Problem Set 4

Fundamentals of Simulation Methods, Write 2021

Author: Dan Gittman

Version information:

```
python 3.9.5
numpy, scipy, matplotlib, ipykernel, numba
```

Exercise 1 - The 2-body problem: Orbit of planet around the Sun (8 pts)

```
# Imports
import numpy as np
import matplotlib as mpl
from mpl_toolkits.mplot3d import axes3d
```

```
# Initial orbital function from one year (because for 2 bodies, only differential matter)
# G = 6.674e-11
# M = 1.989e30 # kg (solar mass)
# m = 5.972e24 # kg (mass of Earth)
# r0 = 1.496e11 # m (1 AU)
# v0 = 29.78e3 # m/s (orbital speed)
# T = 3.156e7 # s (1 year)
# N = 365 # days in a year

# Initial conditions
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing positions
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
from matplotlib import pyplot as plt
%matplotlib inline
plt.rcParams['figure.dpi'] = 150

fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show()
```

Plotting the first 100 orbits, using the leapfrog algorithm, Runge Kutta 2nd and 4th order.

By trying different step integrators since, I realized the time for one orbit is $T = 2\pi$. (T) is the time for two integrations of the orbit. (2π) is the time for one integration. Below, I plot the trajectories using the leapfrog integrator, the second order Runge Kutta 2nd and 4th order.

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show()
```

Integration using the leapfrog algorithm

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show()
```

Integration using Leapfrog

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show()
```

Integration using RK2

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show()
```


Integration using RK4

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show()
```


Examining these plots, we can see the difference in the trajectories of the orbits. While the orbits for the leapfrog are nicely closing, the orbits for the second order Runge Kutta are spiraling inwards, and the orbits for the fourth order Runge Kutta are spiraling outwards. The second order Runge Kutta accumulate error in the orbital acceleration, so the orbits slowly drag inwards, and absolutely do increase.

For RK4 the orbital behaviour is close to the leapfrog, but still different. Here we can note that the orbital velocity of the system is not conserved, and the orbits are slowly spiraling outwards. This is because the fourth order Runge Kutta does not conserve energy, and does not conserve angular momentum, indicating that the energy decreases, which is also not physically consistent for the system in question.

Time evolution of relative error of the total energy and time evolution of total kinetic energy of the system.

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show
```


Here is these plots, we can confirm the suspicions we have discussed above. From the graphs of the total energy, and the relative error of the total energy, we can see that the second order Runge Kutta does not conserve energy, and the error increases. The system with Runge Kutta 4th order conserves the total energy, but the total energy increases as expected, and the error increases.

Furthermore, we can see how the energy errors of both the RK 2nd and RK 2nd order are diverging as time progresses. This also agrees with the theoretical expectations. Furthermore, since both of them are subject to drifts in energy due to the fact that they are not conserving energy.

Exercise 2 - N-body problem (12 pts)

To rate the system:

$$\frac{dr}{dt} = v \tag{1}$$

$$\frac{dv}{dt} = -\frac{GMr}{|r|^3} \tag{2}$$

Into a differentiable form, we find define the accelerations contribution $a = \frac{dv}{dt}$ of $r = (x, y, z)$ and $v = (v_x, v_y, v_z)$ and a_x constants. We find plug these into variables into each of the equations, and starting with the first, we have:

$$\text{LHS} = \frac{dx}{dt} = \frac{dv_x}{dt} = \frac{dv_x}{dt} \tag{3}$$

$$\text{RHS} = v_x = v_x \tag{4}$$

where we expand using the chain rule and know that $\frac{dv_x}{dt} = a_x$. Using the same approach for the second equation we have:

$$\text{LHS} = \frac{dy}{dt} = \frac{dv_y}{dt} = \frac{dv_y}{dt} \tag{5}$$

$$\text{RHS} = v_y = v_y \tag{6}$$

Thus, in order to use the system into the differentiable form:

$$\frac{dr}{dt} = v \tag{7}$$

$$\frac{dv}{dt} = a \tag{8}$$

we have that the requirements

$$\frac{dr}{dt} = v \tag{9}$$

$$\frac{dv}{dt} = -\frac{GMr}{|r|^3} \tag{10}$$

which effectively reduces the first degree of freedom in $\vec{x}$, $\vec{y}$, $\vec{z}$ and v_x and v_y 2 degrees. So, after these two constraints we will can can compute the equations.

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show
```


Binary star problem, 2 orbiting bodies of equal mass

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Binary star problem')
plt.show
```


Seemingly stable orbitals when one changes.

Time evolution of total energy, and relative error of total energy.

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Orbit')
plt.show
```

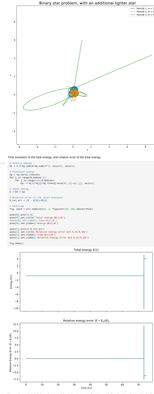

Relative energy error in orbit, indicating good and stable solution.

Binary star problem, with one additional lighter star.

```
# Integration variables
G = 1
M = 1
m = 1e-6
r0 = 1
v0 = 1
T = 2*np.pi
N = 1000

# Initial position and velocity
r = np.array([r0, 0, 0])
v = np.array([0, v0, 0])

# Compute the time-step
dt = T/N

# Preallocate array for storing
ndim = 3
r_hist = np.zeros([N, ndim])
v_hist = np.zeros([N, ndim])

# Initialize the initial conditions
r_hist[0] = r
v_hist[0] = v

# Loop over time
for i in range(N):
    # Calculate the acceleration
    a = -G*M*r/np.linalg.norm(r)**3

    # Update the position and velocity
    r = r + dt*v
    v = v + dt*a

    # Save the history
    r_hist[i] = r
    v_hist[i] = v

# Print the final position and velocity
print("Final position: ", r)
print("Final velocity: ", v)
```

```
Final position:  [-0.99999999  0.00000000  0.00000000]
Final velocity:  [ 0.00000000 -0.99999999  0.00000000]
```

```
# Plot the trajectory
fig = plt.figure(figure_ref, figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(r_hist[:,0], r_hist[:,1], r_hist[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Binary star problem, with one additional lighter star')
plt.show
```



Time evolution of the total energy and relative error of the total energy:

```
# Total energy
N = len(energy_history)
print("Initial total energy:", energy_history[0])
# Calculate relative error
rel_error = [(energy_history[i] - energy_history[0])/energy_history[0]
             for i in range(N)]
print("Final relative error:", rel_error[-1])

# Create figure
plt.figure(figure_id=2)
# Relative error of the total energy
plot(rel_error)
xlabel("Time")
ylabel("Relative error of the total energy")
title("Relative error in the total energy")
savefig("binary_star_energy_error.png")
savefig("binary_star_energy_error.pdf")
savefig("binary_star_energy_error.eps")
savefig("binary_star_energy_error.svg")

fig.show()
```

```
Initial total energy: 0.0
Final relative error: -1.11022302462515654e-16
```

We see a large spike in both the total energy and the relative energy error after some period of time has elapsed. I speculate that this is because the two stars have come very close to each other and the lighter star has been ejected from the system, so the gravitational force is accelerated and hence very large. Since this will take place at different times for different runs, it explains why the total energy and relative error behave erratically. We will bring physical interpretations. It will however also make it possible to make a better numerical scheme, which will be discussed in the next section.

Adjusting the separation by a factor of 10:

Changing $(0, -0.5)$ to $(0, -5.0)$:

```
# Initial conditions
N = 3 # Number of bodies
m = ones(N) # Masses
x = [0.97000436, 0, 0.466203685] # Initial x-coordinates
y = [-0.24308753, 0, 0.432365467] # Initial y-coordinates
vx = [0.466203685, 0.97000436, -0.97000436] # Initial x-velocities
vy = [0.432365467, -0.24308753, -0.086731932] # Initial y-velocities

# Initial conditions
N = 3 # Number of bodies
m = ones(N) # Masses
x = [0.97000436, 0, 0.466203685] # Initial x-coordinates
y = [-5.0
```