Steven Wray

Physics 4610

3/6/12

**Improved Methods for ODEs**

***Introduction***

This project introduces the fourth order Runge-Kutta and adaptive Runge-Kutta methods for the numerical solutions of differential equations. These approaches are applied to a system described by Kepler’s Laws and used to model the orbit of a comet.

***Activities***

3.2: Show that the Euler-Cromer method exactly preserves angular momentum for the Kepler problem.

In general, angular momentum is described by the relation .

Suppose the coordinate system is constructed so that the comet’s orbit lies strictly in the

-plane. So, on the th step, the position of the comet is given by , and the velocity is given by .

Then, at this step, the angular momentum of the comet is:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

In the Kepler problem, acceleration is . Therefore, applying the Euler-Cromer method to the problem gives:

Therefore, on step , the angular momentum of the comet is:

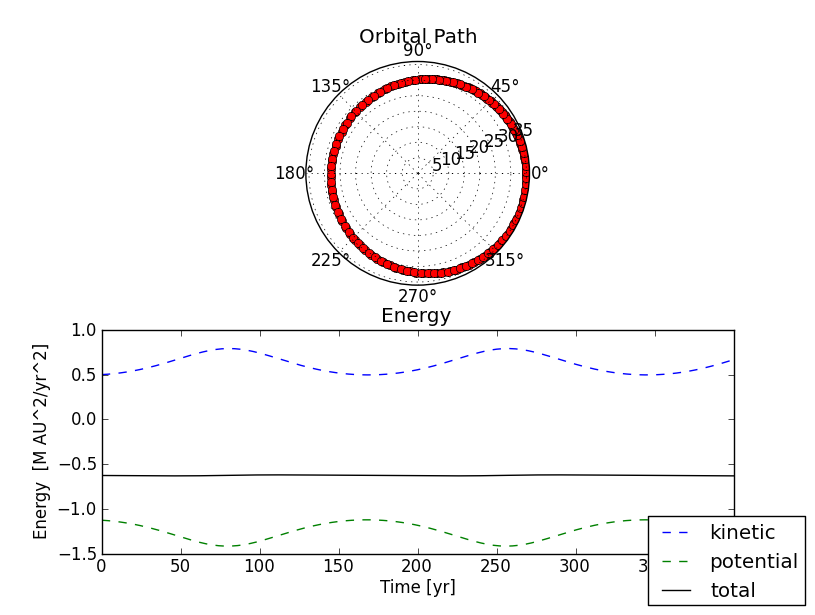
|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

So the Euler-Cromer method preserves the angular momentum of the system at each step.

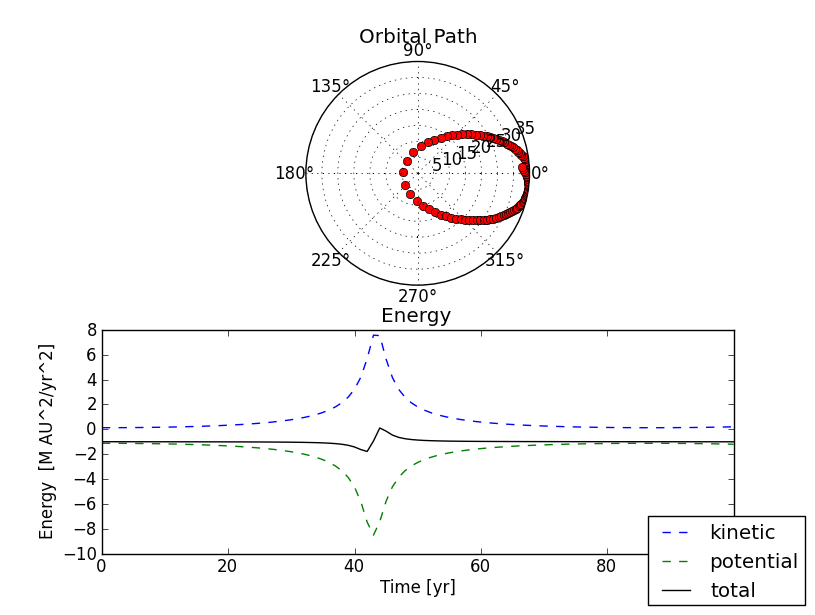
3.6/

3.11: Using an initial position of AU (Halley’s comet) and various values for the aphelion velocity, find the largest value of for which energy is conserved to about per orbit. Estimate the timestep needed to track Halley’s comet. Give estimates for both Euler-Cromer and Runge-Kutta approximations.

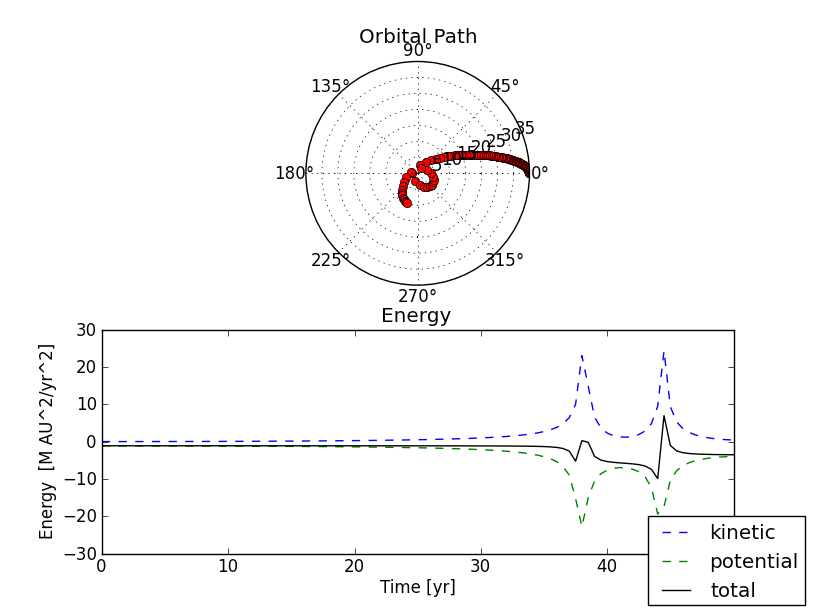
Using the Euler-Cromer method with an aphelion velocity of AU/yr and a timestep yr produced the orbit:



An aphelion velocity of AU/yr and yr gave the result:



Finally, an aphelion velocity of AU/yr and resulted in:



In this last orbit, the Euler-Cromer method does not producing accurate results. Even reducing the timestep by a factor of two did not eliminate the aberrations. This indicates that the Euler-Cromer method is not practical for very eccentric orbits.

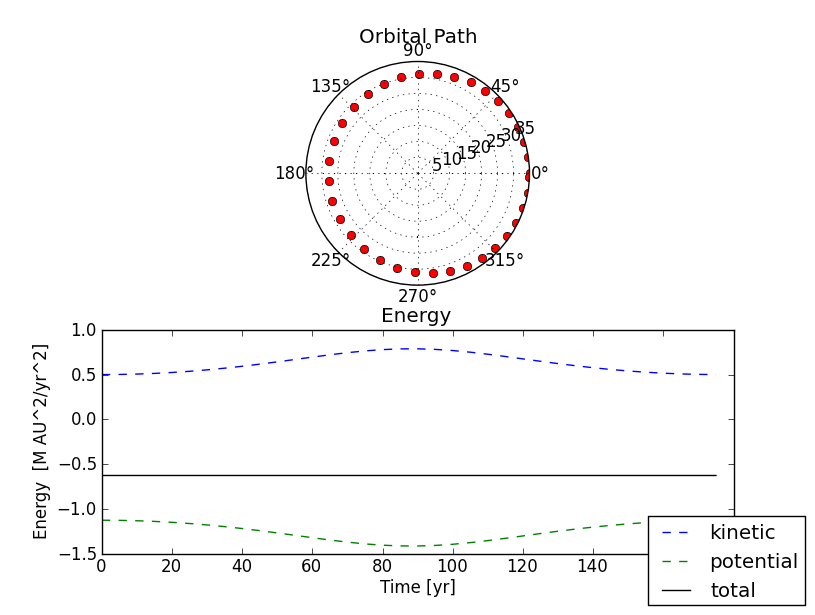
The following table shows the maximum timestep that will conserve energy to within per orbit for the Euler-Cromer method.

|  |  |
| --- | --- |
| **Apehlion velocity [AU/yr]** | **Maximum [yr]** |
| 1.0 | 42 |
| 0.9 | 44 |
| 0.8 | 46 |
| 0.7 | 47 |
| 0.6 | 49 |
| 0.5 | 50 |
| 0.4 | 50 |
| 0.3 | 51 |
| 0.2 | 52 |

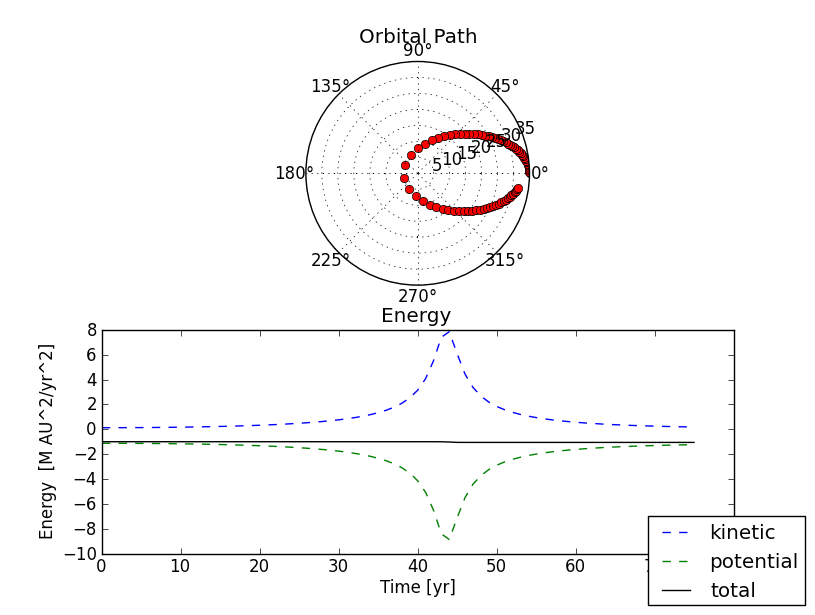
The relationship between these quantities is demonstrated in the following chart:

The chart indicates that, for a smaller aphelion velocity, a larger timestep will provide an acceptable estimate. I do not believe this conclusion is accurate. See discussion under *Conclusions*, below.

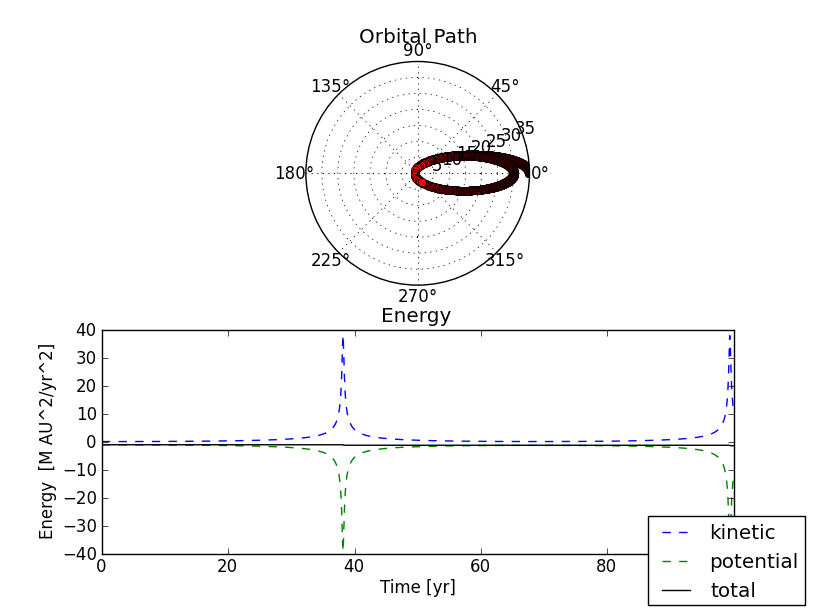
Orbits were also plotted using the Runge-Kutta method. Using an aphelion velocity of AU/yr and a timestep of yr gave the orbit:



For an aphelion velocity of AU/yr and a timestep of yr, the orbit was:



Finally, in order to get a somewhat accurate orbit for an aphelion velocity of AU/yr, it was necessary to use a timestep of yr.



For a given value of , the Runge-Kutta method will generally produce a more accurate orbit than the Euler-Cromer method. The value of needed for conservation of energy to within is outlined in the following table:

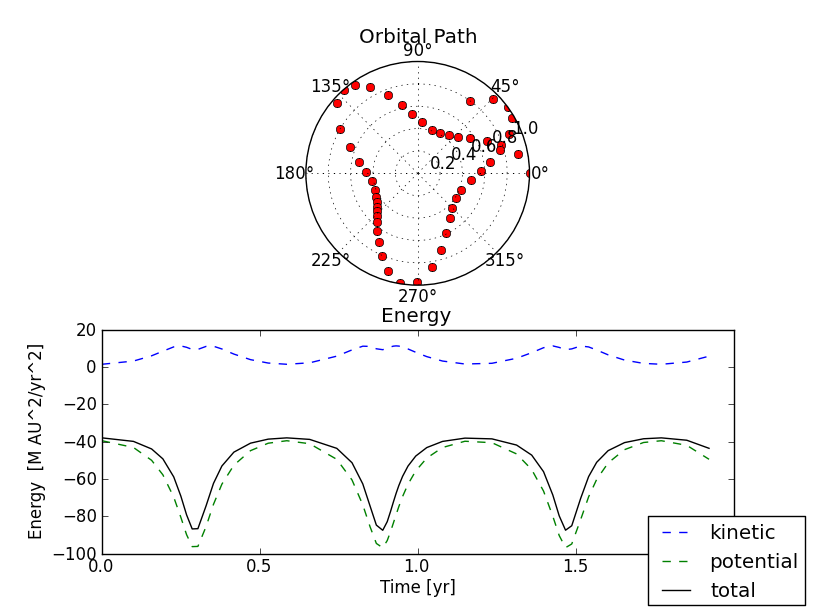
|  |  |
| --- | --- |
| **Apehlion velocity [AU/yr]** | **Maximum [yr]** |
| 1.0 | 60 |
| 0.9 | 56 |
| 0.8 | 45 |
| 0.7 | 59 |
| 0.6 | 38 |
| 0.5 | 23 |

Plotting these values shows the following relationship:

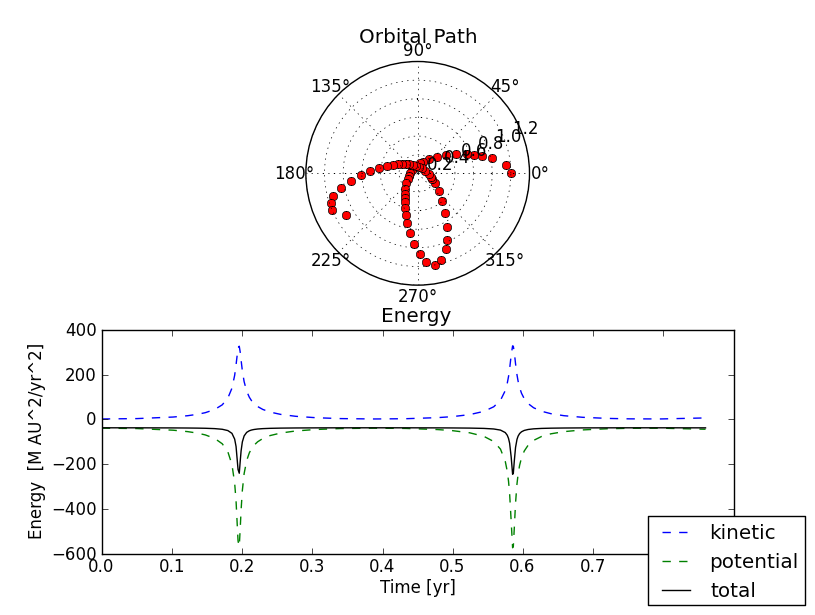
This graph does not show a clear relationship between and the aphelion velocity. It is not clear how to extrapolate these results to an initial velocity of AU/yr. If a linear extrapolation is used, a timestep of about years is needed. Once again, I believe that the data for the relationship is in error. (Further discussion below.)

3.14: Let the central force be defined by the equation . Use the adaptive Runge-Kutta method to compute the orbit of a comet under this force law. Show that the orbit processes by degrees per revolution, where and is the angular momentum of the comet.

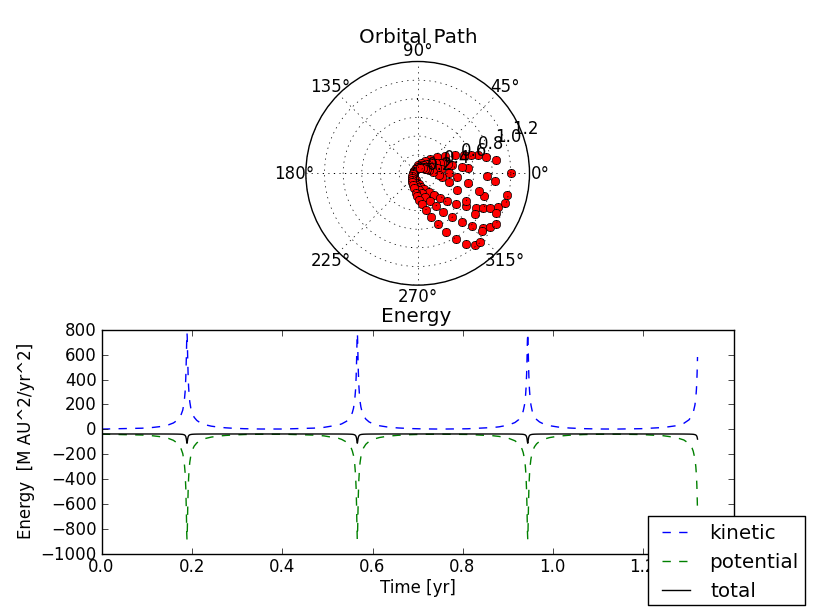
Consider a comet that begins at an initial distance of AU and an initial velocity of AU/yr. If the precession constant in the equation given above is set to , the following orbit is produced:



Using the same starting position and starting velocity with produces:



And using gives:



As gets smaller, the amount of precession that occurs on each orbit is also getting smaller.

In terms of the relationship between and described in the problem statement, the three orbits above give the results:

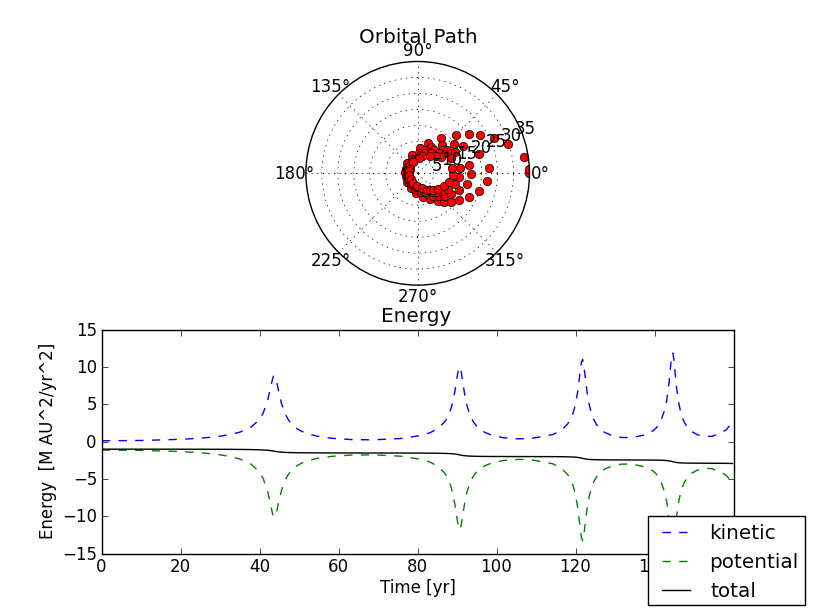
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precession (measured) |  | Precession (predicted) | Relative error |
| 0.5 | 131.53 | 2.729 | -228.07 | 0.3 % |
| 0.05 | -78.76 | 1.282 | -79.28 | 0.7 % |
| 0.0075 | -13.74 | 1.047 | -16.23 | 16 % |

The substitution is used in the table in order to calculating the relative error. The rate of precession predicted by the formula matches the measured precession very closely for the first two values of . The relative error is much larger for the third, very small value of . Because is so small in this case, I believe that round off error is having a larger effect on the results.

3.15: Add a drag force on the comet, . Fix the constant so that

, where and are the initial position and velocity. Show that the kinetic energy of the comet increases over time.

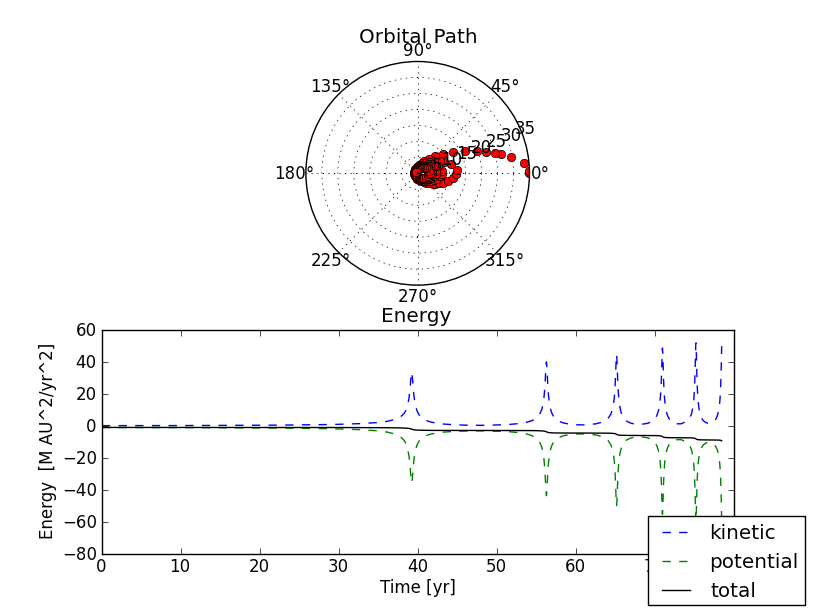
Consider a comet starting at a distance of AU and a velocity of AU/yr. The amount of drag given above produces the orbit:



The comet is spiraling inwards towards the sun. As the orbit decays, the average kinetic enery of the comet increases. For the first five orbits of the comet above, the average kinetic energy is given in the following table:

|  |  |
| --- | --- |
| **Orbit** | **Average KE** |
| 1 | 3.025 |
| 2 | 3.863 |
| 3 | 4.386 |
| 4 | 4.949 |
| 5 | 5.941 |

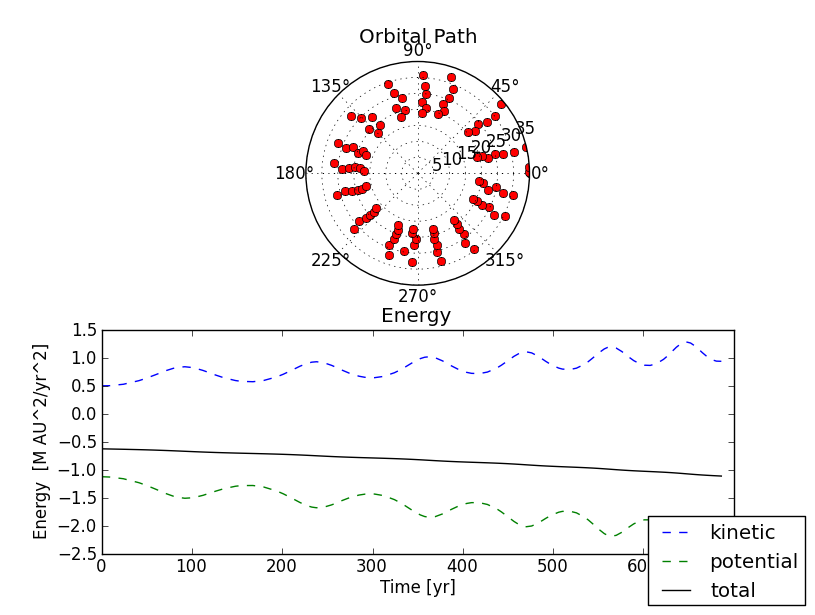
Decreasing the starting velocity of the comet will make the orbit spiral inward more tightly. Using the same initial position as above with an initial velocity of AU/yr gives:



The average kinetic energy for this comet is:

|  |  |
| --- | --- |
| **Orbit** | **Average KE** |
| 1 | 14.317 |
| 2 | 22.093 |
| 3 | 26.215 |
| 4 | 35.010 |
| 5 | 44.524 |

If the initial velocity is very large, the comet will spiral inward less rapidly. For an initial velocity of AU/yr, the orbit is:



The average kinetic energy of this comet is:

|  |  |
| --- | --- |
| **Orbit** | **Average KE** |
| 1 | 0.687 |
| 2 | 0.773 |
| 3 | 0.890 |
| 4 | 1.000 |
| 5 | 1.132 |

***Conclusions***

Even for very small timesteps, the Euler-Cromer method does not always produce an elliptical orbit from Kepler’s equations of planetary motion. For orbits that are highly elliptical, the Runge-Kutta technique produces a superior model. The adaptive Runge-Kutta method is a particularly efficient way to achieve a numerical solution.

One problem with my results was the calculation of the maximum timestep necessary to conserve energy to within per orbit (Exercises 3.6 and 3.11). The numbers produced by my program seemed wildly off base. The maximum minimum timestep from my calculation was in the range of years. A more accurate estimate should be somewhere around years. I also believe that a more eccentric orbit will require a smaller timestep to conserve energy. My data did not show this relationship. Although I made several attempts to correct this problem, I have not been successful in identifying the error in my code.

Another improvement that I would like to make in my code involves error handling. For some parts of this model, there are initial positions and velocities that cause the program to crash. Currently, the program crashes without providing much diagnostic information. I would like to add a descriptive error message that identifies the problem and then return the user to the program’s outer loop.

**APPENDIX**

**Source Code**

**# Program to compute the orbit of a comet**

**import sys**

**import math**

**import numpy as np**

**import matplotlib.pyplot as plt**

**import matplotlib.figure as fig**

**# Set physical parameters**

**GM = 4\*(math.pi\*\*2) # Newton's gravitational constant times**

**# mass of the sun (au^3/yr^2)**

**MASS\_COMET = 1. # Mass of the comet**

**ADAPT\_ERR = 0.001 # Error constant for adaptive RK**

**MAX\_ITER = 100000 # Maximum iterations when looking for an aphelion**

**# Constants representing different numerical methods**

**EULER = 1**

**EULER\_CROMER = 2**

**RUNGE\_KUTTA = 3**

**ADAPTIVE\_RK = 4**

**# Flag to turn on force law with precession**

**PRECESSION\_FLAG = False**

**precession\_constant = 0**

**# Flag to turn on molecular drag**

**DRAG\_FLAG = False**

**drag\_constant = 0**

**def norm(v):**

**# Calculate the norm of a vector**

**# Inputs**

**# v a vector**

**#Output**

**# n vector norm as float**

**n = math.sqrt(np.dot(v,v))**

**return(n)**

**def gravrk(s, t, GM):**

**# Return right hand side of Kepler ODE**

**# Called by RK routine**

**# Inputs**

**# s state vector [r[0] r[1] v[0] v[1]]]**

**# t time (not used in this model)**

**# GM parameter GM (gravitational constant \* solar mass)**

**# Output**

**# deriv state vector [dr[0]/dt dr[1]/dt dv[0]/dt dv[1]/dt]**

**# Create position and velocity vectors**

**r = np.array([s[0], s[1]], dtype=float)**

**v = np.array([s[2], s[3]], dtype=float)**

**# Compute acceleration**

**if PRECESSION\_FLAG:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* (1 - precession\_constant/norm(r)) \* r**

**elif DRAG\_FLAG:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* r - drag\_constant \* norm(v) \* v**

**else:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* r**

**# Return state vector of derivatives**

**deriv = np.array([v[0], v[1], a[0], a[1]], dtype=float)**

**return deriv**

**def rk4(x, t, tau, derivsRK, param):**

**# 4th order Runge-Kutta integrator**

**# Input**

**# x current value of dependent variable**

**# t independent variable**

**# tau stepsize**

**# derivsRK righthand side of ODE**

**# returns dx/dt**

**# format derivsRK (x, param)**

**# param extra parameters passed to derivsRK**

**# Output**

**# xout new value of x after step of size tau**

**half\_tau = 0.5 \* tau**

**f1 = derivsRK(x, t, param)**

**xtemp = x + half\_tau \* f1**

**f2 = derivsRK(xtemp, t, param)**

**xtemp = x + half\_tau \* f2**

**f3 = derivsRK(xtemp, t, param)**

**xtemp = x + tau \* f3**

**f4 = derivsRK(xtemp, t, param)**

**xout = x + (tau/6.) \* (f1 + f4 + 2.\*(f2+f3))**

**return xout**

**def rka(x, t, tau, err, derivsRK, param):**

**# Adaptive Runge-Kutta routine**

**# Inputs**

**# x current value of dependent variable**

**# t independent variable (time)**

**# tau step size**

**# err desired local truncation error**

**# derivsRK righthand side of ODE**

**# returns dx/dt**

**# param extra parameters passed to derivsRK**

**# Outputs**

**# x\_small new value of dependent variable**

**# t new value of independent variable**

**# tau suggested step size for next call to rka**

**# Save initial values**

**t\_save = t**

**x\_save = x**

**# Safety factors**

**safe1 = 0.9**

**safe2 = 4.**

**# Loop over max attempts to satisfy error bound**

**max\_try = 100**

**for i in range(max\_try):**

**# Take two small time steps**

**half\_tau = 0.5 \* tau**

**x\_temp = rk4(x\_save, t\_save, half\_tau, derivsRK, param)**

**t = t\_save + half\_tau**

**x\_small = rk4(x\_temp, t, half\_tau, derivsRK, param)**

**# Take single big time step**

**t = t\_save + tau**

**x\_big = rk4(x\_save, t\_save, tau, derivsRK, param)**

**# Compute the estimated truncation error**

**error\_ratio = 0**

**eps = 10 \*\* (-16)**

**for j in range(x\_big.size):**

**scale = err \* (abs(x\_small[j]) + abs(x\_big[j])) / 2.**

**x\_diff = x\_small[j] - x\_big[j]**

**ratio = abs(x\_diff) / (scale + eps)**

**if ratio > error\_ratio:**

**error\_ratio = ratio**

**# Estimate new tau value**

**tau\_old = tau**

**tau = safe1 \* tau\_old \* error\_ratio \*\* (-0.20)**

**tau = max(tau, tau\_old/safe2)**

**tau = min(tau, safe2\*tau\_old)**

**# If error is acceptable, return computed values**

**if error\_ratio < 1:**

**return x\_small, t, tau**

**# Issue error message if bound is never satisfied**

**print("Error: Adaptive RK routine failed.")**

**sys.exit()**

**def orbit(initial\_distance, initial\_velocity, n\_iter, tau, \**

**numerical\_method):**

**# Calculate the orbit of a comet around the sun**

**# Produce a list of polar coordinates describing the**

**# comet's position. Also produce a list of kinetic and**

**# potential energy for the points of the orbit.**

**# Inputs**

**# initial\_distance starting position of comet (AU)**

**# initial\_velocity starting velocity (AU/yr)**

**# n\_iter total data points to calculate -**

**# if n\_iter = -1, compute orbit to**

**# next aphelion**

**# tau timestep (yr)**

**# numerical\_method one of:**

**# EULER**

**# EULER\_CROMER**

**# RUNGE\_KUTTA**

**# ADAPTIVE\_RK**

**# Outputs - arrays of values. One value for each timestep.**

**# radius\_list polar coordinate R**

**# theta\_list polar coordinate theta**

**# kinetic\_list kinetic energy**

**# potential\_list gravitational potential energy**

**global drag\_constant**

**time = 0**

**# Create vectors for distance, velocity, acceleration**

**r = np.array([initial\_distance, 0.0], dtype=float)**

**v = np.array([0.0, initial\_velocity], dtype=float)**

**a = np.array([0.0, 0.0], dtype=float)**

**state = np.array([r[0], r[1], v[0], v[1]], dtype=float)**

**# Lists of polar coordinates used in plotting**

**radius\_list = [norm(r)]**

**theta\_list = [math.atan2(r[1], r[0])]**

**# Lists of time/energy values for plotting**

**time\_list = [time]**

**kinetic\_list = [0.5 \* MASS\_COMET \* norm(v)\*\*2]**

**potential\_list = [-1 \* GM \* MASS\_COMET / norm(r)]**

**i = 1 # Total data points calculated**

**if DRAG\_FLAG:**

**initial\_gravity = norm((-1 \* GM / (norm(r)\*\*3)) \* r)**

**drag\_constant = initial\_gravity/(100 \* norm(v) \*\* 2)**

**while True:**

**# Calculate the values for the next step using selected**

**# numerical method**

**if numerical\_method == EULER:**

**if PRECESSION\_FLAG:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* (1 - precession\_constant/norm(r)) \* r**

**elif DRAG\_FLAG:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* r - drag\_constant \* norm(v) \* v**

**else:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* r**

**r = r + tau\*v**

**v = v + tau\*a**

**time = time + tau**

**elif numerical\_method == EULER\_CROMER:**

**if PRECESSION\_FLAG:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* (1 - precession\_constant/norm(r)) \* r**

**elif DRAG\_FLAG:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* r - drag\_constant \* norm(v) \* v**

**else:**

**a = (-1 \* GM / (norm(r)\*\*3)) \* r**

**v = v + tau\*a**

**r = r + tau\*v**

**time = time + tau**

**elif numerical\_method == RUNGE\_KUTTA:**

**state = rk4(state, time, tau, gravrk, GM)**

**r = [state[0], state[1]]**

**v = [state[2], state[3]]**

**time = time + tau**

**else: # ADAPTIVE\_RK**

**state, time, tau = rka(state, time, tau, ADAPT\_ERR, gravrk, GM)**

**r = [state[0], state[1]]**

**v = [state[2], state[3]]**

**# Write new values to list**

**radius\_list.append(norm(r))**

**theta\_list.append(math.atan2(r[1], r[0]))**

**time\_list.append(time)**

**kinetic\_list.append(0.5 \* MASS\_COMET \* norm(v)\*\*2)**

**potential\_list.append(-1 \* GM \* MASS\_COMET/ norm(r))**

**i = i + 1**

**if n\_iter == -1:**

**# If we have reached aphelion, break out of loop**

**if i >= 3 and \**

**radius\_list[-1] - radius\_list[-2] < 0 and \**

**radius\_list[-2] - radius\_list[-3] > 0:**

**break**

**if i > MAX\_ITER:**

**# If we have not returned to apehlion at this point,**

**# the orbit is probably not closed**

**radius\_list[0] = -1 # Set flag for hyperbolic orbit**

**break**

**else:**

**if i > n\_iter:**

**break**

**return(radius\_list, theta\_list, time\_list, kinetic\_list, potential\_list)**

**def plot\_orbit():**

**# Plot the orbit and energy for an arbitrary comet.**

**global precession\_constant**

**# Get initial position and velocity from user**

**s = input("Enter initial radial distance (AU): ")**

**initial\_distance = float(s)**

**s = input("Enter initial tangential velocity (AU/yr): ")**

**initial\_velocity = float(s)**

**# Get total interations and timestep from user**

**s = input("Enter number of steps: ")**

**n\_iter = int(s)**

**s = input("Enter time step (yr): ")**

**tau = float(s)**

**if PRECESSION\_FLAG:**

**# Get precession constant from user**

**print("Enter constant alpha for force calculation")**

**s = input("(alpha must be between 0 and the initial radius): ")**

**precession\_constant = float(s)**

**print("Using adaptive Runge-Kutta method to calculate orbit.")**

**numerical\_method = ADAPTIVE\_RK**

**else:**

**# Get numerical method from user**

**print("Numerical methods:")**

**print("\t1. Euler")**

**print("\t2. Euler-Cromer")**

**print("\t3. Runge-Kutta")**

**print("\t4. Adaptive RK")**

**s = input("Enter numerical method (1-4): ")**

**numerical\_method = int(s)**

**r\_list, theta\_list, time\_list, kinetic\_list, potential\_list = \**

**orbit(initial\_distance, initial\_velocity, n\_iter, tau, \**

**numerical\_method)**

**# Create list for total energy plot**

**total\_energy\_list = []**

**for i in range(len(kinetic\_list)):**

**total\_energy\_list.append(kinetic\_list[i] + potential\_list[i])**

**# Plot orbit**

**plt.figure(1)**

**plt.subplot(211, polar='True')**

**plt.plot(theta\_list, r\_list, 'ro')**

**plt.title('Orbital Path')**

**plt.subplot(212)**

**line1 = plt.plot(time\_list, kinetic\_list, 'b--')**

**line2 = plt.plot(time\_list, potential\_list, 'g--')**

**line3 = plt.plot(time\_list, total\_energy\_list, 'k')**

**plt.ylabel('Energy [M AU^2/yr^2]')**

**plt.xlabel('Time [yr]')**

**plt.figlegend((line1, line2, line3), ('kinetic', 'potential', \**

**'total'), 'lower right')**

**plt.title('Energy')**

**plt.show()**

**print("\n")**

**def find\_timestep():**

**# Get input from user**

**print("\nUsing a comet with an aphelion of 35 AU.")**

**print("Based on aphelion of Halley's comet.")**

**s = input("Enter velocity at aphelion (AU/yr): ")**

**initial\_velocity = float(s)**

**print("Numerical methods:")**

**print("\t1. Euler-Cromer")**

**print("\t2. Runge-Kutta")**

**s = input("Enter numerical method (1-2): ")**

**if s == 1:**

**numerical\_method = EULER\_CROMER**

**else:**

**numerical\_method = RUNGE\_KUTTA**

**# Calculate range for 30 years to 1 year**

**for i in range (60,0,-1):**

**tau = i**

**r\_list, theta\_list, time\_list, kinetic\_list, potential\_list = \**

**orbit(35., initial\_velocity, -1, tau, numerical\_method)**

**if r\_list[0] == -1: # Calculated orbit is not closed**

**continue**

**initial\_energy = kinetic\_list[0] + potential\_list[0]**

**final\_energy = kinetic\_list[-2] + potential\_list[-2]**

**if abs(initial\_energy - final\_energy)/initial\_energy < 0.01:**

**s = '\nTimestep of ' + str(tau) + ' years required for 1% error.\n'**

**print(s)**

**return**

**# Calculate loss of energy for fractions of a year**

**for i in range(1,6):**

**for j in range (9,0,-1):**

**tau = j \* 10\*\*(-1 \* i)**

**r\_list, theta\_list, time\_list, kinetic\_list, potential\_list = \**

**orbit(35, initial\_velocity, -1, tau, numerical\_method)**

**if r\_list[0] == -1: # Calculated orbit is not closed**

**continue**

**initial\_energy = kinetic\_list[0] + potential\_list[0]**

**final\_energy = kinetic\_list[-1] + potential\_list[-1]**

**if abs(initial\_energy - final\_energy)/initial\_energy < 0.01:**

**s = '\nTimestep of ' + str(tau) + ' years required for 1% error.\n'**

**print(s)**

**return**

**print("\nTimestep smaller than 10^-5 required or orbit not closed.\n")**

**return**

**def plot\_orbit\_precession():**

**global PRECESSION\_FLAG**

**PRECESSION\_FLAG = True**

**plot\_orbit()**

**PRECESSION\_FLAG = False**

**precession\_constant = 0**

**def calculate\_precession():**

**global PRECESSION\_FLAG**

**global precession\_constant**

**PRECESSION\_FLAG = True**

**# Get initial position and velocity from user**

**s = input("Enter initial radial distance (AU): ")**

**initial\_distance = float(s)**

**s = input("Enter initial tangential velocity (AU/yr): ")**

**initial\_velocity = float(s)**

**# Get timestep from user**

**s = input("Enter time step (yr): ")**

**tau = float(s)**

**# Get precession constant from user**

**print("Enter constant alpha for force calculation")**

**s = input("\t(alpha must be between 0 and the initial radius): ")**

**precession\_constant = float(s)**

**print("Using adaptive Runge-Kutta method to calculate orbit.\n")**

**# Calculate orbit**

**r\_list, theta\_list, time\_list, kinetic\_list, potential\_list = \**

**orbit(initial\_distance, initial\_velocity, -1, tau, \**

**ADAPTIVE\_RK)**

**# Calculate angle of precession**

**angle = math.degrees(theta\_list[-2])**

**print("Angle of precession is " + str(angle) + " degrees per revolution.")**

**L = initial\_velocity \* initial\_distance # Angular momentum**

**a = math.sqrt(1 + GM \* precession\_constant / L\*\*2)**

**print("a = Sqrt(1 + G M m^2 alpha / L^2 = " + str(a))**

**angle = 360 \* (1 - a) / a**

**print("Angle of precession = 360(1 - a)/a = " + str(angle))**

**print("\tdegrees per revolution.\n")**

**PRECESSION\_FLAG = False**

**precession\_constant = 0**

**def plot\_orbit\_drag():**

**global DRAG\_FLAG**

**DRAG\_FLAG = True**

**plot\_orbit()**

**DRAG\_FLAG = False**

**def kinetic\_energy\_drag():**

**global DRAG\_FLAG**

**DRAG\_FLAG = True**

**# Get initial position and velocity from user**

**s = input("Enter initial radial distance (AU): ")**

**initial\_distance = float(s)**

**s = input("Enter initial tangential velocity (AU/yr): ")**

**initial\_velocity = float(s)**

**# Get total interations and timestep from user**

**s = input("Enter total number of orbits to evaluate: ")**

**n\_orbits = int(s)**

**s = input("Enter time step (yr): ")**

**tau = float(s)**

**print("Using adaptive Runge-Kutta method to evaluate orbit.\n")**

**average\_kinetic\_energy = 0**

**print("Orbit\t\tAverage Kinetic Energy\n")**

**for i in range(n\_orbits):**

**r\_list, theta\_list, time\_list, kinetic\_list, potential\_list = \**

**orbit(initial\_distance, initial\_velocity, -1, tau, \**

**ADAPTIVE\_RK)**

**# Calculate average kinetic energy for the current orbit**

**total\_energy = 0**

**for j in range(len(kinetic\_list) - 2):**

**total\_energy = total\_energy + kinetic\_list[j]**

**average\_kinetic\_energy = total\_energy/j**

**print(str(i + 1) + "\t\t" + str(average\_kinetic\_energy))**

**# Set values for beginning of next orbit**

**initial\_distance = r\_list[-2]**

**initial\_velocity = math.sqrt(2 \* kinetic\_list[-2])**

**print("\n")**

**DRAG\_FLAG = False**

**def main():**

**while True:**

**# Ask user for desired function**

**print("Functions:")**

**print("\t1. Plot the orbit of an arbitrary comet.")**

**print("\t2. Find the largest timestep that conserves total energy ")**

**print("\t\tto 1% per orbit. (Exercise 3.6, 3.11)")**

**print("\t3. Plot the orbit of an arbitrary comet using a force law")**

**print("\t\tthat causes precession. ")**

**print("\t4. Measure the precession of an orbit in degress per")**

**print("\t\trevolution. (Exercise 3.14)")**

**print("\t5. Plot the orbit of an arbitrary comet experiencing")**

**print("\t\ta drag force. ")**

**print("\t6. Determine the change in average kinetic energy for a comet")**

**print("\t\texperiencing a drag force. (Exercise 3.15)")**

**print("\t7. Exit")**

**s = input("Enter a number (1-7): ")**

**n = int(s)**

**if n == 1:**

**plot\_orbit()**

**elif n==2:**

**find\_timestep()**

**elif n==3:**

**plot\_orbit\_precession()**

**elif n ==4:**

**calculate\_precession()**

**elif n==5:**

**plot\_orbit\_drag()**

**elif n==6:**

**kinetic\_energy\_drag()**

**else: # Exit program**

**break**

**main()**