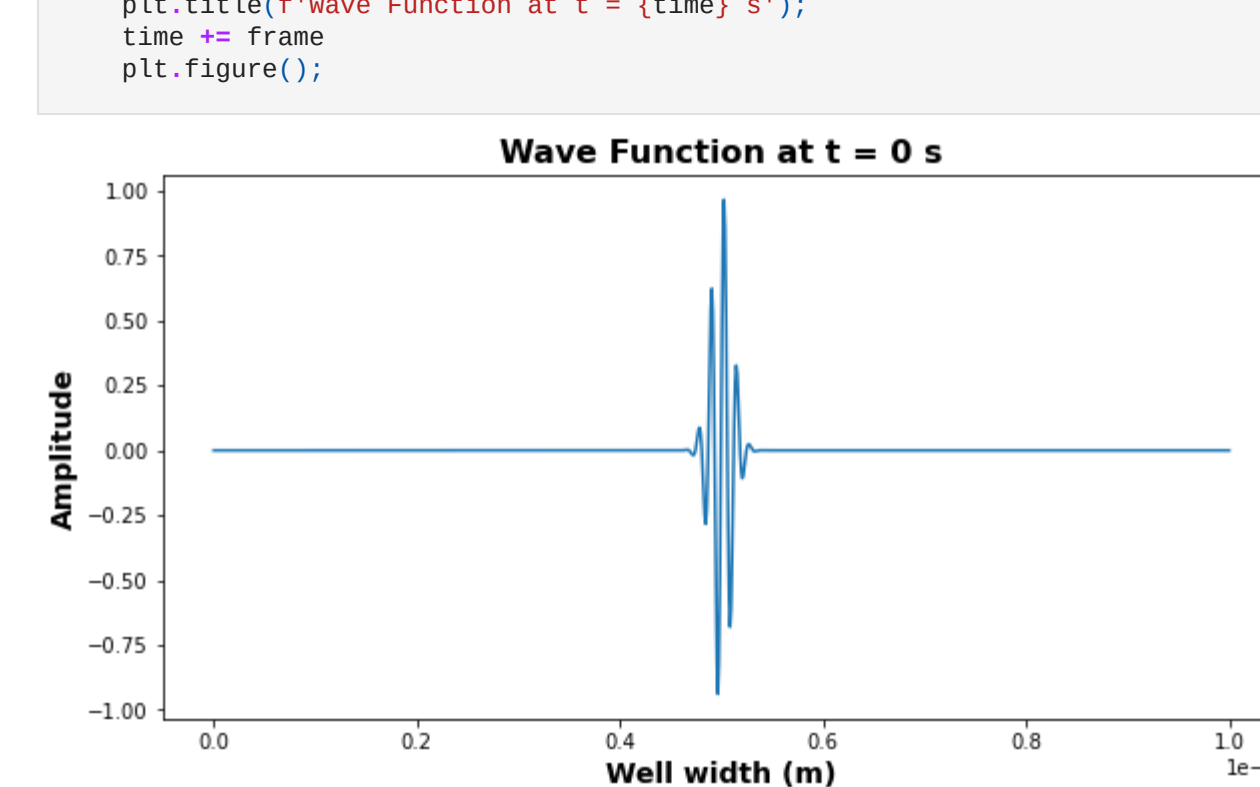
b

In [649…

```
def psi(x, t, bk):
    bkr = bk.real
    bki = bk.imag
    h = c.hbar.value
    n=len(bkr)
    M = 9.109e-31
    sm = bkr*np.cos((np.pi**2)*h*(k**2)*t/(2*M*(L**2))) - bki*np.sin((np.pi**2)*h*(k**2)*t/(2*M*(L**2)))
    return idst(sm)
```

In [656…

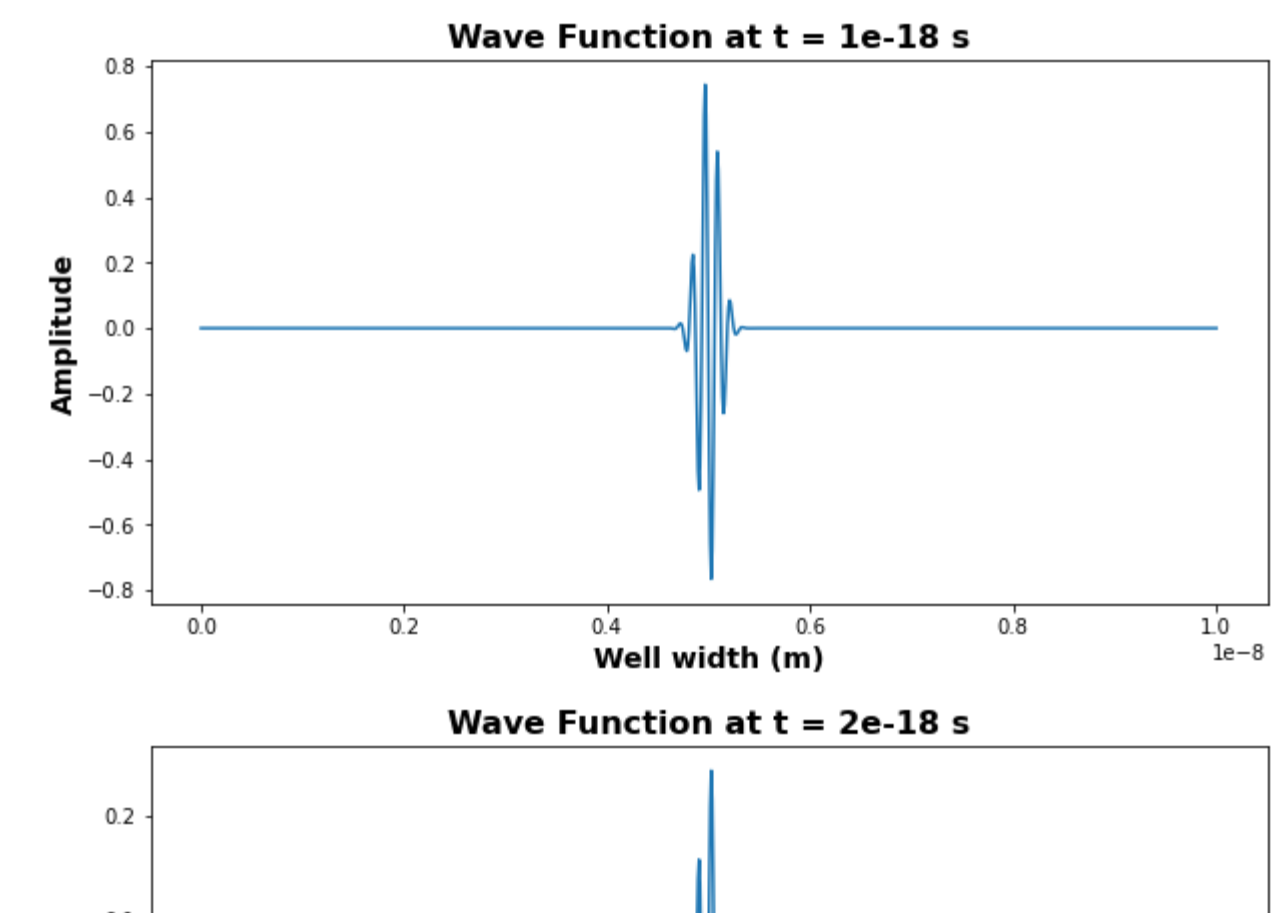
```
p = psi(xaxis, 1e-16, bk)
plt.plot(xaxis, p)
plt.xlabel('Well width (m)')
plt.ylabel('Amplitude')
plt.title('Wave Function at t = 1e-16 s');
```

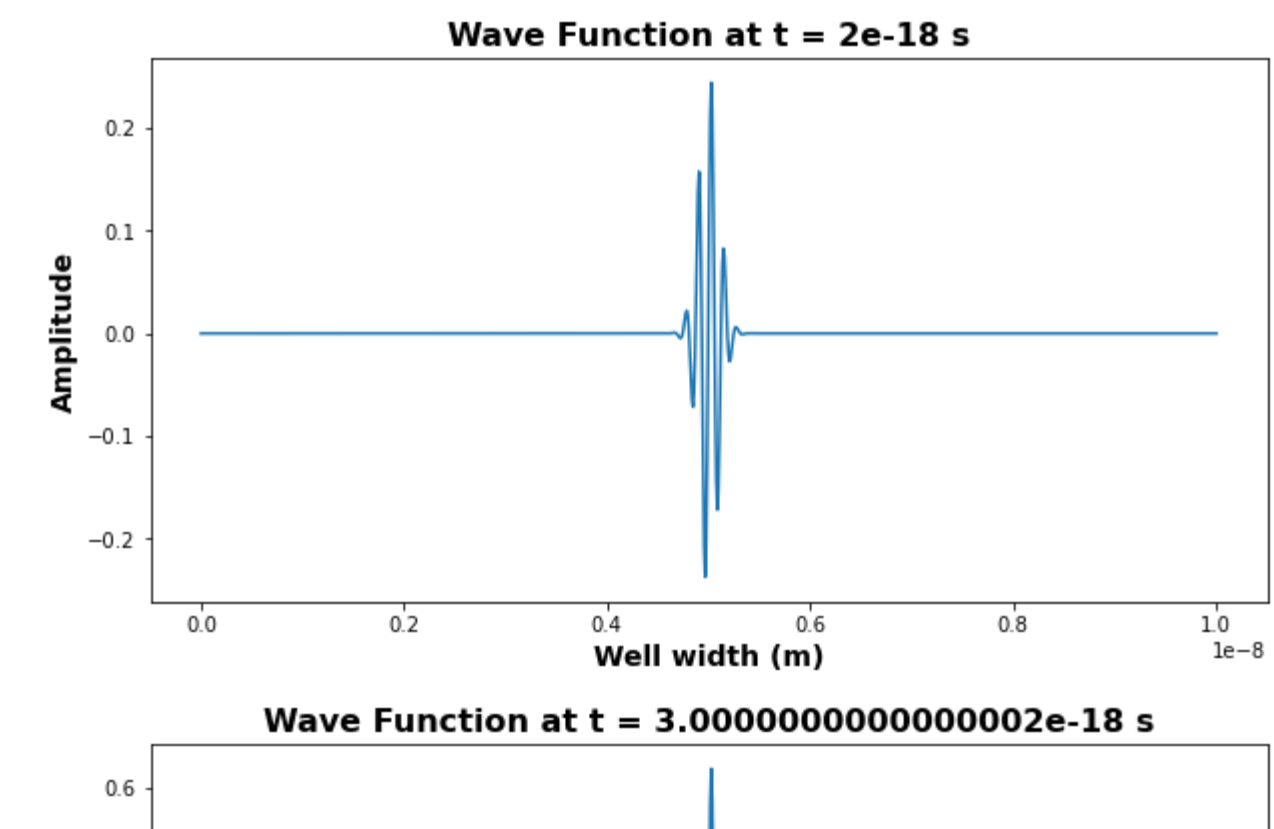


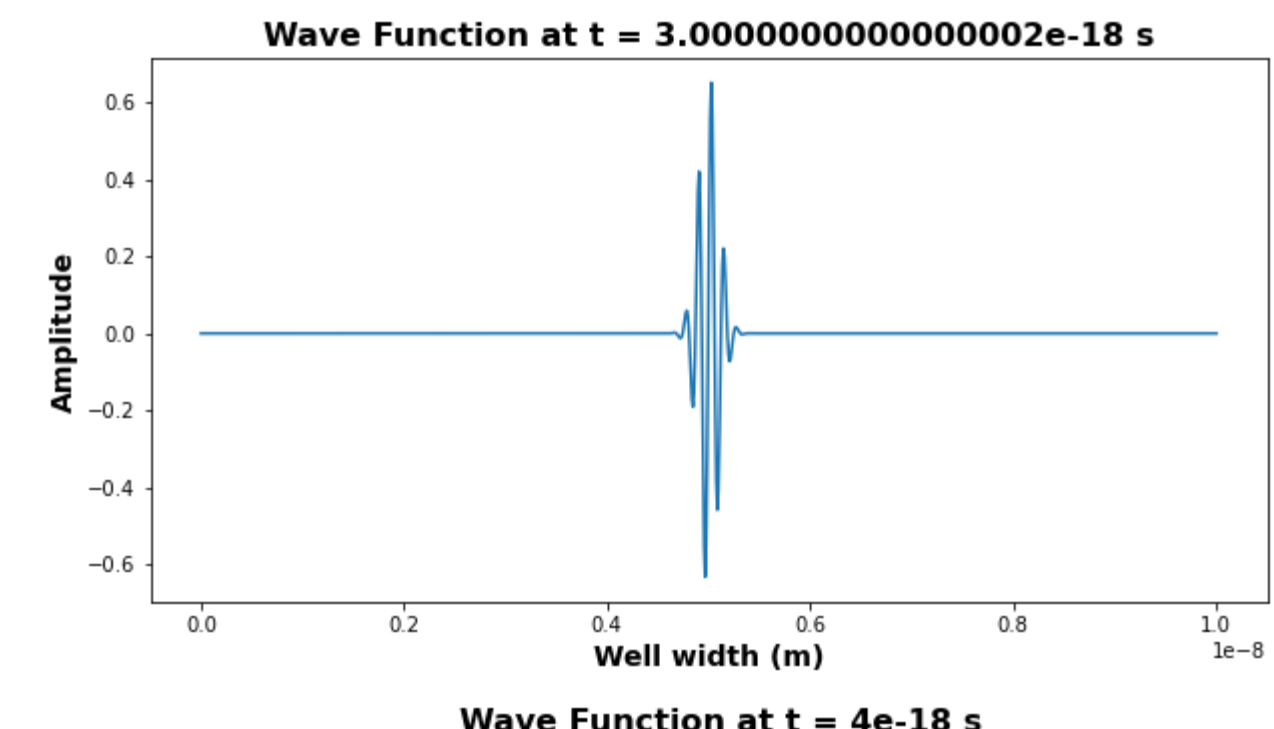
c

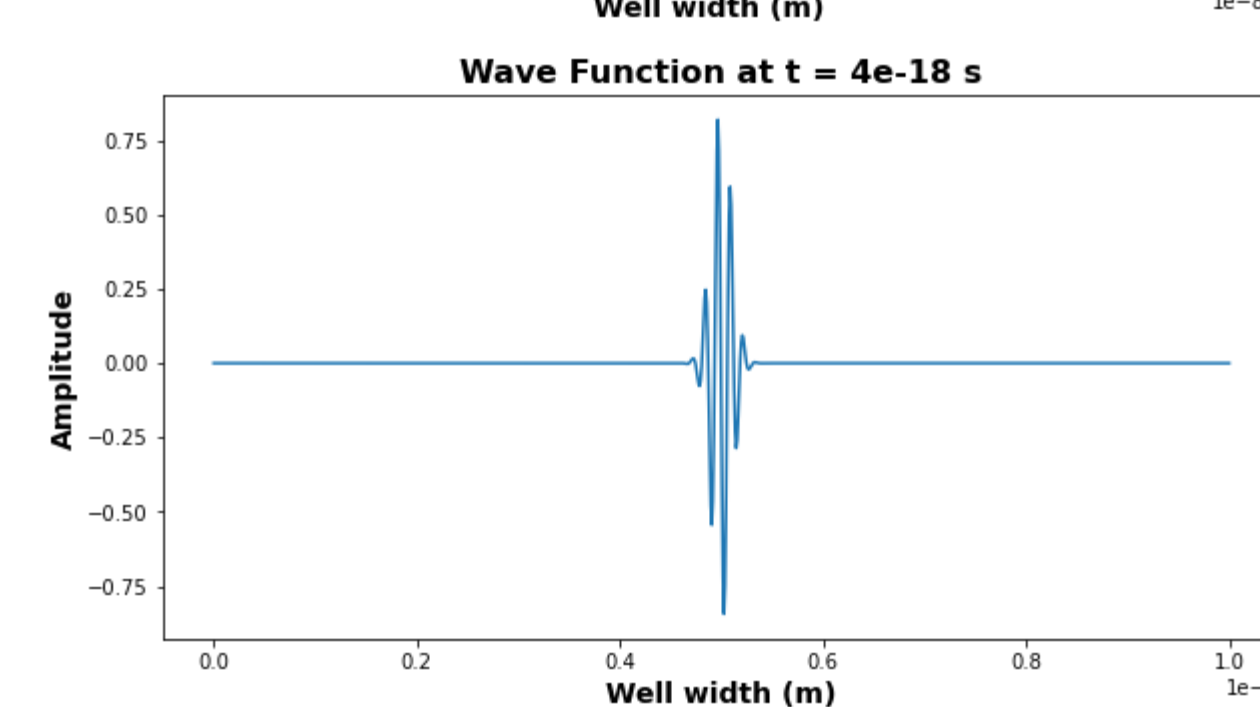
In [654…

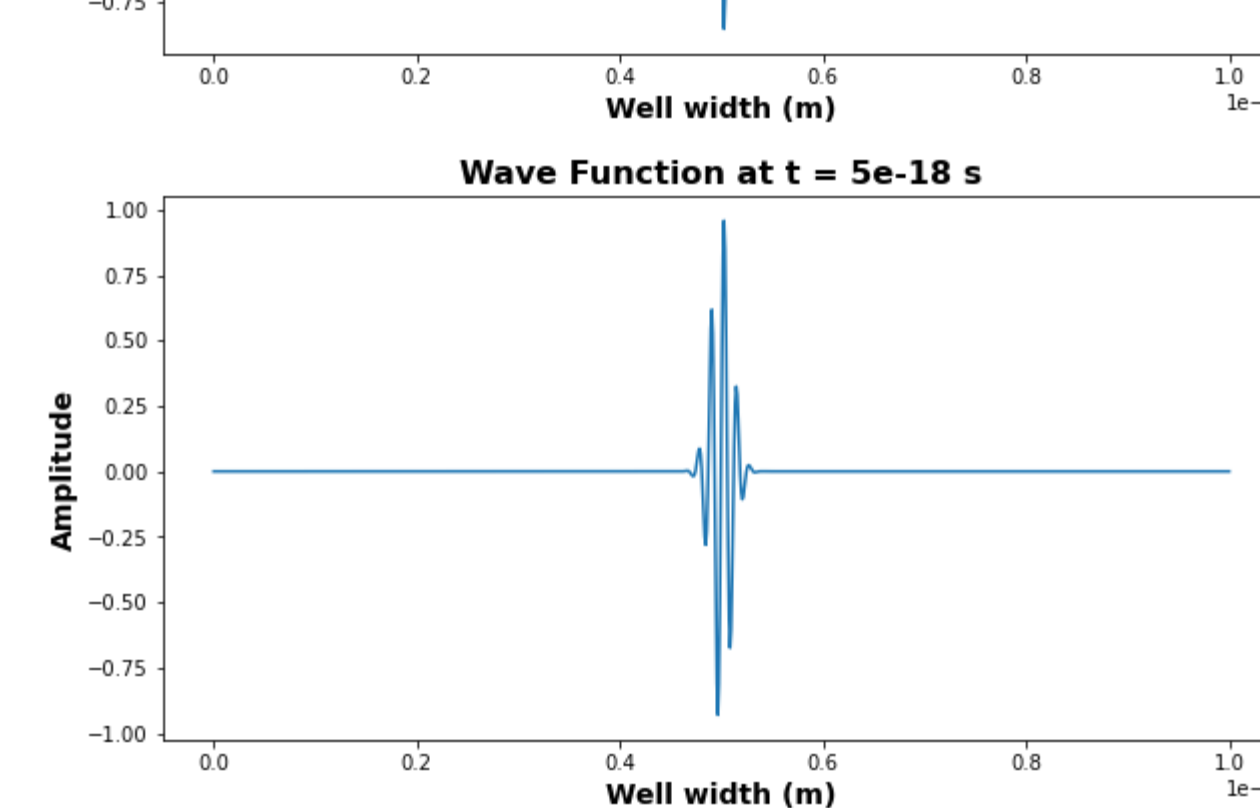
```
frame = 1e-18
time = 0
for i in range(1,10):
    p = psi(xaxis, time, bk)
    plt.plot(xaxis, p)
    plt.xlabel('Well width (m)')
    plt.ylabel('Amplitude')
    plt.title(f'Wave Function at t = {time} s');
    time += frame
    plt.figure();
```

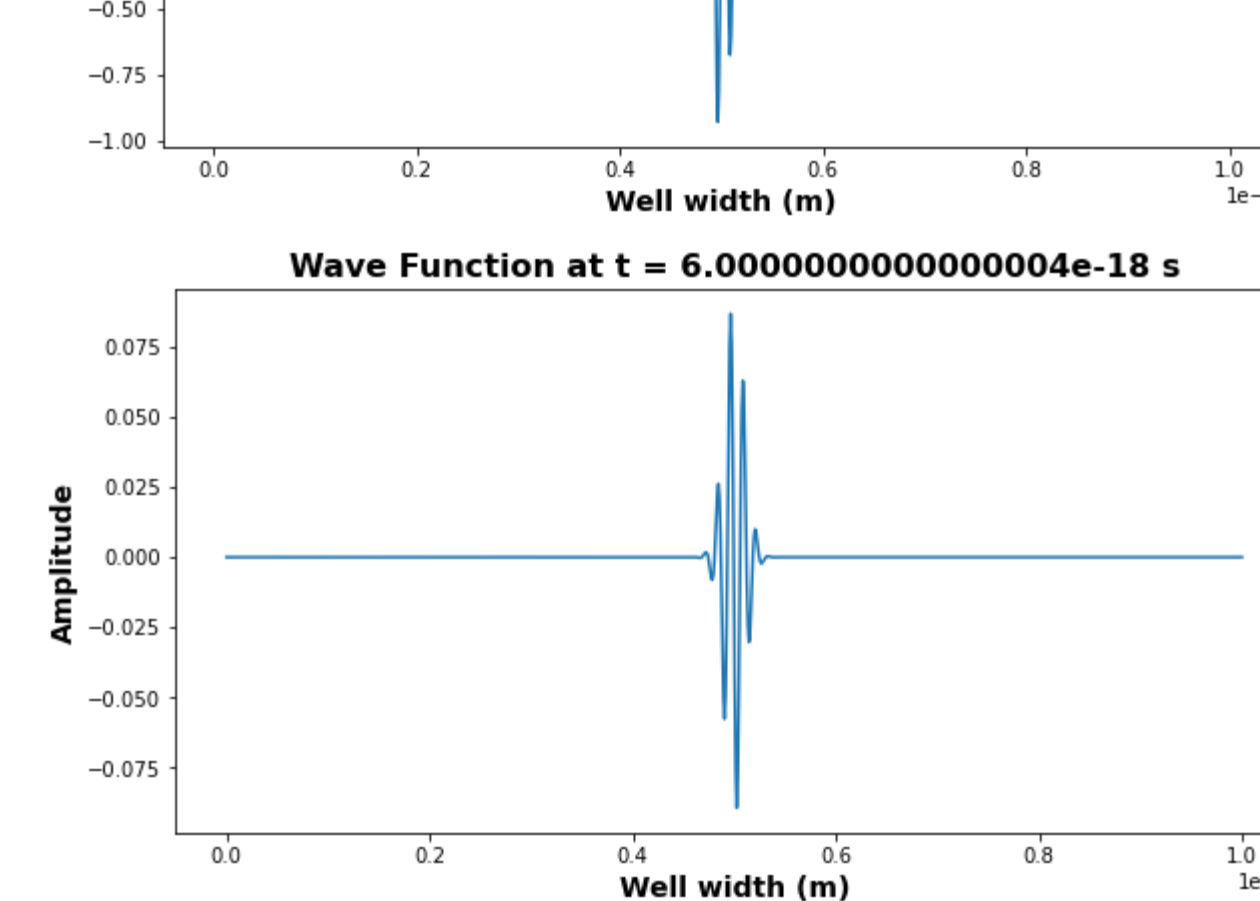


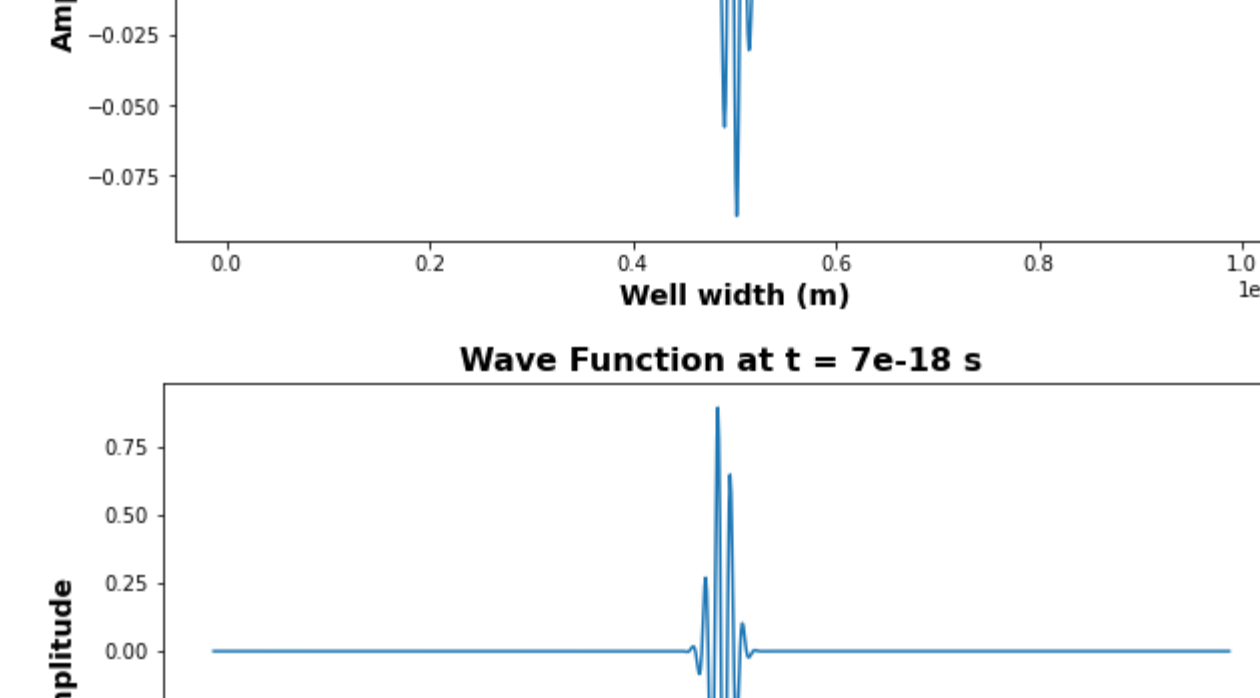


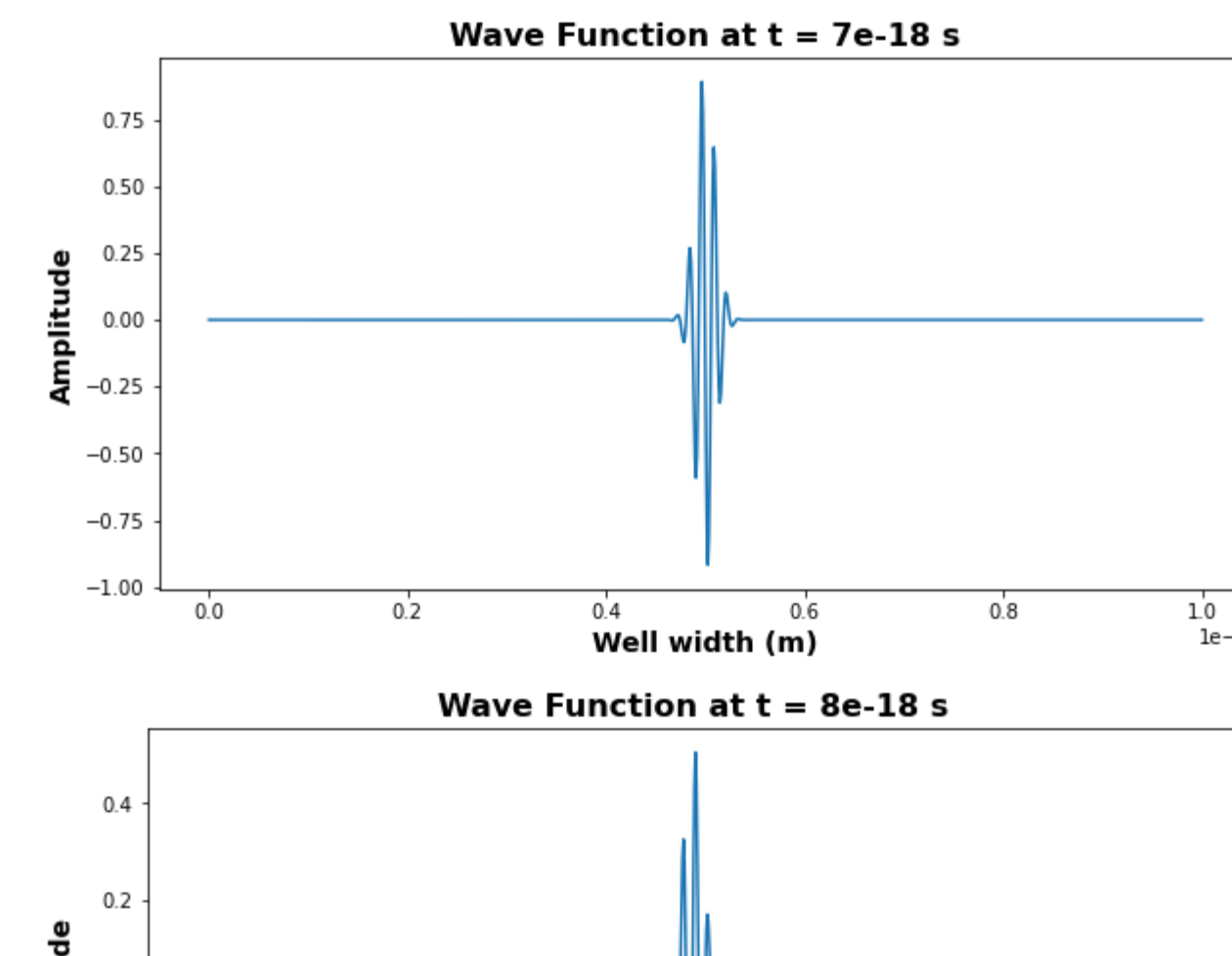


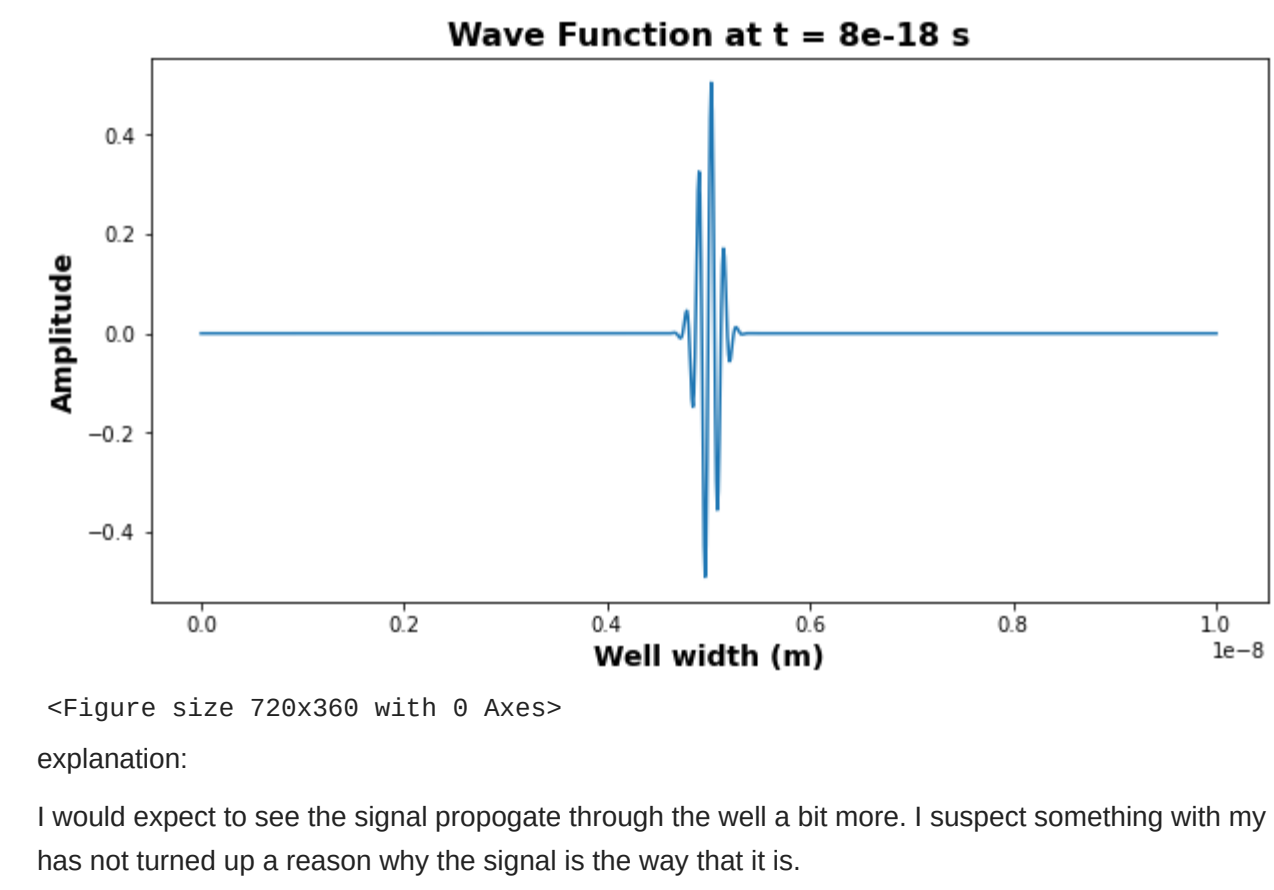













```
<Figure size 720x360 with 0 Axes>
```

explanation:

I would expect to see the signal propogate through the well a bit more. I suspect something with my constants but playing around with it has not turned up a reason why the signal is the way that it is.