1 Initial Settings

We start by importing the libraries we will use, defining the constants, and our function f:

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

# Constants
mu = 0.012277471
T0 = 17.065216560157962558891720624 9 # Period
x_0 = 0.994
y_0 = 0
x_p_0 = 0
y_p_0 = -2.00158510637908252240537862224

# Coordinate function
def f(t, point: list):
    x = point[0]
    y = point[1]
    x_p = point[2]
    y_p = point[3]
    A = ((x+mu)**2 + y**2)**(3/2)
    B = ((x-1 + mu)**2 + y**2)**(3/2)
    # x"
    x_b = x + 2*y_p - (1-mu)*(x+mu)/A - mu*(x-1+mu)/B
    # y"
    y_b = y - 2*x_p - (1 - mu)*y/A - mu*y/B
    return np.array([x_p, y_p, x_b, y_b])
```

2 Euler and RK4 Methods

We implement the methods similarly to the examples from the exercises:

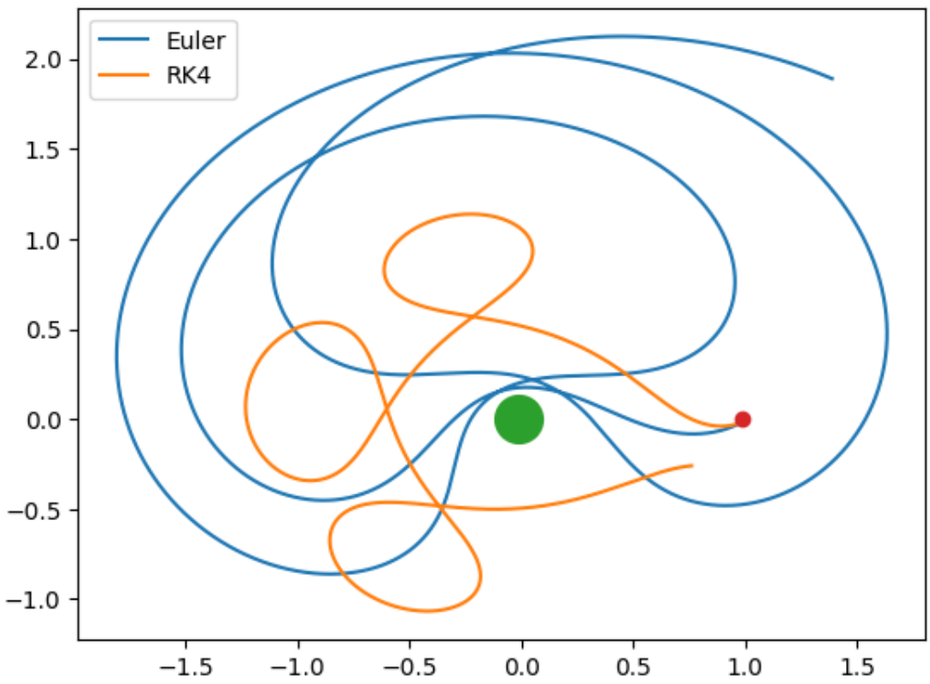
```
# Euler method
def Euler(f, a, b, x_0, N):
  h = (b-a)/N
  X = [x_0]
  T = [0]
  t = a
  x = x_0
  for i in range(N):
        x = x + f(t,x)*h
        t = t + h
        T.append(t)
        X.append(x)
  return T,X

# Runge-Kutta method
def RK4(f, a, b, x_0, N):
  h = (b-a)/N
  X = [x_0]
  T = [0]
  t = a
  x = x_0

  for i in range(N):
    K1 = f(t, x)
    K2 = f(t+h/2, x+h*K1/2)
    K3 = f(t+h/2, x+h*K2/2)
    K4 = f(t+h, x+h*K3)
    x = x + h*(K1+2*K2+2*K3+K4)/6
    t = t + h
    T.append(t)
    X.append(x)
  return T, X
```

Next, we generate the data, which we plot:

```
# Data from the Euler method
T_E, XY_E = Euler(f, 0, T0, np.array([x_0, y_0, x_p_0, y_p_0]), 24000)
XY_E = np.array(XY_E)

# Data from the Runge-Kutta method
T_RK, XY_RK = RK4(f, 0, T0, np.array([x_0, y_0, x_p_0, y_p_0]), 6000)
```

```
XY_RK = np.array(XY_RK)

# Plotting the data
plt.plot(XY_E[:,0], XY_E[:,1], label="Euler")
plt.plot(XY_RK[:,0], XY_RK[:,1], label="RK4")
plt.plot(-mu, 0,"o",markersize=30)
plt.plot(1-mu, 0, "o", markersize=10)
plt.legend(loc="best")
plt.show()
```

We obtain the following plot:



We observe that despite using fewer steps, the Runge-Kutta method is much closer to the periodic solution than the Euler method, which quickly deviates from the stable trajectory and significantly differs from the desired periodic solution.
Note: The depicted size difference between the Earth and the Moon is purely illustrative and not to scale.

3 Dormand-Prince Method

We implement the method as follows:

```
# Implementation of Dormand-Prince
def DP(f, a, b, x0, N):
    h = (b - a) / N
    T = [a]
    X = [x0]
    t = a
    x = x0
    for i in range(N):
        K1 = f(t, x)
        K2 = f(t + h / 5, x + h * (K1 / 5))
        K3 = f(t + 3 * h / 10, x + h * (K1 * 3 / 40 + K2 * 9 / 40))
        K4 = f(t + 4 * h / 5, x + h * (K1 * 44 / 45 - K2 * 56 / 15 + K3 * 32 / 9))
        K5 = f(t + 8 * h / 9, x + h * (K1 * 19372 / 6561 - K2 * 25360 / 2187 + K3 *
   64448 / 6561 - K4 * 212 / 729))
        K6 = f(t + h, x + h * (K1 * 9017 / 3168 - K2 * 355 / 33 + K3 * 46732 / 5247 + K4
    * 49 / 176 - K5 * 5103 / 18656))
        K7 = f(t+ h, x + h*(K1*35/384 + K3*500/1113 + K4*125/192 - K5*2187/6784 + K6
   *11/84))

        x = x + h * (K1 * 35 / 384 + K3 * 500 / 1113 + K4 * 125 / 192 - K5 * 2187 / 6784
    + K6 * 11 / 84)
        t = t + h

        T.append(t)
        X.append(x)

    return T, X
```

We will now test this scheme for several known examples from the exercises and laboratory, namely $x' = x \implies x = e^t$ and $x' = -2t \cdot x^2 \implies x = \frac{1}{1+t^2}$. To determine the order, we will iteratively examine the errors of our solutions over the interval $[0, 1]$, doubling the partition at each iteration. We will then examine how our errors behave.

```
# Checking function
def check(f, x0, t1, method, xt):

    # Find the maximum error
    def err_max(p):
        T, X = method(f, 0, t1, x0, p)
        return max(abs(X - xt(np.array(T))))

    # Errors collected by passing successive powers of 2 as the number of steps
    errs = np.array([err_max(2 ** (i + 1)) for i in range(7)])

    # Examining error ratios between successive error values
    ratios = errs[:-1] / errs[1:]

    # The order is the logarithm of the error ratio; we used powers of 2 for better
   representation, so we take the logarithm base 2
    return np.log2(ratios)

test1 = lambda t, x: x # exp
test2 = lambda t, x: -2 * t * (x ** 2)

result1 = check(test1, 1, 1, DP, np.exp)
result2 = check(test2, 1, 1, DP, lambda t: 1 / (1 + t ** 2))

print(result1)
print(result2)
```

We get

```
[4.24178468 4.6629271  4.83907669 4.92119045 4.96148632 4.83476364]
[3.81179864 5.26234021 5.21968578 5.14066296 5.07992758 5.03271988]
```

Based on this, we can conclude that the order of the DP scheme is 5.

4 Optimal Step Size

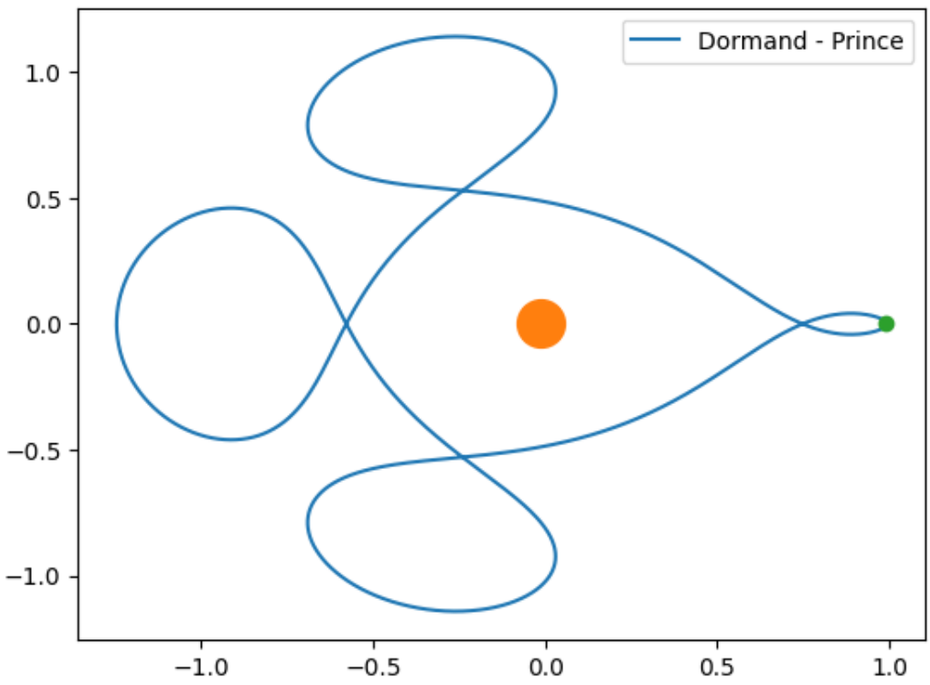
We want to adjust the step size h, so *de facto* we want to check the accuracy of the method for different partitions N and how close the given solution is to the periodic solution. For this purpose, we define a function that, for a fixed N and a given method, checks how close the last step is to the starting point.

```
# Function that finds the partition N for a given method where we get closest to the
   periodic solution
def find_best_step(method):
  diff = [] # array of differences
  for N in range(1000, 15000, 1000):
    T, XY = method(f, 0, T0, np.array([x_0, y_0, x_p_0, y_p_0]), N) # generating data
   for the given method
    start = XY[0][:2]
    stop = XY[-1][:2]
    d = np.linalg.norm(start - stop) # comparing the error between the beginning and end
    of the data
    diff.append((d, N))
  min_tuple = min(diff, key=lambda x: x[0]) # finding the partition with the smallest
   difference

  return min_tuple

best_step = find_best_step(DP)[1]
print(best_step)
```

To keep the calculations from taking too long, we set the partition increment to 1000 and the maximum partition to 14000. We find that in this case, this partition is the closest to the correct solution. We plot our scheme:

```
T_DP, XY_DP = DP(f, 0, T0, np.array([x_0, y_0, x_p_0, y_p_0]), best_step)
XY_DP = np.array(XY_DP)
plt.plot(XY_DP[:,0], XY_DP[:,1], label="Dormand - Prince")
plt.plot(-mu, 0, "o", markersize=20)
plt.plot(1-mu, 0, "o")
plt.legend(loc="best")
```

```
plt.show()
```



We observe an improvement over the RK4 scheme; this time, we obtained a trajectory much closer to a stable orbit.

5 Improved Dormand-Prince Method

We implement a function for the improved Dormand-Prince method, which adjusts the step size based on the approximations obtained:

```
# Improved Dormand-Prince method
def imp_DP(f, a, b, x0, N):
    h = (b - a) / N
    T = [a]
    X = [x0]
    t = a
    x = x0
    while t < b: # Ensure our step does not go beyond the right boundary
      K1 = f(t, x)
      K2 = f(t + h / 5, x + h * (K1 / 5))
      K3 = f(t + 3 * h / 10, x + h * (K1 * 3 / 40 + K2 * 9 / 40))
      K4 = f(t + 4 * h / 5, x + h * (K1 * 44 / 45 - K2 * 56 / 15 + K3 * 32 / 9))
      K5 = f(t + 8 * h / 9, x + h * (K1 * 19372 / 6561 - K2 * 25360 / 2187 + K3 * 64448
   / 6561 - K4 * 212 / 729))
      K6 = f(t + h, x + h * (K1 * 9017 / 3168 - K2 * 355 / 33 + K3 * 46732 / 5247 + K4 *
    49 / 176 - K5 * 5103 / 18656))
      K7 = f(t + h, x + h * (K1 * 35 / 384 + K3 * 500 / 1113 + K4 * 125 / 192 - K5 *
   2187 / 6784 + K6 * 11 / 84))

      x_d = x + h * (K1 * 5179 / 57600 + K3 * 7571 / 16695 + K4 * 393 / 640 - K5 * 92097
    / 339200 + K6 * 11 / 84 + K7 * 1 / 40) # auxiliary x with a hat
      x = x + h * (K1 * 35 / 384 + K3 * 500 / 1113 + K4 * 125 / 192 - K5 * 2187 / 6784 +
    K6 * 11 / 84)

      # Calculate error and adjust step size
      err = np.linalg.norm((x - x_d)[:2])
      toll = 0.0001
      gamma = (toll / err) ** (1 / 6)
      h = gamma * h
      t = t + h
      T.append(t)
      X.append(x)
    return T, X
```

Now let's compare the improved scheme with the previous one. Since the solution is periodic, we will check which scheme ends up closer to the starting point:

```
# Data from the improved scheme
T_iDP, XY_iDP = imp_DP(f, 0, T0, np.array([x_0, y_0, x_p_0, y_p_0]), best_step)
XY_iDP = np.array(XY_iDP)

# Calculate errors between the start and end of both schemes
err_DP = np.linalg.norm(XY_DP[-1][:2] - XY_DP[0][:2])
err_iDP = np.linalg.norm(XY_iDP[-1][:2] - XY_iDP[0][:2])

print("Error of regular Dormand-Prince:", round(err_DP, 9))
print("Error of improved Dormand-Prince:", round(err_iDP, 9))
print((np.linalg.norm(XY_DP[-1][:2] - XY_DP[0][:2])) > (np.linalg.norm(XY_iDP[-1][:2] -
    XY_iDP[0][:2])))
```

For a step size of 14,000, we get:

```
Error of regular Dormand-Prince: 0.000166677
Error of improved Dormand-Prince: 0.000195515
```

Both methods have similar precision, but the apparent "improvement" is difficult to notice. The explanation can be found if we compare the execution time of both functions for a large step size:

```
import time

# Function measuring the execution time of a given method
def measure_time(func, *args):
    start_time = time.time()  # Start time measurement
    result = func(*args)

    end_time = time.time()  # End time measurement

    elapsed_time = end_time - start_time  # Calculate execution time
    print(f"Execution time of function {func.__name__}: {elapsed_time:.6f} seconds")
    return result[0], result[1]

# Measure the execution time of both methods for the best given step size
r = measure_time(DP, f, 0, T0, np.array([x_0, y_0, x_p_0, y_p_0]), best_step)
r = measure_time(imp_DP, f, 0, T0, np.array([x_0, y_0, x_p_0, y_p_0]), best_step)
```

We get:

```
Execution time of function DP: 7.066391 seconds
Execution time of function imp_DP: 1.473765 seconds
```

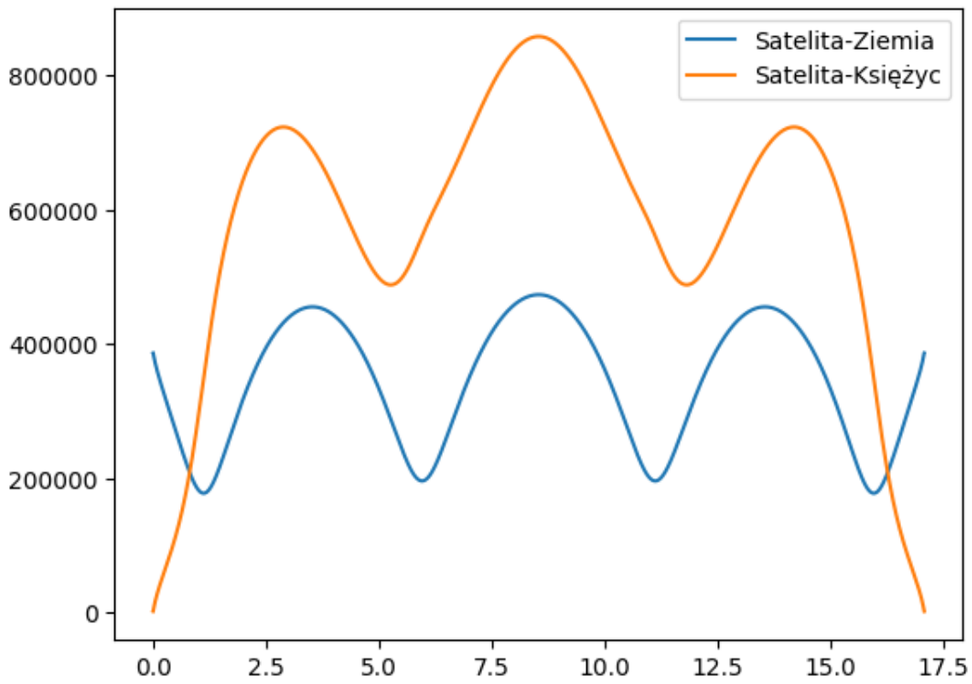
For larger step sizes, we see that both methods have similar order of precision, but the execution time of the original scheme increases steadily, whereas the method with adaptive step size performs in roughly the same time regardless of the number of steps.

6 Satellite - Earth and Satellite - Moon Distance

We read that the distance between the Earth and the Moon is approximately 384,400 km. We implement this in our code:

```
ME_dist = 384400 # Earth-Moon distance
Earth = np.array([-mu, 0]) # Earth's position in our system
Moon = np.array([1 - mu, 0]) # Moon's position in our system
Satelite = XY_DP[:, :2] # Satellite positions

# function to calculate distance between a list of coordinates and a single object
def calculate_distance(coord_list, object):
    return np.array([np.linalg.norm(item - object) for item in coord_list])

SE = ME_dist * calculate_distance(Satelite, Earth) # array of distances between
   Satellite and Earth
SM = ME_dist * calculate_distance(Satelite, Moon) # array of distances between Satellite
    and Moon

# Plotting our results
plt.plot(T_DP, SE, label="Satellite-Earth")
plt.plot(T_DP, SM, label="Satellite-Moon")
plt.legend(loc="best")
plt.xlabel("Time")
plt.ylabel("Distance (km)")
plt.title("Distance between Satellite and Earth/Moon")
plt.show()
```

We obtain a plot consistent with the trajectories obtained earlier:



7 Comparison of Methods

Let's generate data with a step size of 14,000:

```
# Generating data for a single step
interval = 14000
T_E, XY_E = Euler(f, 0, T0, np.array([x_0,y_0, x_p_0, y_p_0]), interval)
XY_E = np.array(XY_E)

T_iDP, XY_iDP = imp_DP(f, 0, T0, np.array([x_0,y_0, x_p_0, y_p_0]), interval)
XY_iDP = np.array(XY_iDP)

T_DP, XY_DP = DP(f, 0, T0, np.array([x_0,y_0, x_p_0, y_p_0]), interval)
XY_DP = np.array(XY_DP)

# Printing results
print("Euler's Method:", np.linalg.norm(XY_E[-1][:2] - XY_E[0][:2]) * ME_dist)
print("Dormand-Prince Method:", np.linalg.norm(XY_DP[-1][:2] - XY_DP[0][:2]) * ME_dist)
print("RK4 Method:", np.linalg.norm(XY_RK[-1][:2] - XY_RK[0][:2]) * ME_dist)
print("Improved Dormand-Prince Method:", np.linalg.norm(XY_iDP[-1][:2] - XY_iDP[0][:2])
    * ME_dist)
```

We obtain the following results:

```
Euler's Method: 5489167.964680192
Dormand-Prince Method: 64.0706078502976
RK4 Method: 133911.85537837923
Improved Dormand-Prince Method: 75.15584790004709
```