PH30110 Computational Astrophysics - Coursework 1
21777

1a)
This question involved modelling the orbit of Halley's comet around the sun using the equation of motion

$$m\frac{d^2\boldsymbol{r}}{dt^2} = -\frac{GMm}{r^2}\frac{\boldsymbol{r}}{r}, \qquad (1)$$

where m is the mass of the orbiting body, $\boldsymbol{r}$ is the position vector of the orbiting body, t is the time, G is the gravitational constant, M is the mass of the body being orbited and r is the distance between the two bodies (the coordinate system is orientated such that the body orbits in the $z = 0$ plane, therefore $r = \sqrt{x^2 + y^2}$). The method used to solve this equation was a fourth order Runge-Kutta (RK4) method which relies on solving first order ordinary differential equations (ODEs). Equation 1 can first be split into its x and y components creating two second order ODEs. Each of these were then split into two first order ODEs to produce a final set of four first order ODEs,

$$\frac{dx}{dt} = A, \frac{dA}{dt} = -\frac{GMx}{(x^2+y^2)^{3/2}} \qquad (2)(3)$$
$$\frac{dy}{dt} = B, \frac{dB}{dt} = -\frac{GMy}{(x^2+y^2)^{3/2}}, \qquad (4)(5)$$

where x and y are the coordinates of the satellite and A and B are new variables. The RK4 method was initially designed using a single first order ODE with a known solution. The aim was to design a general method that could solve any arbitrary number of first order ODEs. To achieve this a vector notation for the equations was adopted. The program designed requires a strict definition of the order of the variables to be used, for example in this question the order of variables chosen was `[x,A,y,B,t]`. The order of the equations and initial conditions must be consistent with this definition (equations: `[dx/dt,dA/dt,dy/dt,dB/dt]`, initial conditions: `[x0,dx/dt0,y0,dy/dt0]`). The definition of each function must have two arguments, an array of the variables used within all the functions (as shown above) and an array of the constants used within all the functions (`[G,M]`). For example, equation (3) is defined as

```python
def dAdt(Cn,Var):
    return -1*Cn[1]*Cn[0]*(Var[0]/(Var[0]**2+Var[2]**2)**(3/2))
```

The arguments for the RK4 function are an array of the functions to be solved, an array of the initial conditions, an array of the constants to be used, the time to integrate to and the number of points. To test the generality of the function it was tested on three different situations, the single equation problem that it was initially developed with, the two Lotka-Volterra equations that model predator-prey interactions and the four equations defined in this problem ((2)(3)(4)(5)). The algorithm could solve all three situations simply by altering the setup of the program before the algorithm was run.

Figure 1 was produced from using the RK4 function with the four first order ODEs defined above and by plotting $x(t)$ against $y(t)$ for 75 years (the orbital period of Haley's Comet) where each data point gets darker with time. The initial conditions used were $x(0) = 5.2\times10^9 km$, $\frac{dx}{dt}(0) = 0, y(0) = 0, \frac{dy}{dt}(0) = 880\, m/s\,, t(0) = 0$, and it was set to be plotted with 10,000 points. The red dot indicates the fixed position of the Sun at (0,0). The difference of the scales on each axis shows how highly elliptical the orbit of Haley's comet is. The much lower rate of change of the colour next to the Sun shows that it moves much quicker at the perihelion than the aphelion.

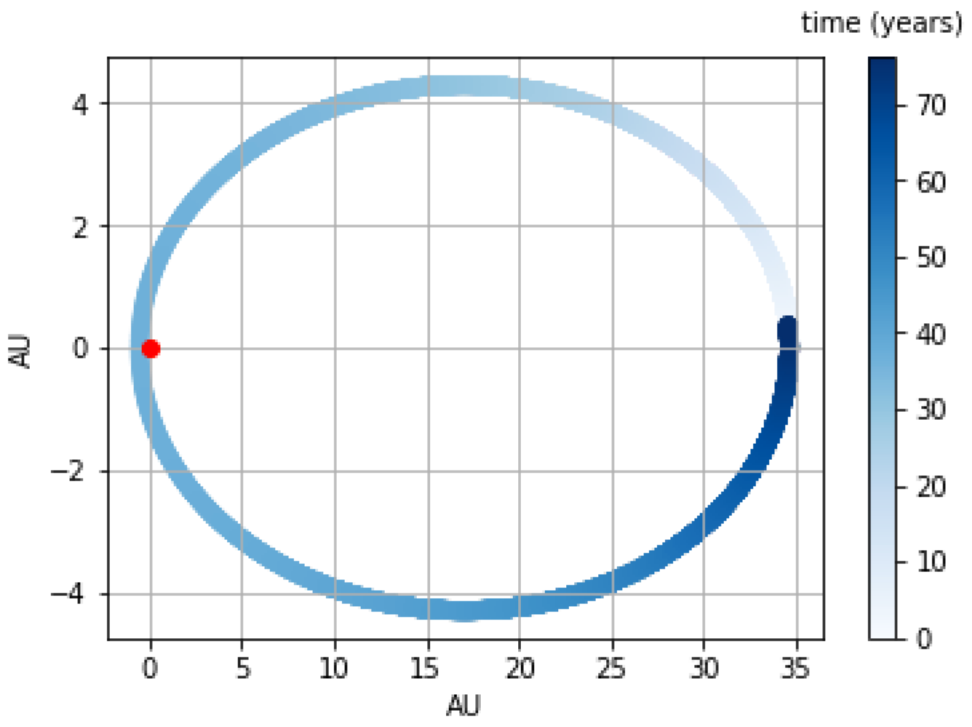

Figure 1. The orbit of Haley's comet using an RK4 method integrated for 75 years.

The plotting currently occurs inside of the RK4 function, to increase generality the function could return the two-dimensional array used to store all the data. This would allow the data to be used for any other potential purpose.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants

#define all the constants to be used in all the equations
M = 1.99*10**30
G = scipy.constants.G

#put the constants into an array
cnsts = [M,G]

#define the order of the variables that will be used as if in an array
#Var = [x,A,y,B,t]
```

```
#define the differential equations using the array of constants and the
array of variables
def dxdt(Cn,Var):
    return Var[1]

def dAdt(Cn,Var):
    return -1*Cn[1]*Cn[0]*(Var[0]/(Var[0]**2+Var[2]**2)**(3/2))


def dydt(Cn,Var):
    return Var[3]

def dBdt(Cn,Var):
    return -1*Cn[1]*Cn[0]*(Var[2]/(Var[0]**2+Var[2]**2)**(3/2))

#create an array of the functions
drdt = [dxdt,dAdt,dydt,dBdt]

#create an array of the initial conditions (keeping the order the same as
defined above)
#x0, dxdt0, y0, dydt0, t0
iniConds = [5.2*10**12,0,0,880,0]

#arguments(array of functions, array of initial conditions, array of
constants, time to integrate to, number of points)
def Rk4(dRdt,Ics,Cn,t_f,N):

    n=len(dRdt) #find the number of equations
    h=(t_f-Ics[n])/N #calculate step size

    #2D array to store the values for all the equations and time from 0 to
t_f
    Str = np.zeros((n+1, N))

    #populate the first value of each of the equations with the initial
conditions
    for i in range(0,n+1):
        Str[i][0] = Ics[i]

    #create arrays to store all 4 k values for each of the equations
    k1_= np.zeros(n)
    k2_= np.zeros(n)
    k3_= np.zeros(n)
    k4_= np.zeros(n)

    #iterate through all of the points
    for i in range(1,N):

        #create an array to store the input values for k1 function
        Var1 = np.zeros(n+1)

        #populate the array with stored values from the functions
        for k in range(0,n+1):
            Var1[k] = Str[k][i-1]

        #calculate the k1 constants for each function
        for j in range(0,n):
            k1_[j]=h*dRdt[j](Cn,Var1)

        #repeat as above but for the k2 constants
        Var2 = np.zeros(n+1)

        for k in range(0,n):
            Var2[k] = Str[k][i-1] + k1_[k]/2

        Var2[n] = Str[n][i-1] + h/2

        for j in range(0,n):
            k2_[j]=h*dRdt[j](Cn,Var2)

        #repeat as above but for the k3 constants
        Var3 = np.zeros(n+1)

        for k in range(0,n):
            Var3[k] = Str[k][i-1] + k2_[k]/2

        Var3[n] = Str[n][i-1] + h/2

        for j in range(0,n):
            k3_[j]=h*dRdt[j](Cn,Var3)

        #repeat as above but for the k4 constants
        Var4 = np.zeros(n+1)

        for k in range(0,n):
            Var4[k] = Str[k][i-1] + k3_[k]

        Var4[n] = Str[n][i-1] + h

        for j in range(0,n):
            k4_[j]=h*dRdt[j](Cn,Var4)

        #use the k constants to calculate the next values in each of the
functions
        for j in range(0,n):
            Str[j][i] = Str[j][i-1] + 1/6*(k1_[j]+2*k2_[j]+2*k3_[j]+k4_[j])

        #step forward one stepseize in time
        Str[n][i] = Str[n][i-1]+h

    #use the calculated values to plot as desired
    plt.scatter(Str[0]/(1.5*10**11),Str[2]/(1.5*10**11), c =
Str[n]/(3.154*10**7), cmap='Blues')

    clb = plt.colorbar()
    clb.set_label('time (years)', labelpad=-25, y=1.08, rotation=0)

    plt.grid()
    plt.xlabel('AU')
    plt.ylabel('AU')
    plt.plot(0,0, 'ro')
    plt.show()

Rk4(drdt,iniConds,cnsts,2.4*10**9,10000)
```

1b)
The function developed in the previous question relied on using a defined number of steps (and therefore a fixed step size) to integrate between the initial time and the final time (set at 10,000 for 1a). To improve the algorithm an adaptive step size can be used which relies on keeping the error per unit time constant. This speeds up the method over slowly changing sections and can increase the accuracy in quickly changing sections. This improvement to the algorithm was again developed using a single first order ODE ($\frac{dx}{dt} = \sin(t) + x^3$, see Figure 2). The error at a given point in time is calculated using the equation

$$\epsilon = \frac{1}{30}|x_1 - x_2|, \qquad (6)$$

where ϵ is the error, x_1 is value calculated at $x(t+2h)$ using two separate steps of h where h is the time step and x_2 is the value calculated at $x(t+2h)$ using one step of $2h$. This error is then compared to a set tolerance level, if the error is larger the time step is halved and the step is repeated, if the error is smaller the time step is doubled and the integration continues. One issue that had to be overcome implementing this was that the algorithm would get stuck at positions where the time step would be so small that the integration would take hours. This was solved by implementing a minimum value that the time step could take. However, this created the situation where even the minimum time step could not produce an error smaller than the tolerance and therefore the integration could not move forward. An array was created to store the time values for the previous three iterations of the loop and if all three were the same then this indicated that the loop was trapped. The function could then be moved forward by the minimum time step and the loop can continue.

In the previous algorithm, the length of the array used to store the data was known but using an adaptive time step means that this length is unknown. This was solved by implementing a routine that could extend the length of the storage array. The array is initially set to a length of 10 and then every time it fills up with data the length is increased by 10.

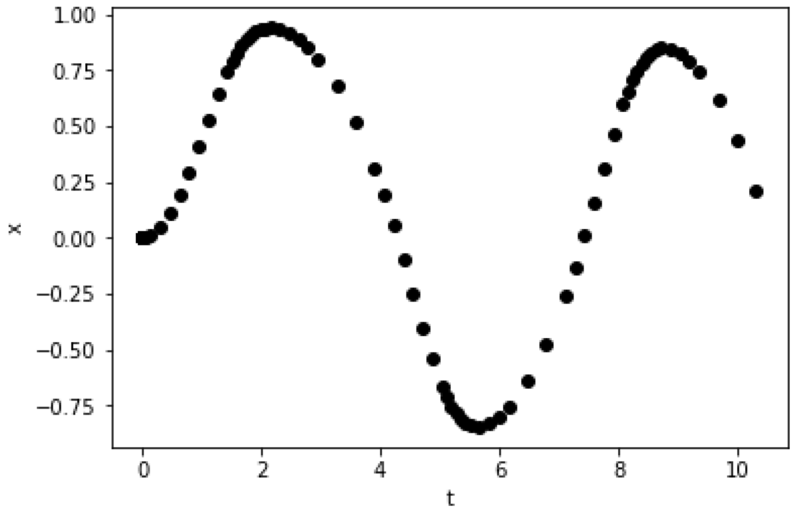

Figure 2. The adaptive time step method applied to the first order ODE $\frac{dx}{dt} = \sin(t) + x^3$ for time from 0 to 10.

To apply the algorithm to the orbit of Haley's comet it was first adapted, from the initial version created to produce Figure 2) so that the error was calculated from the distance between the Sun and the comet using both the *x* and *y* coordinates.

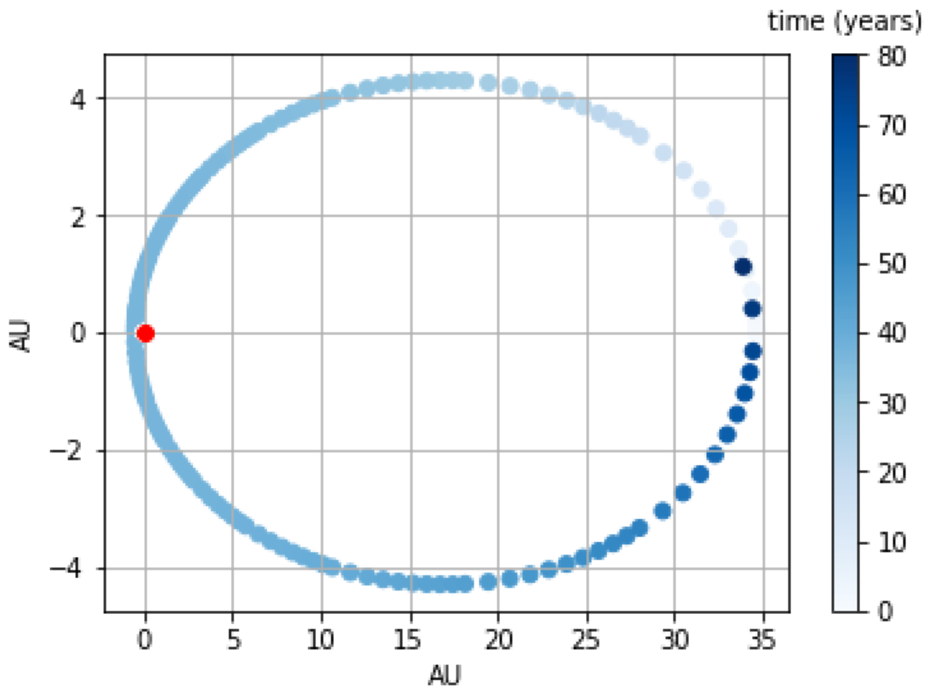

Figure 3. The adaptive time step method applied to the four first order ODEs describing the orbit of Haley's comet.

Figure 3 again shows that the speed of the comet is much faster at the perihelion than the aphelion as this is where the algorithm recorded many more data points. The function could further be improved to generalise the calculation for the tolerance, currently this must be set and adjusted by the user until a desirable plot is produced. A more advanced system for increasing and decreasing the step size could also be implemented that depends on the magnitude on the error at each point.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants

#define all the constants to be used in all the equations
M = 1.99*10**30
G = scipy.constants.G
#put the constants into an array
cnsts = [M,G]

#define the order of the variables that will be used as if in an array
#Var = [x,A,y,B,t]

#define the differential equations using the array of constants and the
array of variables
def dxdt(Cn,Var):
    return Var[1]
```

```
def dAdt(Cn,Var):
    return -1*Cn[1]*Cn[0]*(Var[0]/(Var[0]**2+Var[2]**2)**(3/2))

def dydt(Cn,Var):
    return Var[3]

def dBdt(Cn,Var):
    return -1*Cn[1]*Cn[0]*(Var[2]/(Var[0]**2+Var[2]**2)**(3/2))

#create an array of the functions
drdt = [dxdt,dAdt,dydt,dBdt]

#create an array of the initial conditions (keeping the order the same as
defined above)
#x0, dxdt0, y0, dydt0, t0
iniConds = [5.2*10**12,0,0,880,0]

#arguments(array of functions, array of initial conditions, array of
constants, time to integrate to, number of points)
def Rk4(dRdt,Ics,Cn,t_f,N):

    n=len(dRdt) #find the number of equations
    dt=(t_f-Ics[n])/N #calculate step size
    dt_min=dt/(100) #use the intial stepsize to calculate a minimum
acceptable stepsize

    #two 2D arrays to store the values for all the equations and time from
0 to t_f
    Str1 = np.zeros((n+1, 10))
    Str2 = np.zeros((n+1, 10))

    #populate the first value of each of the equations with the initial
conditions for the first array
    for i in range(0,n+1):
        Str1[i][0] = Ics[i]

    #define a new function to calculate the value of every functions at a
time h later
    #arguments(integer position, stepsize, array of equations, number of
equations)
    def step(i,h,Str,n):

        #create arrays to store all 4 k values for each of the equations
        k1_= np.zeros(n)
        k2_= np.zeros(n)
        k3_= np.zeros(n)
        k4_= np.zeros(n)

        #create an array to store the input values for k1 function
        Var1 = np.zeros(n+1)

        #populate the array with stored values from the functions
        for k in range(0,n+1):
            Var1[k] = Str[k][i-1]

        #calculate the k1 constants for each function
        for j in range(0,n):
            k1_[j]=h*dRdt[j](Cn,Var1)

        #repeat as above but for the k2 constants
        Var2 = np.zeros(n+1)

        for k in range(0,n):
            Var2[k] = Str[k][i-1] + k1_[k]/2

        Var2[n] = Str[n][i-1] + h/2

        for j in range(0,n):
            k2_[j]=h*dRdt[j](Cn,Var2)

        #repeat as above but for the k3 constants
        Var3 = np.zeros(n+1)

        for k in range(0,n):
            Var3[k] = Str[k][i-1] + k2_[k]/2

        Var3[n] = Str[n][i-1] + h/2

        for j in range(0,n):
            k3_[j]=h*dRdt[j](Cn,Var3)

        #repeat as above but for the k4 constants
        Var4 = np.zeros(n+1)

        for k in range(0,n):
            Var4[k] = Str[k][i-1] + k3_[k]

        Var4[n] = Str[n][i-1] + h

        for j in range(0,n):
            k4_[j]=h*dRdt[j](Cn,Var4)


        #use the k constants to calculate the next values in each of the 
functions
        for j in range(0,n):
            Str[j][i] = Str[j][i-1] + 1/6*(k1_[j]+2*k2_[j]+2*k3_[j]+k4_[j])

        #step forward one stepseize in time
        Str[n][i] = Str[n][i-1]+h

        #return the updated equations array
        return Str

    c = 1 #set counter to 1
    time = dt #set time to one stepsize (first time after the initial 
conditions)
    tol = 500000 #set the error tolerance

    #create an empty array used to prevent function getting trapped at the 
minimum stepsize
    t_check = np.zeros(3)
```

```
    #continue to iterate unitl the time has reached the final time set by
the user
    while time < t_f:

        #update the check array so that it contains the 3 most recent times
        t_check[2] = t_check[1]
        t_check[1] = t_check[0]
        t_check[0] = time

        #if the counter is up to the limit of the size storage array then
increase the size of the array
        if c == Str1.shape[1] - 1:

            Str3 = np.zeros((n+1, Str1.shape[1] + 10))
            Str3[:,0:Str1.shape[1]] = Str1[:,:]
            Str1 = Str3.copy()


        #take two steps in time each of size dt
        Str1 = step(c,dt,Str1,n)
        Str1 = step(c+1,dt,Str1,n)

        #calculate the distance from the sun
        r1 = (Str1[0][c+1]**2 + Str1[2][c+1]**2)**0.5

        Str2 = Str1.copy() #copy over the data to a temporary array

        #take one step in time of size 2*dt
        Str2 = step(c,2*dt,Str2,n)
        #print(Str2[0][c])

        #calculate the distance from the sun
        r2 = (Str2[0][c]**2 + Str2[2][c]**2)**0.5

        err = abs(r1 - r2)/30 #use the two distances to calculate the error


        #if the error is bigger than the set tolerance level
        if err > tol:

            #if the stepsize is at its minimum possible value and the time
hasnt changed in 3 iterations
            #this shows that the function is trapped at a point
            if dt == dt_min and t_check[2] == t_check[0]:

                c = c+1 #increment the counter by 1
                time = time + dt #increase the time by dt

            #if half dt is less than the minimum value then set dt to the
minimum value otherwise half dt
            elif dt/2 < dt_min:
                dt=dt_min
            else:
                dt = dt/2

        else:
            c = c+1 #increment the counter by 1
            time = time + dt #increase the time by dt
            dt = dt*2 #set dt to double its value


    #use the calculated values to plot as desired
    plt.scatter(Str1[0]/(1.5*10**11),Str1[2]/(1.5*10**11), c =
Str1[n]/(3.154*10**7), cmap='Blues')

    clb = plt.colorbar()
    clb.set_label('time (years)', labelpad=-25, y=1.08, rotation=0)

    plt.grid()
    plt.xlabel('AU')
    plt.ylabel('AU')
    plt.plot(0,0, 'ro')
    plt.show()

Rk4(drdt,iniConds,cnsts,2.4*10**9,10000)
```

2a)

To simulate a three-body problem using an RK4 method required altering the equations of motion used in the previous questions to account for the gravitational attraction between the satellites. This lead to eight first order ODEs

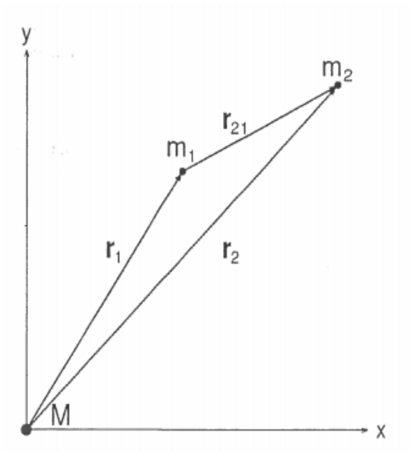

Figure 4. The labels for the bodies in the three-body problem.

$$\frac{dx_1}{dt} = A, \frac{dA}{dt} = -\frac{GMx_1}{(x_1{}^2+y_1{}^2)^{3/2}} + \frac{Gm_2(x_2-x_1)}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}} \quad (7)(8)$$

$$\frac{dy_1}{dt} = B, \frac{dB}{dt} = -\frac{GMy_1}{(x_1{}^2+y_1{}^2)^{3/2}} + \frac{Gm_2(y_2-y_1)}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}} \quad (9)(10)$$

$$\frac{dx_2}{dt} = C, \frac{dC}{dt} = -\frac{GMx_2}{(x_2{}^2+y_2{}^2)^{3/2}} - \frac{Gm_1(x_2-x_1)}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}} \quad (11)(12)$$

$$\frac{dy_2}{dt} = D, \frac{dD}{dt} = -\frac{GMy_2}{(x_2{}^2+y_2{}^2)^{3/2}} - \frac{Gm_1(y_2-y_1)}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}} \quad (13)(14)$$

where x_1 and y_1 are the coordinates of m_1 from Figure 4, x_2 and y_2 are the coordinates of m_2 and C and D are two more variables. These are solved using the adaptive time step method from the previous question. The method was adapted so that an error was calculated for both satellites, these are then both compared against the set tolerance level.

The initial conditions for both satellites also had to be calculated. The period of each orbit was calculated using the equation

$$P = \sqrt{\frac{4\pi^2 a^3}{G(m+M)}}, \quad (15)$$

where P is the period of orbit, a is the semi-major axis (radius for a circular orbit), m is the mass of the satellite and M is the mass of the body being orbited.

The initial velocities are then calculated using the equation

$$v = \sqrt{G(m+M)(\tfrac{2}{r}-\tfrac{1}{a})}, \tag{16}$$

where v is a velocity at any point in the orbit, r is the distance from the satellite to the object being orbited at that point and a is the semi-major axis of the orbit.
For the first run a circular orbit was assumed and the radii of the orbits were provided ($a_1 = 2.52AU, a_2 = 5.24AU, m_1 = 10^{-3}M_\circ, m_2 = 4\times10^{-2}M_\circ$, $M_\circ$ is one solar mass).

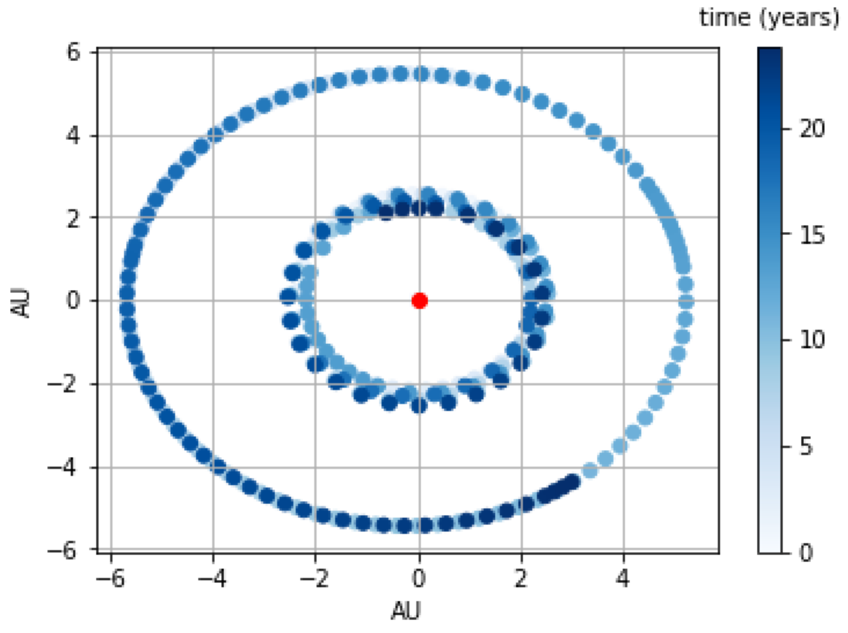

Figure 5. The orbits of two satellites around a central solar mass star fixed at (0,0) solved using an RK4 method with an adaptive step size. $a_1 = 2.52AU, a_2 = 5.24AU, m_1 = 10^{-3}M_\circ, m_2 = 4\times10^{-2}M_\circ$.

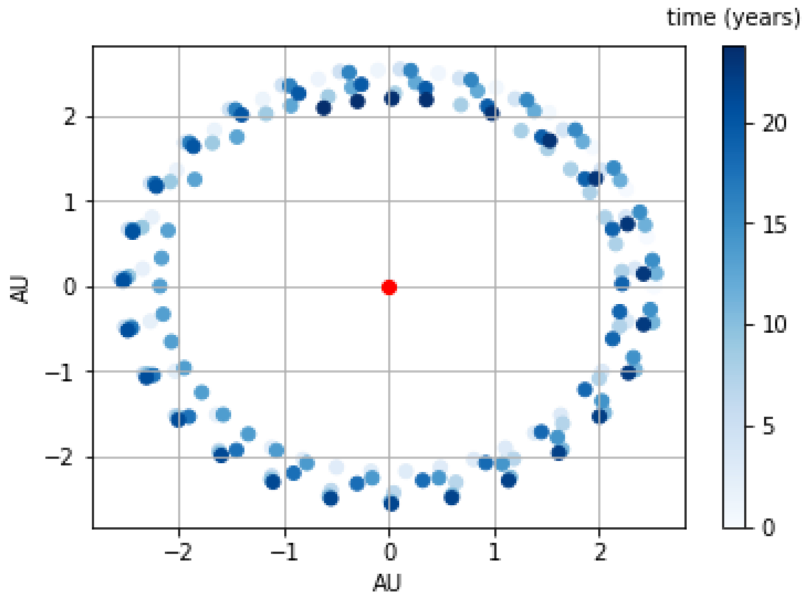

Figure 6. The inner orbit around a central solar mass star fixed at (0,0) solved using an RK4 method with an adaptive step size. $a_1 = 2.52AU, a_2 = 5.24AU, m_1 = 10^{-3}M_\circ$, $m_2 = 4\times10^{-2}M_\circ$.

The interactions between the two bodies can clearly be seen in Figure 6 which only displays the orbits of the inner, less massive satellite. Without the interaction, each orbit would exactly overlay the last but here there are variations between every orbit. Figures 7 and 8 show the same simulation repeated with data for Earth and Jupiter and then Jupiter and Saturn.

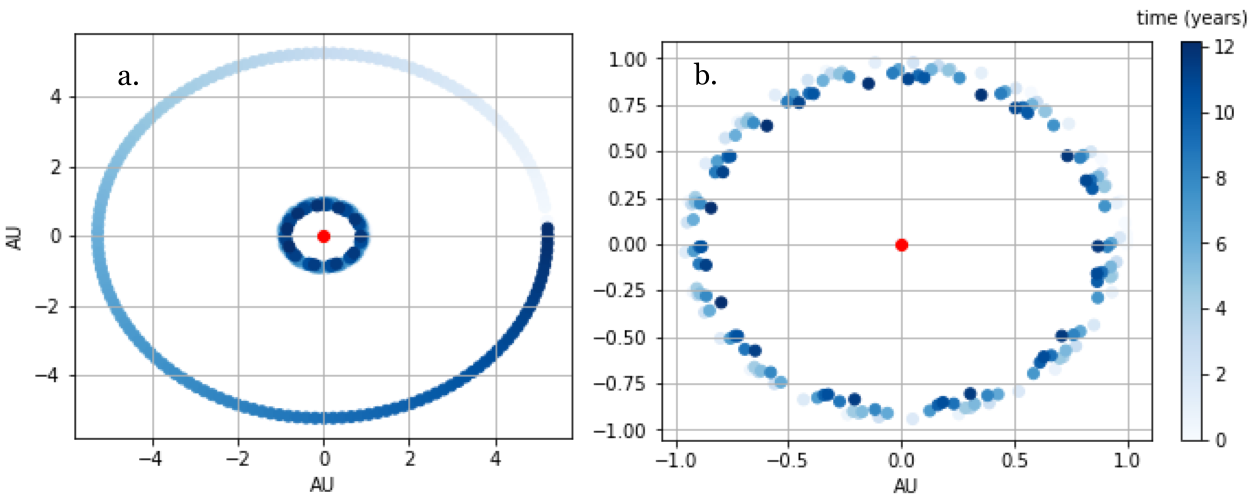

Figure 7a. the orbits of Earth and Jupiter plotted for one period of Jupiter's orbits (12 years). 7b. the orbits of the Earth over 12 years around a stationary Sun considering the gravitational interactions with Jupiter.

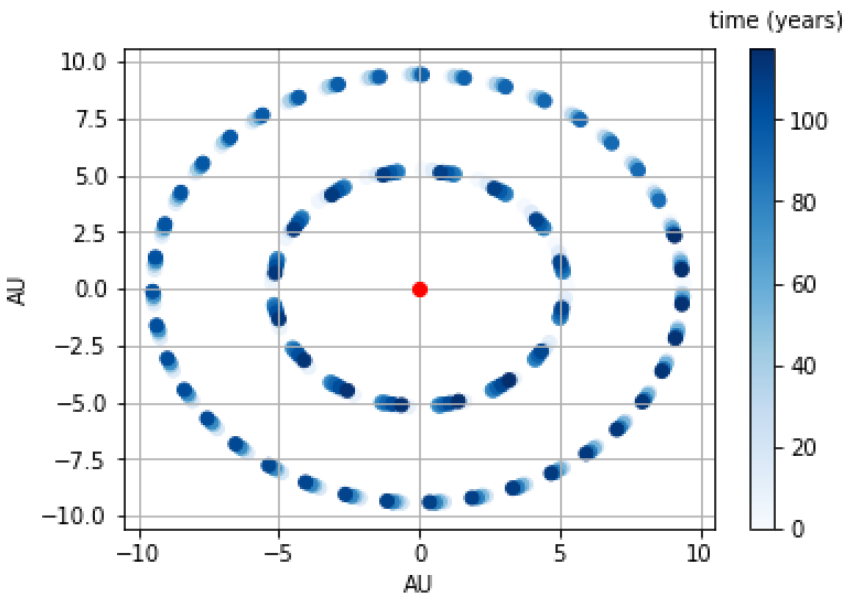

Figure 8. the orbits of Jupiter and Saturn plotted for four of Saturn's orbital periods (116 years).

Figure 7b again shows the orbital variations of the less massive body. Figure 8 shows that when the masses are much more similar the orbital variations are much subtler. The orbits of both Jupiter and Saturn process with each orbit but stay along almost identical paths.

The program could be further improved by generalising to account for any number of bodies and to have a personalised tolerance for each satellite based on their velocity.

Code (for plotting Figures 5 and 6)

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants

#define all the constants to be used in all the equations
M = 1.99*10**30
m1 = 10**-3*M
m2 = 4*10**-2*M
G = scipy.constants.G
#put the constants into an array
cnsts = [M,m1,m2,G]

#define the order of the variables that will be used as if in an array
#Var = [x1,A,y1,B,x2,C,y2,D,t]

#define the differential equations using the array of constants and the
array of variables
def dx1dt(Cn,Var):
    return Var[1]

def dAdt(Cn,Var):
    return -
1*Cn[3]*Cn[0]*(Var[0]/(Var[0]**2+Var[2]**2)**(3/2))+(Cn[3]*Cn[2]*(Var[4]-
Var[0]))/((Var[4]-Var[0])**2+(Var[6]-Var[2])**2)**(3/2)

def dy1dt(Cn,Var):
    return Var[3]


def dBdt(Cn,Var):
    return -
1*Cn[3]*Cn[0]*(Var[2]/(Var[0]**2+Var[2]**2)**(3/2))+(Cn[3]*Cn[2]*(Var[6]-
Var[2]))/((Var[4]-Var[0])**2+(Var[6]-Var[2])**2)**(3/2)

def dx2dt(Cn,Var):
    return Var[5]

def dCdt(Cn,Var):
    return -1*Cn[3]*Cn[0]*(Var[4]/(Var[4]**2+Var[6]**2)**(3/2))-
(Cn[3]*Cn[1]*(Var[4]-Var[0]))/((Var[4]-Var[0])**2+(Var[6]-
Var[2])**2)**(3/2)

def dy2dt(Cn,Var):
    return Var[7]

def dDdt(Cn,Var):
    return -1*Cn[3]*Cn[0]*(Var[6]/(Var[4]**2+Var[6]**2)**(3/2))-
(Cn[3]*Cn[1]*(Var[6]-Var[2]))/((Var[4]-Var[0])**2+(Var[6]-
Var[2])**2)**(3/2)

#create an array of the functions
drdt = [dx1dt,dAdt,dy1dt,dBdt,dx2dt,dCdt,dy2dt,dDdt]
```

```
#use the initial distances away to calculate the initial velocities
a1 = 2.52*1.5*10**11
a2 = 5.24*1.5*10**11

P1 = ((4*np.pi**2*a1**3)/(G*(m1+M)))**0.5
P2 = ((4*np.pi**2*a2**3)/(G*(m2+M)))**0.5

v1 = (G*(m1+M)*((2/a1) - (1/a1)))**0.5
v2 = (G*(m2+M)*((2/a2) - (1/a2)))**0.5

#create an array of the initial conditions (keeping the order the same as
defined above)
#x10, dx1dt0, y10, dy1dt0, x20, dx2dt0, y20, dy20dt, t0
iniConds = [a1,0,0,v1,a2,0,0,v2,0]

#arguments(array of functions, array of initial conditions, array of
constants, time to integrate to, number of points)
def Rk4(dRdt,Ics,Cn,t_f,N):

    n=len(dRdt) #find the number of equations
    dt=(t_f-Ics[n])/N #calculate step size
    dt_min=dt/(100) #use the intial stepsize to calculate a minimum
acceptable stepsize

    #two 2D arrays to store the values for all the equations and time from
0 to t_f
    Str1 = np.zeros((n+1, 10))
    Str2 = np.zeros((n+1, 10))

#populate the first value of each of the equations with the initial
conditions for the first array
    for i in range(0,n+1):
        Str1[i][0] = Ics[i]

    #define a new function to calculate the value of every functions at a
time h later
    #arguments(integer position, stepsize, array of equations, number of
equations)
    def step(i,h,Str,n):

        #create arrays to store all 4 k values for each of the equations
        k1_= np.zeros(n)
        k2_= np.zeros(n)
        k3_= np.zeros(n)
        k4_= np.zeros(n)

        #create an array to store the input values for k1 function
        Var1 = np.zeros(n+1)

        #populate the array with stored values from the functions
        for k in range(0,n+1):
            Var1[k] = Str[k][i-1]

        #calculate the k1 constants for each function
        for j in range(0,n):
            k1_[j]=h*dRdt[j](Cn,Var1)
        #repeat as above but for the k2 constants
        Var2 = np.zeros(n+1)

        for k in range(0,n):
            Var2[k] = Str[k][i-1] + k1_[k]/2

        Var2[n] = Str[n][i-1] + h/2

        for j in range(0,n):
            k2_[j]=h*dRdt[j](Cn,Var2)

        #repeat as above but for the k3 constants
        Var3 = np.zeros(n+1)

        for k in range(0,n):
            Var3[k] = Str[k][i-1] + k2_[k]/2

        Var3[n] = Str[n][i-1] + h/2

        for j in range(0,n):
            k3_[j]=h*dRdt[j](Cn,Var3)

        #repeat as above but for the k4 constants
        Var4 = np.zeros(n+1)

        for k in range(0,n):
            Var4[k] = Str[k][i-1] + k3_[k]

        Var4[n] = Str[n][i-1] + h

        for j in range(0,n):
            k4_[j]=h*dRdt[j](Cn,Var4)


        #use the k constants to calculate the next values in each of the
functions
        for j in range(0,n):
            Str[j][i] = Str[j][i-1] + 1/6*(k1_[j]+2*k2_[j]+2*k3_[j]+k4_[j])

        #step forward one stepseize in time
        Str[n][i] = Str[n][i-1]+h

        #return the updated equations array
        return Str

    c = 1 #set counter to 1
    time = dt #set time to one stepsize (first time after the initial
conditions)
    tol = 5000000 #set the error tolerance

    #create an empty array used to prevent function getting trapped at the
minimum stepsize
    t_check = np.zeros(3)

    #continue to iterate unitl the time has reached the final time set by
the user
    while time < t_f:
        #update the check array so that it contains the 3 most recent times
        t_check[2] = t_check[1]
        t_check[1] = t_check[0]
        t_check[0] = time

        #if the counter is up to the limit of the size storage array then
increase the size of the array
        if c == Str1.shape[1] - 1:

            Str3 = np.zeros((n+1, Str1.shape[1] + 10))
            Str3[:,0:Str1.shape[1]] = Str1[:,:]
            Str1 = Str3.copy()

        #take two steps in time each of size dt
        Str1 = step(c,dt,Str1,n)
        Str1 = step(c+1,dt,Str1,n)

        #calculate the distance from the sun for both planets
        r1_1 = (Str1[0][c+1]**2 + Str1[2][c+1]**2)**0.5
        r1_2 = (Str1[4][c+1]**2 + Str1[6][c+1]**2)**0.5

        Str2 = Str1.copy()

        #take one step in time of size 2*dt
        Str2 = step(c,2*dt,Str2,n)
        #print(Str2[0][c])

        #calculate the  new distance from the sun for both planets
        r2_1 = (Str2[0][c]**2 + Str2[2][c]**2)**0.5
        r2_2 = (Str2[4][c]**2 + Str2[6][c]**2)**0.5

        #use the two distances to calculate the error for both planets
        err1 = abs(r1_1 - r2_1)/30
        err2 = abs(r1_2 - r2_2)/30

        #if the error is bigger than the set tolerance level
        if err1 > tol or err2 > tol:

            #if the stepsize is at its minimum possible value and the time
hasnt changed in 3 iterations
            #this shows that the function is trapped at a point
            if dt == dt_min and t_check[2] == t_check[0]:

                c = c+1 #increment the counter by 1
                time = time + dt #increase the time by dt

            #if half dt is less than the minimum value then set dt to the
minimum value otherwise half dt
            elif dt/2 < dt_min:
                dt=dt_min
            else:
                dt = dt/2

        else:

            c = c+1 #increment the counter by 1
            time = time + dt #increase the time by dt
            dt = dt*2 #set dt to double its value


    #use the calculated values to plot as desired
    plt.scatter(Str1[0]/(1.5*10**11),Str1[2]/(1.5*10**11), c = 
Str1[8]/(3.154*10**7), cmap='Blues')
    plt.scatter(Str1[4]/(1.5*10**11),Str1[6]/(1.5*10**11), c = 
Str1[8]/(3.154*10**7), cmap='Blues')

    clb = plt.colorbar()
    clb.set_label('time (years)', labelpad=-25, y=1.08, rotation=0)

    plt.grid()
    plt.xlabel('AU')
    plt.ylabel('AU')
    plt.plot(0,0, 'ro')
    plt.show()


Rk4(drdt,iniConds,cnsts,2*P2,10000)
```

2b)
To move the coordinate system into a frame based on the centre-of-mass firstly the fixed central mass had to be allowed to move. This meant altering the existing equations of motion to include the gravitational attraction between the orbiting masses and a moving central mass. Equations 7, 8, 9 and 10 were altered to

$$\frac{dx_1}{dt} = A, \frac{dA}{dt} = -\frac{GM(x_1-x_3)}{((x_1-x_3)^2+(y_1-y_3)^2)^{3/2}} + \frac{Gm_2(x_2-x_1)}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}} \quad (17)(18)$$

$$\frac{dy_1}{dt} = B, \frac{dB}{dt} = -\frac{GM(y_1-y_3)}{((x_1-x_3)^2+(y_1-y_3)^2)^{3/2}} + \frac{Gm_2(y_2-y_1)}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}} \quad (19)(20)$$

where x_3 and y_3 represent the x and y coordinate of the central mass. This process was repeated for equations 11, 12, 13 and 14 and finally four extra first order ODEs were added to describe the motion of the central mass.

To move into the centre-of-mass frame the position and velocity of the centre-of-mass was calculated at each discrete step in time using the equations

$$\boldsymbol{r}_{cm}(t) = \sum_i \frac{m_i}{M_{tot}} \boldsymbol{r}_i(t), \boldsymbol{v}_{cm}(t) = \sum_i \frac{m_i}{M_{tot}} \boldsymbol{v}_i(t), \quad (21)(22)$$

where $\boldsymbol{r}_{cm}$ is the position of the centre-of-mass at time t, m_i is the mass of each orbital in the system, $\boldsymbol{r}_i$ is the position of each orbital at time t, $\boldsymbol{v}_{cm}$ is the velocity of the centre-of-mass at time t and $\boldsymbol{v}_i$ is the velocity of each orbital at time t. These were then subtracted from the calculated position and velocity of each of the masses at every time step.

This allowed more complex systems to be modelled. The example chosen used data from the Alpha Centauri binary star system using Alpha Centauri A and B with an imaginary planet placed in orbit around Alpha Centauri A.

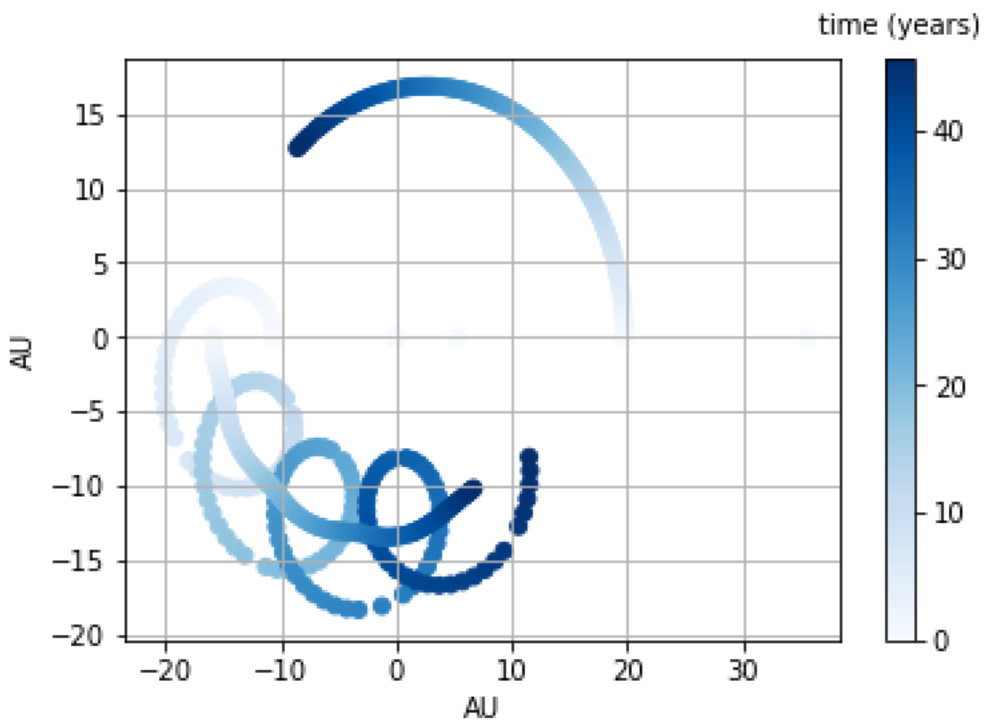

Figure 9. The binary star system of Alpha Centauri A and B plotted for four orbital periods of the exoplanet using an RK4 method with an adaptive step size.

Figure 9 shows the tracks of the two, almost solar mass, stars orbiting a common centre of mass. It also shows the four orbits of the much less massive planet orbiting Alpha Centauri A. This function could again be improved by generalising it to be able to account for any arbitrary number of masses.

Code

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants

#define all the constants to be used in all the equations
MS = 1.99*10**30
M = 1.1*MS
m1 = 0.907*MS
m2 = 4*10**-2*MS
G = scipy.constants.G
#put the constants into an array
cnsts = [M,m1,m2,G]

#define the order of the variables that will be used as if in an array
#Var = [x1,A,y1,B,x2,C,y2,D,x3,E,y3,F,t]

#define the differential equations using the array of constants and the
array of variables
def dx1dt(Cn,Var):
    return Var[1]

def dAdt(Cn,Var):
    return -Cn[3]*Cn[0]*((Var[0]-Var[8])/((Var[0]-Var[8])**2+(Var[2]-
Var[10])**2)**(3/2))+(Cn[3]*Cn[2]*(Var[4]-Var[0]))/((Var[4]-
Var[0])**2+(Var[6]-Var[2])**2)**(3/2)

def dy1dt(Cn,Var):
    return Var[3]

def dBdt(Cn,Var):
    return -1*Cn[3]*Cn[0]*((Var[2]-Var[10])/((Var[0]-Var[8])**2+(Var[2]-
Var[10])**2)**(3/2))+(Cn[3]*Cn[2]*(Var[6]-Var[2]))/((Var[4]-
Var[0])**2+(Var[6]-Var[2])**2)**(3/2)

def dx2dt(Cn,Var):
    return Var[5]

def dCdt(Cn,Var):
    return -1*Cn[3]*Cn[0]*((Var[4]-Var[8])/((Var[4]-Var[8])**2+(Var[6]-
Var[10])**2)**(3/2))-(Cn[3]*Cn[1]*(Var[4]-Var[0]))/((Var[4]-
Var[0])**2+(Var[6]-Var[2])**2)**(3/2)

def dy2dt(Cn,Var):
    return Var[7]

def dDdt(Cn,Var):
    return -1*Cn[3]*Cn[0]*((Var[6]-Var[10])/((Var[4]-Var[8])**2+(Var[6]-
Var[10])**2)**(3/2))-(Cn[3]*Cn[1]*(Var[6]-Var[2]))/((Var[4]-
Var[0])**2+(Var[6]-Var[2])**2)**(3/2)

def dx3dt(Cn,Var):
    return Var[9]

def dEdt(Cn,Var):
    return Cn[3]*Cn[1]*((Var[0]-Var[8])/((Var[0]-Var[8])**2+(Var[2]-
Var[10])**2)**(3/2))+(Cn[3]*Cn[2]*(Var[4]-Var[8]))/((Var[4]-
Var[8])**2+(Var[6]-Var[10])**2)**(3/2)

def dy3dt(Cn,Var):
    return Var[11]

def dFdt(Cn,Var):
    return Cn[3]*Cn[1]*((Var[2]-Var[10])/((Var[0]-Var[8])**2+(Var[2]-
Var[10])**2)**(3/2))+(Cn[3]*Cn[2]*(Var[6]-Var[10]))/((Var[4]-
Var[8])**2+(Var[6]-Var[10])**2)**(3/2)

#create an array of the functions
drdt = [dx1dt,dAdt,dy1dt,dBdt,dx2dt,dCdt,dy2dt,dDdt,dx3dt,dEdt,dy3dt,dFdt]

#use the initial distances away to calculate the initial velocities
a1 = 35.6*1.5*10**11
a2 = 5.24*1.5*10**11

P1 = ((4*np.pi**2*a1**3)/(G*(m1+M)))**0.5
P2 = ((4*np.pi**2*a2**3)/(G*(m2+M)))**0.5

v1 = (G*(m1+M)*((2/a1) - (1/a1)))**0.5
v2 = (G*(m2+M)*((2/a2) - (1/a2)))**0.5
```

```
#create an array of the initial conditions (keeping the order the same as
defined above)
#x10, dx1dt0, y10, dy1dt0, x20, dx2dt0, y20, dy20dt, x30, dx3dt0, y30,
dy3dt0, t0
iniConds = [a1,0,0,v1,a2,0,0,v2,0,0,0,0,0]

#arguments(array of functions, array of initial conditions, array of
constants, time to integrate to, number of points)
def Rk4(dRdt,Ics,Cn,t_f,N):

    n=len(dRdt) #find the number of equations
    dt=(t_f-Ics[n])/N #calculate step size
    dt_min=dt/(100) #use the intial stepsize to calculate a minimum
acceptable stepsize

    #two 2D arrays to store the values for all the equations and time from
0 to t_f
    Str1 = np.zeros((n+1, 10))
    Str2 = np.zeros((n+1, 10))

    #populate the first value of each of the equations with the initial
conditions for the first array
    for i in range(0,n+1):
        Str1[i][0] = Ics[i]

    #define a new function to calculate the value of every functions at a
time h later
    #arguments(integer position, stepsize, array of equations, number of
equations)
    def step(i,h,Str,n):

        #create arrays to store all 4 k values for each of the equations
        k1_= np.zeros(n)
        k2_= np.zeros(n)
        k3_= np.zeros(n)
        k4_= np.zeros(n)

        #create an array to store the input values for k1 function
        Var1 = np.zeros(n+1)

        #populate the array with stored values from the functions
        for k in range(0,n+1):
            Var1[k] = Str[k][i-1]

        #calculate the k1 constants for each function
        for j in range(0,n):
            k1_[j]=h*dRdt[j](Cn,Var1)

        #repeat as above but for the k2 constants
        Var2 = np.zeros(n+1)

        for k in range(0,n):
            Var2[k] = Str[k][i-1] + k1_[k]/2

        Var2[n] = Str[n][i-1] + h/2

        for j in range(0,n):
            k2_[j]=h*dRdt[j](Cn,Var2)

        #repeat as above but for the k3 constants
        Var3 = np.zeros(n+1)

        for k in range(0,n):
            Var3[k] = Str[k][i-1] + k2_[k]/2

        Var3[n] = Str[n][i-1] + h/2

        for j in range(0,n):
            k3_[j]=h*dRdt[j](Cn,Var3)

        #repeat as above but for the k4 constants
        Var4 = np.zeros(n+1)

        for k in range(0,n):
            Var4[k] = Str[k][i-1] + k3_[k]

        Var4[n] = Str[n][i-1] + h

        for j in range(0,n):
            k4_[j]=h*dRdt[j](Cn,Var4)


        #use the k constants to calculate the next values in each of the
functions
        for j in range(0,n):
            Str[j][i] = Str[j][i-1] + 1/6*(k1_[j]+2*k2_[j]+2*k3_[j]+k4_[j])

        #step forward one stepseize in time
        Str[n][i] = Str[n][i-1]+h

        #      M     m1    m2
        Mtot = Cn[0]+Cn[1]+Cn[2]

        #calculate the x and y coordinates of the centre of mass
        xcm =
(Cn[0]/Mtot)*Str[8][i]+(Cn[1]/Mtot)*Str[0][i]+(Cn[2]/Mtot)*Str[4][i]
        ycm =
(Cn[0]/Mtot)*Str[10][i]+(Cn[1]/Mtot)*Str[2][i]+(Cn[2]/Mtot)*Str[6][i]

        #move the position of all of the masses into the centre of mass
frame
        Str[0][i] = Str[0][i] - xcm
        Str[4][i] = Str[4][i] - xcm
        Str[8][i] = Str[8][i] - xcm

        Str[2][i] = Str[2][i] - ycm
        Str[6][i] = Str[6][i] - ycm
        Str[10][i] = Str[10][i] - ycm

        #calculate the x and y velocites of the centre of mass
        vxcm =
(Cn[0]/Mtot)*Str[9][i]+(Cn[1]/Mtot)*Str[1][i]+(Cn[2]/Mtot)*Str[5][i]
        vycm =
(Cn[0]/Mtot)*Str[11][i]+(Cn[1]/Mtot)*Str[3][i]+(Cn[2]/Mtot)*Str[7][i]
```

```
        #move the velocity of all of the masses into the centre of mass
frame
        Str[1][i] = Str[1][i] - vxcm
        Str[5][i] = Str[5][i] - vxcm
        Str[9][i] = Str[9][i] - vxcm

        Str[3][i] = Str[3][i] - vycm
        Str[7][i] = Str[7][i] - vycm
        Str[11][i] = Str[11][i] - vycm

        #return the updated equations array
        return Str

    c = 1 #set counter to 1
    time = dt #set time to one stepsize (first time after the initial
conditions)
    tol = 5000000 #set the error tolerance

    #create an empty array used to prevent function getting trapped at the
minimum stepsize
    t_check = np.zeros(3)

    #continue to iterate unitl the time has reached the final time set by
the user
    while time < t_f:

        #update the check array so that it contains the 3 most recent times
        t_check[2] = t_check[1]
        t_check[1] = t_check[0]
        t_check[0] = time

        #if the counter is up to the limit of the size storage array then
increase the size of the array
        if c == Str1.shape[1] - 1:

            Str3 = np.zeros((n+1, Str1.shape[1] + 10))
            Str3[:,0:Str1.shape[1]] = Str1[:,:]
            Str1 = Str3.copy()

        #take two steps in time each of size dt
        Str1 = step(c,dt,Str1,n)
        Str1 = step(c+1,dt,Str1,n)

        #calculate the distance from the sun for both planets
        r1_1 = (Str1[0][c+1]**2 + Str1[2][c+1]**2)**0.5
        r1_2 = (Str1[4][c+1]**2 + Str1[6][c+1]**2)**0.5

        Str2 = Str1.copy()

        #take one step in time of size 2*dt
        Str2 = step(c,2*dt,Str2,n)
        #print(Str2[0][c])

        #calculate the  new distance from the sun for both planets
        r2_1 = (Str2[0][c]**2 + Str2[2][c]**2)**0.5
        r2_2 = (Str2[4][c]**2 + Str2[6][c]**2)**0.5
        #use the two distances to calculate the error for both planets
        err1 = abs(r1_1 - r2_1)/30
        err2 = abs(r1_2 - r2_2)/30

        #if the error is bigger than the set tolerance level
        if err1 > tol or err2 > tol:

            #if the stepsize is at its minimum possible value and the time 
hasnt changed in 3 iterations
            #this shows that the function is trapped at a point
            if dt == dt_min and t_check[2] == t_check[0]:

                c = c+1 #increment the counter by 1
                time = time + dt #increase the time by dt

            #if half dt is less than the minimum value then set dt to the 
minimum value otherwise half dt
            elif dt/2 < dt_min:
                dt=dt_min
            else:
                dt = dt/2

        else:

            c = c+1 #increment the counter by 1
            time = time + dt #increase the time by dt
            dt = dt*2 #set dt to double its value


    #use the calculated values to plot as desired
    plt.scatter(Str1[0]/(1.5*10**11),Str1[2]/(1.5*10**11), c = 
Str1[12]/(3.154*10**7), cmap='Blues')
    plt.scatter(Str1[4]/(1.5*10**11),Str1[6]/(1.5*10**11), c = 
Str1[12]/(3.154*10**7), cmap='Blues')
    plt.scatter(Str1[8]/(1.5*10**11),Str1[10]/(1.5*10**11), c = 
Str1[12]/(3.154*10**7), cmap='Blues')

    clb = plt.colorbar()
    clb.set_label('time (years)', labelpad=-25, y=1.08, rotation=0)

    plt.grid()
    plt.xlabel('AU')
    plt.ylabel('AU')
    #plt.plot(0,0, 'ro')
    plt.show()


Rk4(drdt,iniConds,cnsts,P2*4,10000)
```