Computational Astrophics

Coursework 1

Candidate Number: 24697

In [1]:
```
# Imports
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gs
import time
```

Question 1.a)

The aim of question 1 is to model the orbit of Halley's comet, according to the equation

$$m\frac{d^2\vec{r}}{dt^2} = -\left(\frac{GMm}{|\vec{r}|^2}\right)\frac{\vec{r}}{|r|} \quad (1)$$

where $\vec{r}$ is the vector from the Sun to the comet, M is the mass of the Sun, and m is the mass of the comet. It is simple to see that m cancels across the equals sign, leaving a second order ODE of the form

$$\frac{d^2\vec{r}}{dt^2} = -\left(\frac{GM}{|\vec{r}|^2}\right)\frac{\vec{r}}{|r|} \quad (2)$$

where $|\vec{r}| = \sqrt{x^2+y^2}$ in cartesian coordinates and, $\vec{r} = \begin{bmatrix} x \\ y \end{bmatrix}$.

As x and y are orthogonal we can split (2) into two equations for x and y separately. Then we further split each second order ODE into a pair of coupled first order ODEs, giving a total of four equations to solve:

$$\begin{bmatrix} \frac{dx}{dt} = v_x \\ \frac{dv_x}{dt} = -\left(\frac{GMx}{(x^2+y^2)^{\frac{3}{2}}}\right) \end{bmatrix} \quad (3)$$

$$\begin{bmatrix} \frac{dy}{dt} = v_y \\ \frac{dv_y}{dt} = -\left(\frac{GMy}{(x^2+y^2)^{\frac{3}{2}}}\right) \end{bmatrix} \quad (4)$$

In order to model the equations a vectorized, fourth order, constant step size Runge-Kutta (cRK4) method is used.

In [2]:
```python
# Single step of the 4th order Runge-Kutta method
def RK4_step(f, t, h, r):
    k1 = np.array([h * func(t, *r) for func in f])
    k2 = np.array([h * func(t + h/2, *(r + k1/2)) for func in f])
    k3 = np.array([h * func(t + h/2, *(r + k2/2)) for func in f])
    k4 = np.array([h * func(t + h, *(r + k3)) for func in f])
    return r + (1/6)*(k1 + 2*k2 + 2*k3 + k4)

# Constant step size, vectorized Runge-Kutta method
def cRK4(f, y0, t0, tf, N):
    step = abs(tf - t0) / (N - 1)
    result = [y0]
    times = [t0]

    for i in range(N):
        result.append(RK4_step(f, times[-1], step, result[-1]))
        times.append(times[-1] + step)

    return np.array(times), np.array(result)
```

RK4 takes five parameters:

- f Array of functions to numerically approximate.
- y_0 Array of initial values in the same order as their respective function in f.
- t_0 Start point of the time period to numerically integrate.
- t_f End point of the time period to numerically integrate
- N Number of steps to take.

First the step size is approximated and arrays for the time points and approximated points of the curve are set up. Then, the algorithm takes N iterations making a RK4 step each time. The new time value and point on the curve are then appended to their respective arrays. Finally, the time and result arrays are converted to numpy ndarrays and returned from the function.

Next, some constants and the coupled differential equations (3) & (4) are defined.

In [3]:
```
# Constants
s_year = 3.1536e7       # Seconds in a year
G = 6.6743e-11          # Gravitational constant
G_year = G * s_year**2 # Gravitational constant using years for time unit
M = 1.989e30            # Mass of Sun in kg
GM_year = G_year * M    # Combine G_year & M into single constant

# Two sets of coupled eqns, 1 set each for x & y coordinates
def dX(t, x, vx, y, vy):
    return vx
def dV_X(t, x, vx, y, vy):
    return -(GM_year * x) / pow(x**2 + y**2, 1.5)

def dY(t, x, vx, y, vy):
    return vy
def dV_Y(t, x, vx, y, vy):
    return -(GM_year * y) / pow(x**2 + y**2, 1.5)
```

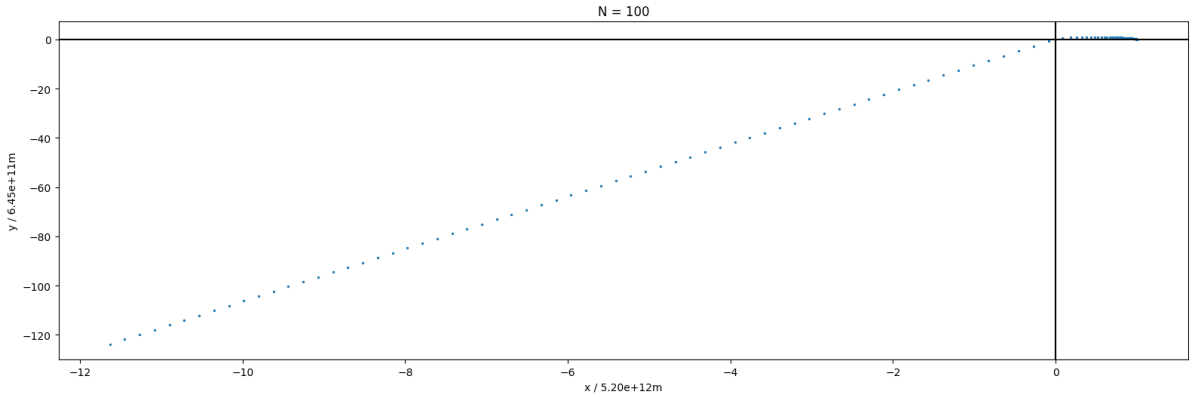
The initial conditions are set so that cRK4 can be run and then plotted. The initial conditions array is in the order $[x, v_x, y, v_y]$. Distances are measured in metres, and all time units are converted to years therefore velocities are in metres per year. It is run three times with different values of N (100, 1,000, & 10,000) over a period of 80 years. Three different values of N are used to demonstrate the improvement in accuracy as N increases and the step size decreases.

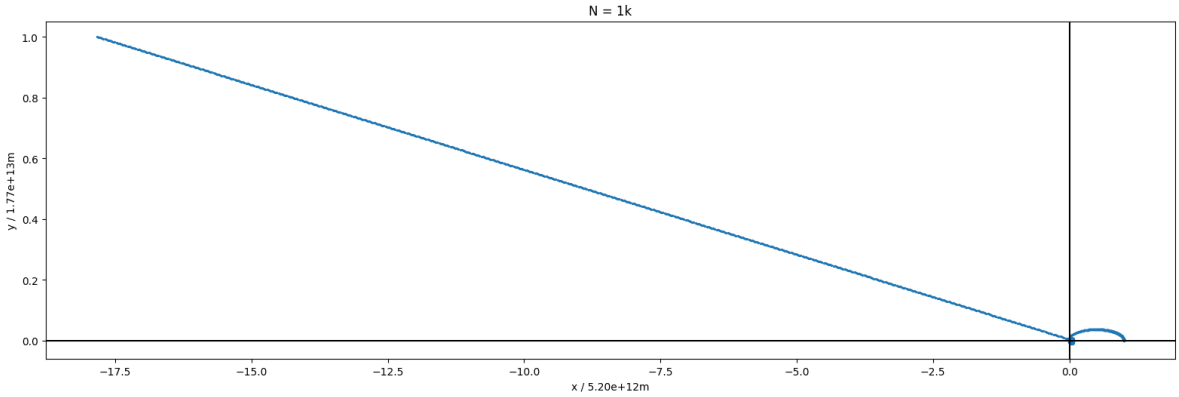
In [4]:
```
# Initial conditions
f = np.array([ dX, dV_X, dY, dV_Y ])
r = np.array([ 5.2e12, 0, 0, 2.775e10 ])

# Simulate for 80 years
# All time units converted from seconds -> years

# Apply Runge-Kutta for x values
# vals_Nk returns:
#  - Array of time arrays.
#  - Array of arrays of [x, vx, y, vy] for each time point.
print("Calculating N = 100...")
start = time.time()
t_100, vals_100 = cRK4(f, r, 0, 100, 100)
print(f"Done. ({(time.time() - start):.3f}s)")
print("Calculating N = 1k...")
start = time.time()
t_1k, vals_1k = cRK4(f, r, 0, 100, 1000)
print(f"Done. ({(time.time() - start):.3f}s)")
print("Calculating N = 10k...")
start = time.time()
t_10k, vals_10k = cRK4(f, r, 0, 100, 10000)
print(f"Done. ({(time.time() - start):.3f}s)")
```

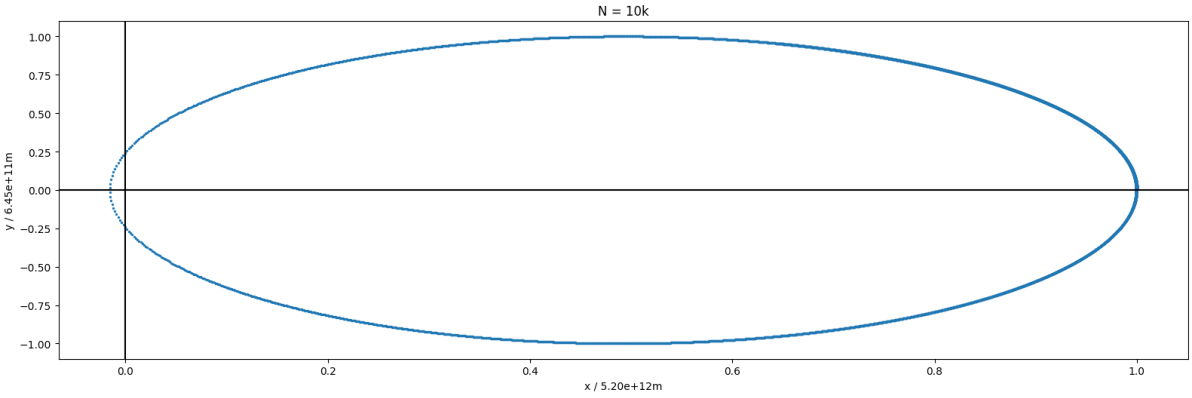
```
Calculating N = 100...
Done. (0.003s)
Calculating N = 1k...
Done. (0.025s)
Calculating N = 10k...
Done. (0.222s)
```

The results are plotted below:

In [5]:
```
def normalize(arr):
    assert(type(arr) == np.ndarray)
    return arr / max(arr)

# Plot result

fig = plt.figure(figsize=(16, 16))
grid = gs.GridSpec(3,2)

ax1 = fig.add_subplot(grid[0, :])
ax2 = fig.add_subplot(grid[1, :])
ax3 = fig.add_subplot(grid[2, :])

ax1.set_title("N = 100")
ax1.plot(normalize(vals_100[:, 0]), normalize(vals_100[:, 2]),
         marker='o', ls='none', markersize=1.4)
ax1.set_xlabel(f"x / {max(vals_100[:, 0]):.2e}m")
ax1.set_ylabel(f"y / {max(vals_100[:, 2]):.2e}m")
ax1.axhline(0, c='k')
ax1.axvline(0, c='k')

ax2.set_title("N = 1k")
ax2.plot(normalize(vals_1k[:, 0]), normalize(vals_1k[:, 2]),
         marker='o', ls='none', markersize=1.4)
ax2.set_xlabel(f"x / {max(vals_1k[:, 0]):.2e}m")
ax2.set_ylabel(f"y / {max(vals_1k[:, 2]):.2e}m")
ax2.axhline(0, c='k')
ax2.axvline(0, c='k')

ax3.set_title("N = 10k")
ax3.plot(normalize(vals_10k[:, 0]), normalize(vals_10k[:, 2]),
         marker='o', ls='none', markersize=1.4)
ax3.set_xlabel(f"x / {max(vals_10k[:, 0]):.2e}m")
ax3.set_ylabel(f"y / {max(vals_10k[:, 2]):.2e}m")
ax3.axhline(0, c='k')
ax3.axvline(0, c='k')

plt.tight_layout()
plt.show()
```


N = 100
y / 6.45e+11m
x / 5.20e+12m


N = 1k
y / 1.77e+13m
x / 5.20e+12m


N = 10k
y / 6.45e+11m
x / 5.20e+12m

Question 1.b)

In order to improve the numerical approximation in 1.a), an adaptive time step version of the 4th order Runge-Kutta method is applied. The variable Runge-Kutta (vRK4) parameters differ from cRK4 by taking arguments for the initial step size ($init\_step$) and an error scale which defines maximum allowed discrete step error based on the previous values (err_scale).

This error evaluation works as follows. First the discrete step error is calculated by evaluating the next value in the sequence from two steps of size h, and from a single step of size $2h$. The discrete error, ϵ is then

$$\epsilon = \frac{1}{30}|y_2 - y_1|$$

where y_1 and y_2 are the single and double steps respectively, and ϵ is a vector the discrete step error in each variable. Next the max allowed error for the next step is evaluated.

First the maximum values of y_1 and y_2 are found and passed to a function, pow_10, which returns the power of 10 of their most significant digit. For example, 10 -> 1, [0, 1, ..., 9] -> 0, 0.004 -> -3. The maximum error for each variable in each vector is then defined as $10^{err\_scale+log\_10(x)}$, where x is the maximum value of that variable between y_1 and y_2.

If any values of ϵ are greater than the maximum error, the step size is halved. Similarly, if the error is much smaller than the threshold error, the step size is increased after checking that it won't cause the largest error to exceed the threshold.

The new function is as follows.

In [6]:
```
def vRK4(f, y0, t0, tmax, init_step=0.001, err_scale=-6):
    if err_scale > 0:
        raise RuntimeError(f"err_scale ({err_scale}) must < 0")

    # Convert to numpy arrays
    f = np.array(f)
    y0 = np.array(y0)

    # Get array of closest powers of 10 for items in the input.
    def pow_10(x):
        result = np.zeros(x.shape)
        not_zero = np.not_equal(np.abs(x), 0)
        result[not_zero] = np.floor(np.log10(np.abs(x[not_zero])))
        return result

    # Check input shapes are compatible
    assert( np.shape(f) == np.shape(y0) )
    # Check valid init_step
    N = int(np.floor(abs(tmax - t0) / init_step))
    if N == 0:
        raise RuntimeError( f"floor(abs({tmax} (tmax) - {t0} (t0) / {init_st

    # Set initial conditions
    # Result has the structure: time, parameters for each respective y0 valu
    times, result = [t0], [y0]

    # Array of max errors for each parameter
    max_err_scale = np.array([10 ** (np.full_like(y0, err_scale) + pow_10(y0
    epsilon_arr = [y0 * np.full_like(y0, 10 ** err_scale)]

    # The current time step value is results[i - 1, 0]
    # The next t value is set to results[i, 0] when the step size is determi
    step = init_step
    i, t = 1, t0
    while t <= tmax:
        t = times[-1]
        params = result[-1]

        # Evaluate if error in step is too large, adjust step accordingly.
        err_too_large = True
        while err_too_large:
            # Two steps of size h
            y = RK4_step(f, t, step, params)
            y1 = RK4_step(f, t, step, y)
            # One step of size 2h
            y2 = RK4_step(f, t, 2 * step, params)

            # Check error is within tolerance
            maximum = np.maximum(np.abs(y1), np.abs(y2))
            max_err = 10 ** ( np.full_like(y0, err_scale) + pow_10(maximum)
            err = (1/30) * np.abs(y2 - y1)
            err_too_small = np.any(np.less(err, 0.5 * max_err))
            err_too_large = np.any(np.greater(err, max_err))

            # Adjust step size
            if err too large:
```

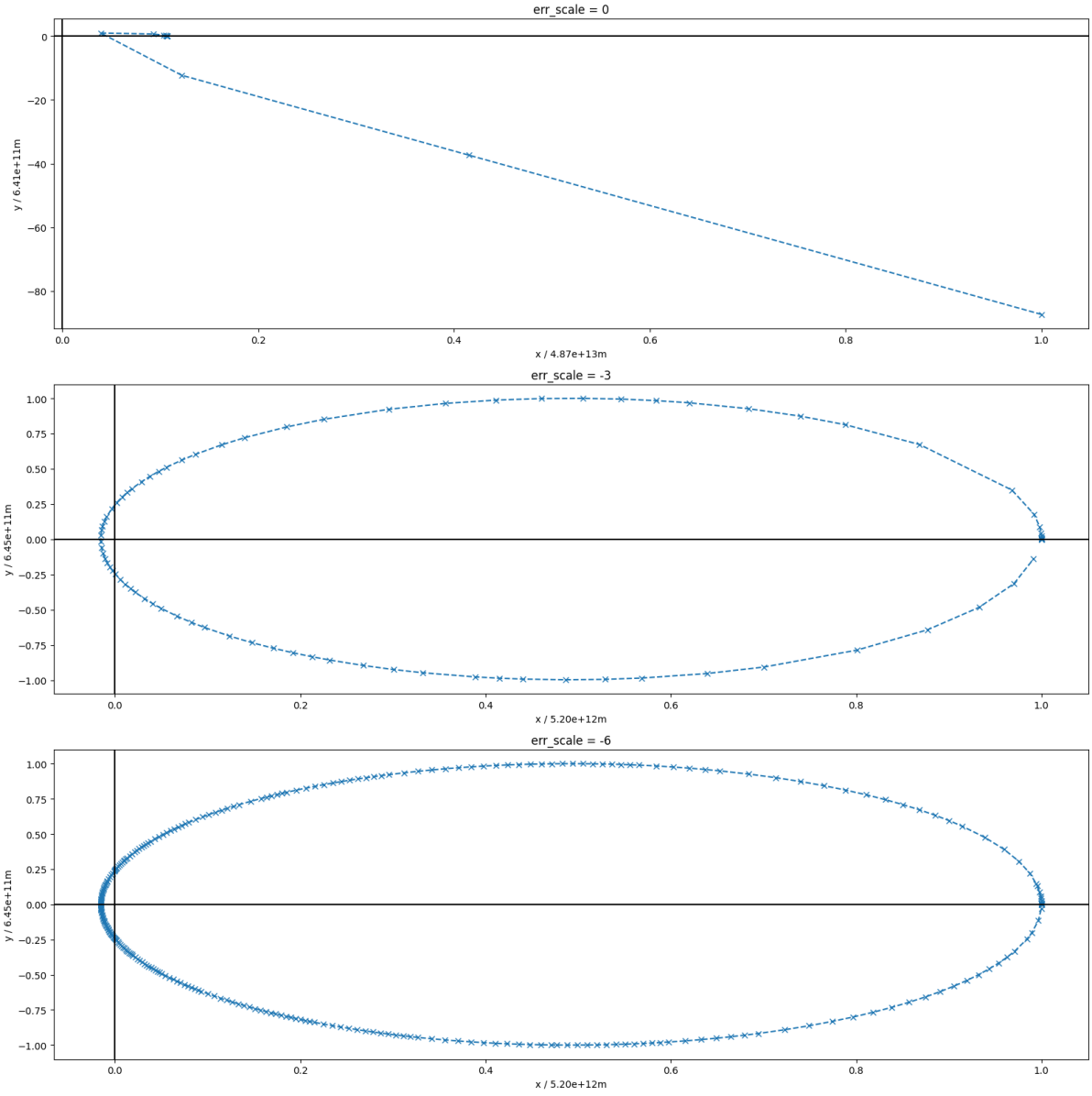
```
                step /= 2
            elif err_too_small and ~np.any(np.greater(100 * err, max_err)):
                step *= 2

        # Write result
        times.append(t + step)
        result.append(y)
        i += 1

    return np.array(times), np.array(result)
```

The same initial values and functions are used. Three levels of precision are calculated, the number of rows (N) is printed, and then the result is plotted.

In [7]:
```
print("Calculating err_scale=0...")
start=time.time()
t_0, vals_0 = vRK4(f, r, 0, 80, err_scale=0)
print("Done. ({time.time()-start:.3f}s)")
print("Calculating err_scale=-3...")
start=time.time()
t_milli, vals_milli = vRK4(f, r, 0, 80, err_scale=-3)
print("Done. ({time.time()-start:.3f}s)")
print("Calculating err_scale=-6...")
t_micro, vals_micro = vRK4(f, r, 0, 80, err_scale=-6)
print("Done. ({time.time()-start:.3f}s)")

print("\n")

print(f"err_scale = 0 -> steps taken = {len(t_0)})")
print(f"err_scale = -3 -> steps taken = {len(t_milli)}")
print(f"err_scale = -6 -> steps taken = {len(t_micro)}")

# Plot result

fig = plt.figure(figsize=(16, 16))
grid = gs.GridSpec(3,2)

ax1 = fig.add_subplot(grid[0, :])
ax2 = fig.add_subplot(grid[1, :])
ax3 = fig.add_subplot(grid[2, :])

ax1.set_title("err_scale = 0")
ax1.plot(normalize(vals_0[:, 0]), normalize(vals_0[:, 2]),
         marker='x', ls='--', markersize=6)
ax1.set_xlabel(f"x / {max(vals_0[:, 0]):.2e}m")
ax1.set_ylabel(f"y / {max(vals_0[:, 2]):.2e}m")
ax1.axhline(0, c='k')
ax1.axvline(0, c='k')

ax2.set_title("err_scale = -3")
ax2.plot(normalize(vals_milli[:, 0]), normalize(vals_milli[:, 2]),
         marker='x', ls='--', markersize=6)
ax2.set_xlabel(f"x / {max(vals_milli[:, 0]):.2e}m")
ax2.set_ylabel(f"y / {max(vals_milli[:, 2]):.2e}m")
ax2.axhline(0, c='k')
ax2.axvline(0, c='k')

ax3.set_title("err_scale = -6")
ax3.plot(normalize(vals_micro[:, 0]), normalize(vals_micro[:, 2]),
         marker='x', ls='--', markersize=6)
ax3.set_xlabel(f"x / {max(vals_micro[:, 0]):.2e}m")
ax3.set_ylabel(f"y / {max(vals_micro[:, 2]):.2e}m")
ax3.axhline(0, c='k')
ax3.axvline(0, c='k')

plt.tight_layout()
plt.show()
```

```
Calculating err_scale=0...
Done. ({time.time()-start:.3f}s)
Calculating err_scale=-3...
Done. ({time.time()-start:.3f}s)
Calculating err_scale=-6...
Done. ({time.time()-start:.3f}s)
```

```
err_scale = 0 -> steps taken = 19)
err_scale = -3 -> steps taken = 86
err_scale = -6 -> steps taken = 301
```



These graphs clearly demonstrate how the accuracy of the approximation improves with a more negative err_scale. The most important result however, is how few steps it took to get a good approximation. In the constant step version it took a number of steps of the order of a thousand to get at a good approximation, whereas we have a achieved a good approximation here with just a few hundred.

Question 2.a)

Question 2.b)