Handin Assignment 4

Ioannis Koutalios s3365530

January 6, 2025

Abstract

Code and results for Handin assignment 4 for the course Numerical Recipes in Astrophysics.

1 Simulating the solar system

In this section, we will use two different differential equation solvers to calculate the gravitational forces in our solar system.

1.1 Initial Positions

Firstly we want to obtain the initial conditions. Our problem is very sensitive to the initial positions which means that we need high precision. We use the `astropy` module to get the positions and velocities of all 8 planets and the Sun. We then create lists to use for plotting and future scripts.

```
from astropy.time import Time
from astropy.coordinates import solar_system_ephemeris, get_body_barycentric_posvel
from astropy import units as u
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import seaborn as sns
import numpy as np

# set the current time
t = Time('2021-12-07 10:00')

with solar_system_ephemeris.set('jpl'):
    sun = get_body_barycentric_posvel('sun', t)
    mercury = get_body_barycentric_posvel('mercury', t)
    venus = get_body_barycentric_posvel('venus', t)
    earth = get_body_barycentric_posvel('earth', t)
    mars = get_body_barycentric_posvel('mars', t)
    jupiter = get_body_barycentric_posvel('jupiter', t)
    saturn = get_body_barycentric_posvel('saturn', t)
    uranus = get_body_barycentric_posvel('uranus', t)
    neptune = get_body_barycentric_posvel('neptune', t)

sunposition = sun[0]
sunvelocity = sun[1]

mercuryposition = mercury[0]
mercuryvelocity = mercury[1]

venusposition = venus[0]
venusvelocity = venus[1]

earthposition = earth[0]
earthvelocity = earth[1]

marsposition = mars[0]
marsvelocity = mars[1]

jupiterposition = jupiter[0]
jupitervelocity = jupiter[1]
```

```
40
41 saturnposition = saturn[0]
42 saturnvelocity = saturn[1]
43
44 uranusposition = uranus[0]
45 uranusvelocity = uranus[1]
46
47 neptuneposition = neptune[0]
48 neptunevelocity = neptune[1]
49
50 x_positions = [sunposition.x.to_value(u.AU),mercuryposition.x.to_value(u.AU),
       venusposition.x.to_value(u.AU)
51 ,earthposition.x.to_value(u.AU),marsposition.x.to_value(u.AU),jupiterposition.x.to_value
       (u.AU)
52 ,saturnposition.x.to_value(u.AU),uranusposition.x.to_value(u.AU),neptuneposition.x.
       to_value(u.AU)]
53
54 y_positions = [sunposition.y.to_value(u.AU),mercuryposition.y.to_value(u.AU),
       venusposition.y.to_value(u.AU)
55 ,earthposition.y.to_value(u.AU),marsposition.y.to_value(u.AU),jupiterposition.y.to_value
       (u.AU)
56 ,saturnposition.y.to_value(u.AU),uranusposition.y.to_value(u.AU),neptuneposition.y.
       to_value(u.AU)]
57
58 z_positions = [sunposition.z.to_value(u.AU),mercuryposition.z.to_value(u.AU),
       venusposition.z.to_value(u.AU)
59 ,earthposition.z.to_value(u.AU),marsposition.z.to_value(u.AU),jupiterposition.z.to_value
       (u.AU)
60 ,saturnposition.z.to_value(u.AU),uranusposition.z.to_value(u.AU),neptuneposition.z.
       to_value(u.AU)]
61
62 x_velocities = [sunvelocity.x.to_value(u.AU/u.day),mercuryvelocity.x.to_value(u.AU/u.day
       ),venusvelocity.x.to_value(u.AU/u.day)
63 ,earthvelocity.x.to_value(u.AU/u.day),marsvelocity.x.to_value(u.AU/u.day),
       jupitervelocity.x.to_value(u.AU/u.day)
64 ,saturnvelocity.x.to_value(u.AU/u.day),uranusvelocity.x.to_value(u.AU/u.day),
       neptunevelocity.x.to_value(u.AU/u.day)]
65
66 y_velocities = [sunvelocity.y.to_value(u.AU/u.day),mercuryvelocity.y.to_value(u.AU/u.day
       ),venusvelocity.y.to_value(u.AU/u.day)
67 ,earthvelocity.y.to_value(u.AU/u.day),marsvelocity.y.to_value(u.AU/u.day),
       jupitervelocity.y.to_value(u.AU/u.day)
68 ,saturnvelocity.y.to_value(u.AU/u.day),uranusvelocity.y.to_value(u.AU/u.day),
       neptunevelocity.y.to_value(u.AU/u.day)]
69
70 z_velocities = [sunvelocity.z.to_value(u.AU/u.day),mercuryvelocity.z.to_value(u.AU/u.day
       ),venusvelocity.z.to_value(u.AU/u.day)
71 ,earthvelocity.z.to_value(u.AU/u.day),marsvelocity.z.to_value(u.AU/u.day),
       jupitervelocity.z.to_value(u.AU/u.day)
72 ,saturnvelocity.z.to_value(u.AU/u.day),uranusvelocity.z.to_value(u.AU/u.day),
       neptunevelocity.z.to_value(u.AU/u.day)]
73
74 POSITIONS = np.array((x_positions, y_positions, z_positions)).T      # AU
75 POSITIONS -= POSITIONS[0]        # subtracting the sun initial position
76 VELOCITIES = np.array((x_velocities, y_velocities, z_velocities)).T # AU/day
77 VELOCITIES -= VELOCITIES[0]      # subtracting the sun initial velocity
78
79
80 if __name__ == '__main__':
81
82     names = ['sun','mercury','venus','earth','mars','jupiter','saturn','uranus','neptune
       ']
83     dic = {i: name for i,name in enumerate(names)}
84     color_list = sns.color_palette("Paired", n_colors=len(names))
85
86     plt.figure()
87     scatter = plt.scatter(x_positions,y_positions,c=list(dic.keys()),cmap=ListedColormap
       (color_list.as_hex()))
88     plt.legend(handles=scatter.legend_elements()[0], labels=names)
89     plt.xlabel('x [AU]')
```

```
90      plt.ylabel('y [AU]')
91      plt.savefig('plots/x-y.png', dpi=300)
92      plt.close()
93
94      plt.scatter(x_positions,z_positions)
95      scatter = plt.scatter(x_positions,z_positions,c=list(dic.keys()),cmap=ListedColormap
        (color_list.as_hex()))
96      plt.legend(handles=scatter.legend_elements()[0], labels=names)
97      plt.xlabel('x [AU]')
98      plt.ylabel('z [AU]')
99      plt.savefig('plots/x-z.png', dpi=300)
100     plt.close()
```

initial.py

In Figures 1 and 2 we can find the initial positions of the planets. The first plot gives us the positions in the x-y plane, while the latter is in the x-z.

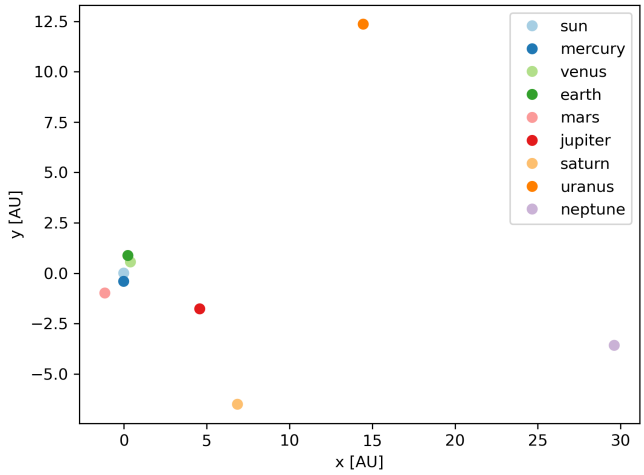

Figure 1: The initial x and y positions of all the 8 planets and the Sun.

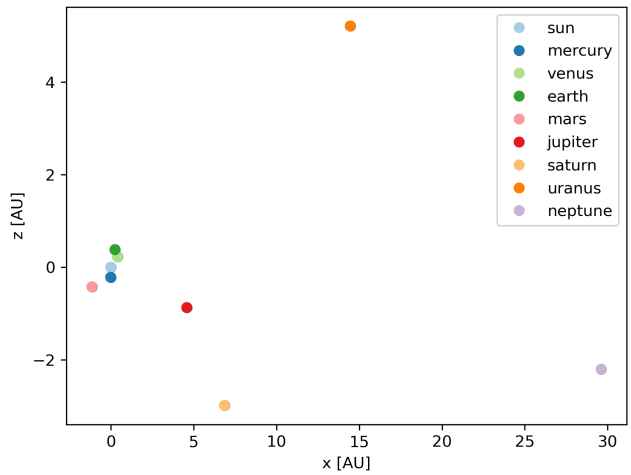

Figure 2: The initial x and z positions of all the 8 planets and the Sun.

1.2 Solving with Leapfrog Algorithm

We will now use the "Leapfrog" algorithm to integrate the differential equation for the gravitational forces. What we want to integrate is:

$$r''(t) = -\frac{GM_{\odot}r(t)}{|r(t)|^3} \tag{1}$$

where $r(t)$ is the (x, y, z) position of the planet and $r''(t)$ the acceleration. G is the gravitational constant and $M_{\odot}$ is the mass of the Sun. We create the "acceleration" function for that use.

To solve this differential equation we implement the "leapfrog" function. We first initialize three arrays for t, r, r', and then we solve using a loop calculating each time the position and velocity of the planet. We do that by kicking the planet at a half-step and then drifting the position of the planet accordingly. We then kick the planet at the other half-step. By doing that we make sure that there is symmetry when solving the 2nd-order differential equation. That ensures that our solutions are time-reversible and therefore energy is conserved. This makes Leapfrog a very suitable algorithm for solving orbital differential equations.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy import units as u
from astropy import constants as const
from initial import POSITIONS, VELOCITIES

G = const.G.to_value((u.AU**3)/(u.kg*u.day**2))

def Acceleration(t: float, pos: np.ndarray) -> np.ndarray:
    '''
    function for the gravitational acceleration
    from the Sun (assumes Sun location is at [0,0,0])
    input:
    t : time
    pos : 3D position of the planet
    returns:
    gravitaional acceleration from the Sun
    AU/day^2
    '''
    return -((G * const.M_sun * pos)/(pos[0]**2 + pos[1]**2 + pos[2]**2)**(3/2) ).
    to_value()

def LeapFrog(f: callable, y0: np.ndarray, v0: np.ndarray, t0: float, tf: float, h: float
    ) -> tuple:
    '''
    leapfrog solver for second order ODE's
    input:
    f : ODE
    y0 : initial y values
    v0 : initial y' values
    t0 : initial t values
    tf : finial t values
    h : step in t
    returns :
    t: t values
    y: y values
    v: y' values
    '''
    # initialize
    t = np.arange(t0, tf + h, h)
    y = np.zeros((len(t), 3))
    v = np.zeros((len(t), 3))
    y[0,:], v[0,:] = y0, v0

    # solve
    for i in range(1, len(t)):
        v[int(i-1/2),:] = v[i-1] + 1/2 * h * f(t[i-1], y[i-1])
        y[i,:] = y[i-1] + h * v[int(i-1/2)]
        v[i,:] = v[int(i-1/2)] + 1/2 * h * f(t[i], y[i])

    return t, y, v
```

```
50
51
52 if __name__ == '__main__':
53     # get the orbits
54     t , y_mercury, v_mercury = LeapFrog(Acceleration, POSITIONS[1], VELOCITIES[1], 0,
       200*365.25, .5)
55     t , y_venus, v_venus = LeapFrog(Acceleration, POSITIONS[2], VELOCITIES[2], 0,
       200*365.25, .5)
56     t , y_earth, v_earth = LeapFrog(Acceleration, POSITIONS[3], VELOCITIES[3], 0,
       200*365.25, .5)
57     t , y_mars, v_mars = LeapFrog(Acceleration, POSITIONS[4], VELOCITIES[4], 0,
       200*365.25, .5)
58     t , y_jupiter, v_jupiter = LeapFrog(Acceleration, POSITIONS[5], VELOCITIES[5], 0,
       200*365.25, .5)
59     t , y_saturn, v_saturn = LeapFrog(Acceleration, POSITIONS[6], VELOCITIES[6], 0,
       200*365.25, .5)
60     t , y_uranus, v_uranus = LeapFrog(Acceleration, POSITIONS[7], VELOCITIES[7], 0,
       200*365.25, .5)
61     t , y_neptune, v_neptune = LeapFrog(Acceleration, POSITIONS[8], VELOCITIES[8], 0,
       200*365.25, .5)
62
63     # save them for next script
64     np.save('output/leapfrog.npy',np.array([y_mercury,y_venus,y_earth,y_mars,y_jupiter,
       y_saturn,y_uranus,y_neptune]))
65
66     # create plots
67     fig = plt.figure(figsize=[6.4, 6.4])
68     plt.plot(y_mercury[:,0],y_mercury[:,1],label='mercury')
69     plt.plot(y_venus[:,0],y_venus[:,1],label='venus')
70     plt.plot(y_earth[:,0],y_earth[:,1],label='earth')
71     plt.plot(y_mars[:,0],y_mars[:,1],label='mars')
72     plt.plot(y_jupiter[:,0],y_jupiter[:,1],label='jupiter')
73     plt.plot(y_saturn[:,0],y_saturn[:,1],label='saturn')
74     plt.plot(y_uranus[:,0],y_uranus[:,1],label='uranus')
75     plt.plot(y_neptune[:,0],y_neptune[:,1],label='neptune')
76     plt.legend()
77     plt.xlabel('x [AU]')
78     plt.ylabel('y [AU]')
79     plt.savefig('plots/xy-leapfrog.png', dpi=300)
80     plt.close()
81
82     fig = plt.figure(figsize=[6.4, 6.4])
83     plt.plot(y_mercury[:,0],y_mercury[:,1],label='mercury')
84     plt.plot(y_venus[:,0],y_venus[:,1],label='venus')
85     plt.plot(y_earth[:,0],y_earth[:,1],label='earth')
86     plt.plot(y_mars[:,0],y_mars[:,1],label='mars')
87     plt.legend()
88     plt.xlabel('x [AU]')
89     plt.ylabel('y [AU]')
90     plt.xlim(-2,2)
91     plt.ylim(-2,2)
92     plt.savefig('plots/xy-leapfrog-zoom.png', dpi=300)
93     plt.close()
94
95     fig = plt.figure()
96     plt.plot(t,y_neptune[:,2],label='neptune')
97     plt.plot(t,y_uranus[:,2],label='uranus')
98     plt.plot(t,y_saturn[:,2],label='saturn')
99     plt.plot(t,y_jupiter[:,2],label='jupiter')
100    plt.plot(t,y_mars[:,2],label='mars')
101    plt.plot(t,y_earth[:,2],label='earth')
102    plt.plot(t,y_venus[:,2],label='venus')
103    plt.plot(t,y_mercury[:,2],label='mercury')
104    plt.legend()
105    plt.ylabel('z [AU]')
106    plt.xlabel('t [days]')
107    plt.savefig('plots/z-leapfrog.png', dpi=300)
108    plt.close()
109
110    fig = plt.figure()
```

```
111     plt.plot(t,y_mars[:,2],label='mars')
112     plt.plot(t,y_earth[:,2],label='earth')
113     plt.plot(t,y_venus[:,2],label='venus')
114     plt.plot(t,y_mercury[:,2],label='mercury')
115     plt.legend()
116     plt.ylim(-1,1)
117     plt.ylabel('z [AU]')
118     plt.xlabel('t [days]')
119     plt.savefig('plots/z-leapfrog-zoom.png', dpi=300)
120     plt.close()
```

leapfrog.py

In the main part of the script we call the "leapfrog" function for all 8 planets and we give their initial positions and velocities. We want to solve it from the starting date and for 200 years with a step of 0.5 days. We then create plots for the evolution of the position of the planets on the x-y plane and also plot the evolution of the z-position of the planets. We can find those plots in Figures 3 to 6. As we can see the solutions are correct since all our planets rotate around the sun with stability over the course of the 200 years. Only Mercury shows some instability, the orbit is however close and does not diverge. For the evolution of the z positions, we can find the expected sinusoidal lines, with higher periods and amplitudes for the planets that are further away from the sun. This is also what we expect since the period of these lines should be the rotational period of each planet and the amplitude represents how far from the ecliptic plane can each planet get, which is higher for planets further away from the sun.

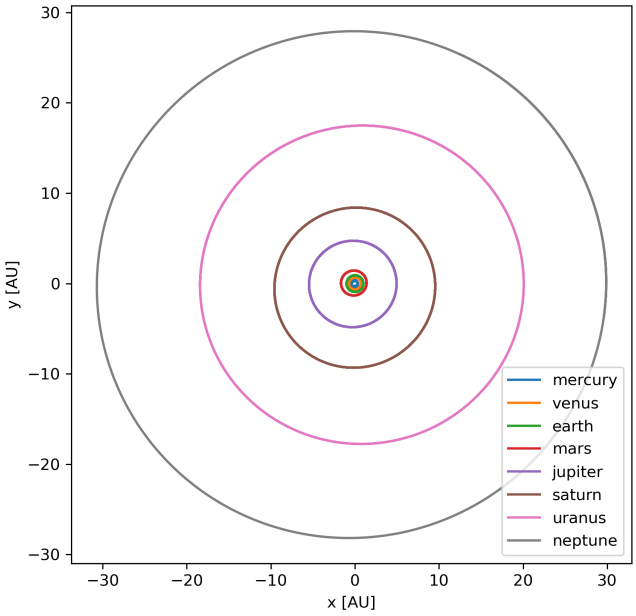

Figure 3: The evolution of the x and y positions of all the 8 planets when integrating using the Leapfrog algorithm.

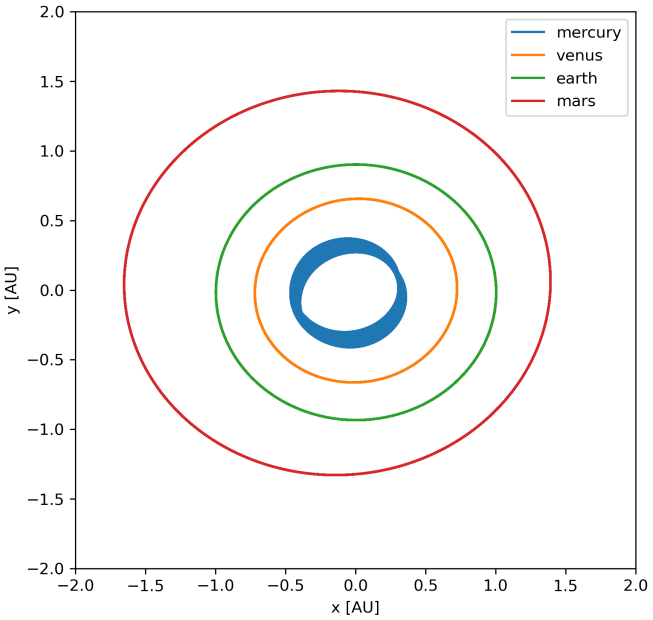

Figure 4: The evolution of the x and y positions of the 4 inner planets when integrating using the Leapfrog algorithm.

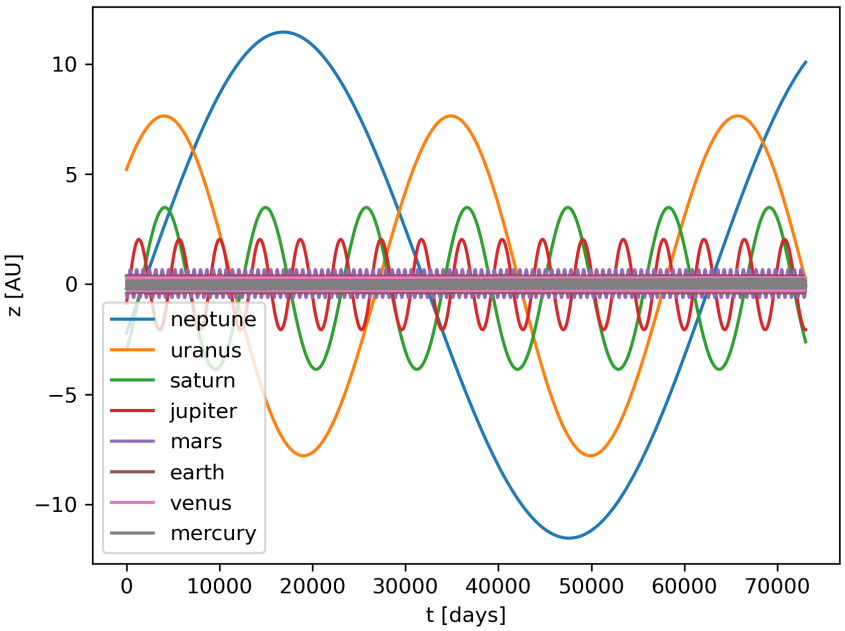

Figure 5: The evolution of the z positions of all the 8 planets when integrating using the Leapfrog algorithm.

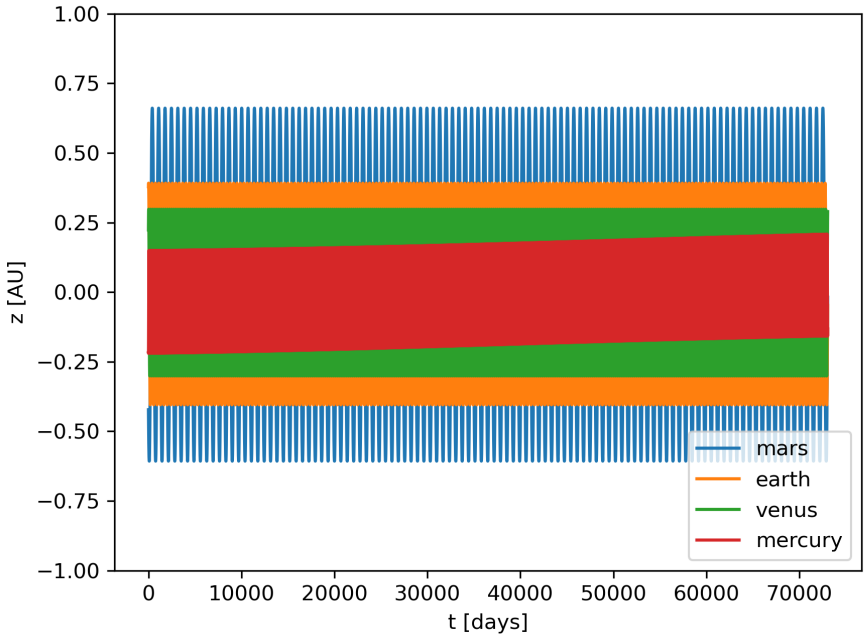

Figure 6: The evolution of the z positions of the 4 inner planets when integrating using the Leapfrog algorithm.

1.3 Solving with Euler's Algorithm

We also want to solve the system using Euler's algorithm. For this, we implement the "euler" function which utilizes this algorithm to solve the 2nd-order differential equation. We once again initialize three arrays for t, r, r' and then solve using a loop and the appropriate equations for Euler's algorithm. We also need to import all the initial conditions from the previous script with the function for the differential equation.

```
import numpy as np
import matplotlib.pyplot as plt
from leapfrog import Acceleration
from initial import POSITIONS, VELOCITIES

def Euler(f: callable, y0: np.ndarray, v0: np.ndarray, t0: float, tf: float, h: float)
    -> tuple:
    '''
    euler solver for second order ODE's
    input:
    f : ODE
    y0 : initial y values
    v0 : initial y' values
    t0 : initial t values
    tf : finial t values
    h : step in t
    returns :
    t: t values
    y: y values
    v: y' values
    '''
    # initialize
    t = np.arange(t0, tf + h, h)
    y = np.zeros((len(t), 3))
    v = np.zeros((len(t), 3))
    y[0,:], v[0,:] = y0, v0

```

```
27      # solve
28      for i in range(1, len(t)):
29          y[i,:] = y[i-1,:] + h * v[i-1,:]
30          v[i,:] = v[i-1,:] + h * f(t[i-1], y[i-1,:])
31
32      return t, y, v
33
34  if __name__ == '__main__':
35      # get the orbits
36      t , y_mercury, v_mercury = Euler(Acceleration, POSITIONS[1], VELOCITIES[1], 0,
        200*365.25, .5)
37      t , y_venus, v_venus = Euler(Acceleration, POSITIONS[2], VELOCITIES[2], 0,
        200*365.25, .5)
38      t , y_earth, v_earth = Euler(Acceleration, POSITIONS[3], VELOCITIES[3], 0,
        200*365.25, .5)
39      t , y_mars, v_mars = Euler(Acceleration, POSITIONS[4], VELOCITIES[4], 0, 200*365.25,
         .5)
40      t , y_jupiter, v_jupiter = Euler(Acceleration, POSITIONS[5], VELOCITIES[5], 0,
        200*365.25, .5)
41      t , y_saturn, v_saturn = Euler(Acceleration, POSITIONS[6], VELOCITIES[6], 0,
        200*365.25, .5)
42      t , y_uranus, v_uranus = Euler(Acceleration, POSITIONS[7], VELOCITIES[7], 0,
        200*365.25, .5)
43      t , y_neptune, v_neptune = Euler(Acceleration, POSITIONS[8], VELOCITIES[8], 0,
        200*365.25, .5)
44
45      #load the orbits of the previous script
46      leapfrog = np.load('output/leapfrog.npy')
47
48      # create plots
49      fig = plt.figure(figsize=[6.4, 6.4])
50      plt.plot(y_mercury[:,0],y_mercury[:,1],label='mercury')
51      plt.plot(y_venus[:,0],y_venus[:,1],label='venus')
52      plt.plot(y_earth[:,0],y_earth[:,1],label='earth')
53      plt.plot(y_mars[:,0],y_mars[:,1],label='mars')
54      plt.plot(y_jupiter[:,0],y_jupiter[:,1],label='jupiter')
55      plt.plot(y_saturn[:,0],y_saturn[:,1],label='saturn')
56      plt.plot(y_uranus[:,0],y_uranus[:,1],label='uranus')
57      plt.plot(y_neptune[:,0],y_neptune[:,1],label='neptune')
58      plt.legend()
59      plt.xlabel('x [AU]')
60      plt.ylabel('y [AU]')
61      plt.savefig('plots/xy-euler.png', dpi=300)
62      plt.close()
63
64      fig = plt.figure(figsize=[6.4, 6.4])
65      plt.plot(y_mercury[:,0],y_mercury[:,1],label='mercury')
66      plt.plot(y_venus[:,0],y_venus[:,1],label='venus')
67      plt.plot(y_earth[:,0],y_earth[:,1],label='earth')
68      plt.plot(y_mars[:,0],y_mars[:,1],label='mars')
69      plt.legend()
70      plt.xlabel('x [AU]')
71      plt.ylabel('y [AU]')
72      plt.xlim(-3,3)
73      plt.ylim(-3,3)
74      plt.savefig('plots/xy-euler-zoom.png', dpi=300)
75      plt.close()
76
77      fig = plt.figure()
78      plt.plot(t,y_neptune[:,2],label='neptune')
79      plt.plot(t,y_uranus[:,2],label='uranus')
80      plt.plot(t,y_saturn[:,2],label='saturn')
81      plt.plot(t,y_jupiter[:,2],label='jupiter')
82      plt.plot(t,y_mars[:,2],label='mars')
83      plt.plot(t,y_earth[:,2],label='earth')
84      plt.plot(t,y_venus[:,2],label='venus')
85      plt.plot(t,y_mercury[:,2],label='mercury')
86      plt.legend()
87      plt.ylabel('z [AU]')
88      plt.xlabel('t [days]')
```

```
    plt.savefig('plots/z-euler.png', dpi=300)
    plt.close()

    fig = plt.figure()
    plt.plot(t,y_mars[:,2],label='mars')
    plt.plot(t,y_earth[:,2],label='earth')
    plt.plot(t,y_venus[:,2],label='venus')
    plt.plot(t,y_mercury[:,2],label='mercury')
    plt.legend()
    plt.ylim(-1,1)
    plt.ylabel('z [AU]')
    plt.xlabel('t [days]')
    plt.savefig('plots/z-euler-zoom.png', dpi=300)
    plt.close()

    fig = plt.figure()
    plt.plot(t,leapfrog[0,:,0]-y_mercury[:,0],label='mercury')
    plt.plot(t,leapfrog[1,:,0]-y_venus[:,0],label='venus')
    plt.plot(t,leapfrog[2,:,0]-y_earth[:,0],label='earth')
    plt.plot(t,leapfrog[3,:,0]-y_mars[:,0],label='mars')
    plt.plot(t,leapfrog[4,:,0]-y_jupiter[:,0],label='jupiter')
    plt.plot(t,leapfrog[5,:,0]-y_saturn[:,0],label='saturn')
    plt.plot(t,leapfrog[6,:,0]-y_uranus[:,0],label='uranus')
    plt.plot(t,leapfrog[7,:,0]-y_neptune[:,0],label='neptune')
    plt.legend()
    plt.ylabel('x [AU]')
    plt.xlabel('t [days]')
    plt.savefig('plots/x-leapfrog-euler.png', dpi=300)
    plt.close()
```

euler.py

In the main part of the code, we solve the orbits of all 8 planets using our function. We then create the same plots we did in Section 1.2 to compare the new implementation with the previous one. As we can see in Figures 7 to 10 the Euler's algorithm managed to solve the orbits there is however the issue of divergence. This is more evident for the planets closer to the Sun which makes sense as they have a smaller orbital period and they therefore achieve more full rotations around the Sun in our simulation time. This divergence can be seen in Figure 7 as the orbits of the inner planets, and especially mercury, get bigger. It also becomes more clear when looking at the orbits of the inner planets in Figure 8. In Figure 9 the amplitudes of the sinusoidal lines of the inner planets get greater as time progresses. In Figure 10 the divergence becomes more clear as we see the z evolution of the inner planets. We also plot the difference in x positions between the Leapfrog and Euler's algorithm in Figure 11. We can see how the divergence affected all the planets. The outer planets got affected less and due to their big orbital radius the difference could not be seen in the previous plots.

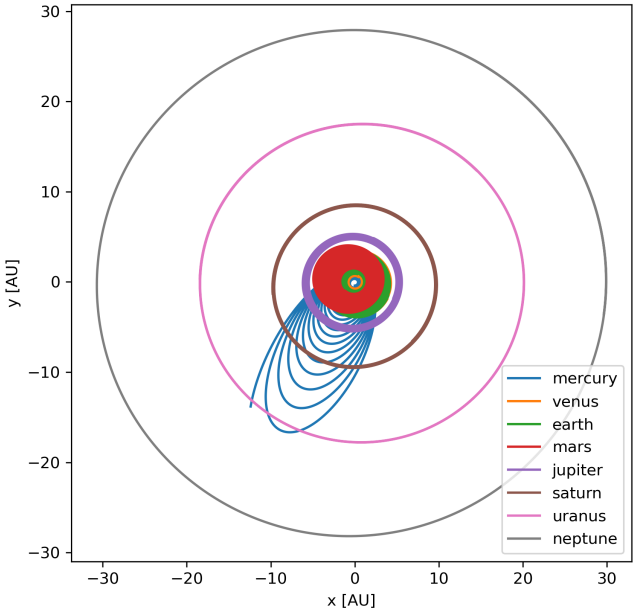

Figure 7: The evolution of the x and y positions of all the 8 planets when integrating using Euler's algorithm. We can see the divergence of the inner planets.

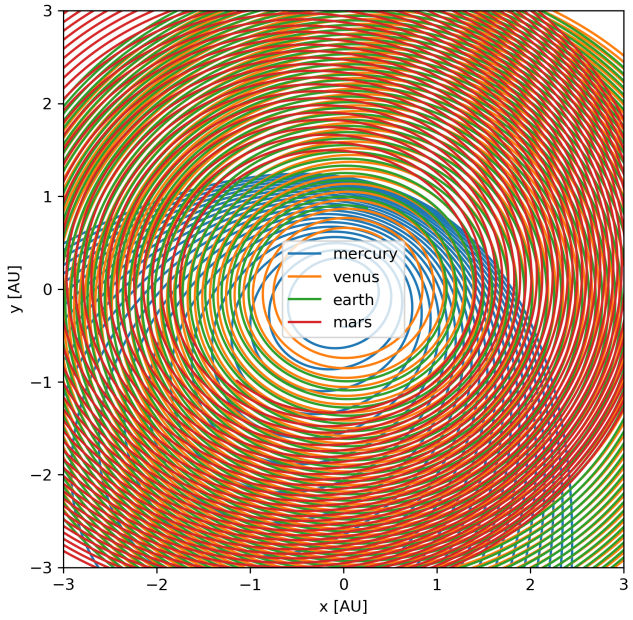

Figure 8: The evolution of the x and y positions of the 4 inner planets when integrating using Euler's algorithm. We can clearly see the divergence in this plot.

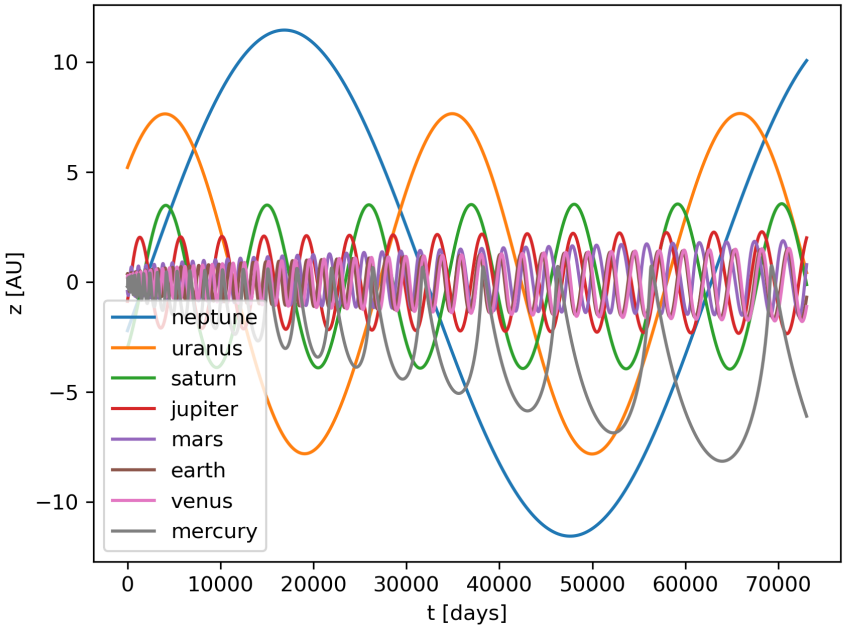

Figure 9: The evolution of the z positions of all the 8 planets when integrating using Euler's algorithm. We can see the divergence of the inner planets.

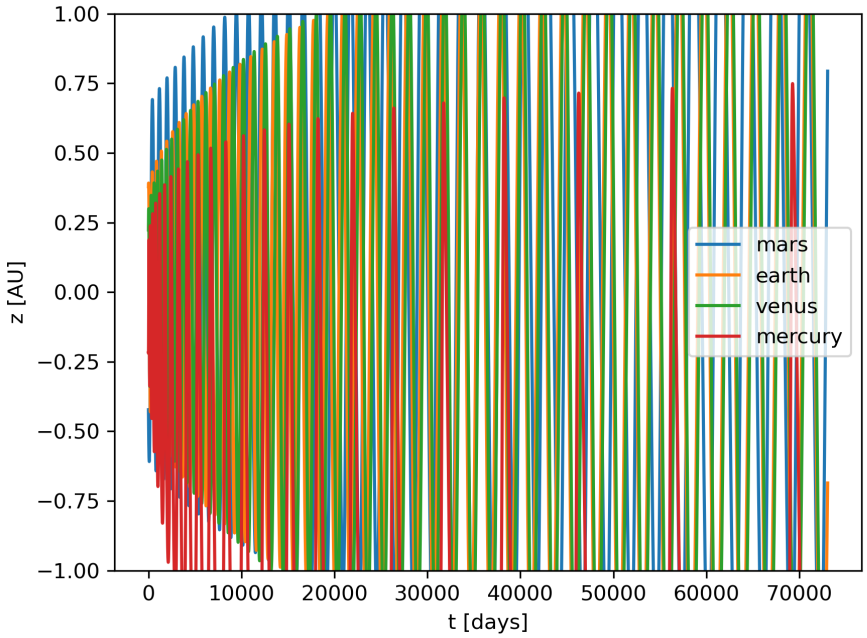

Figure 10: The evolution of the z positions of the 4 inner planets when integrating using Euler's algorithm. We can clearly see the divergence in this plot.

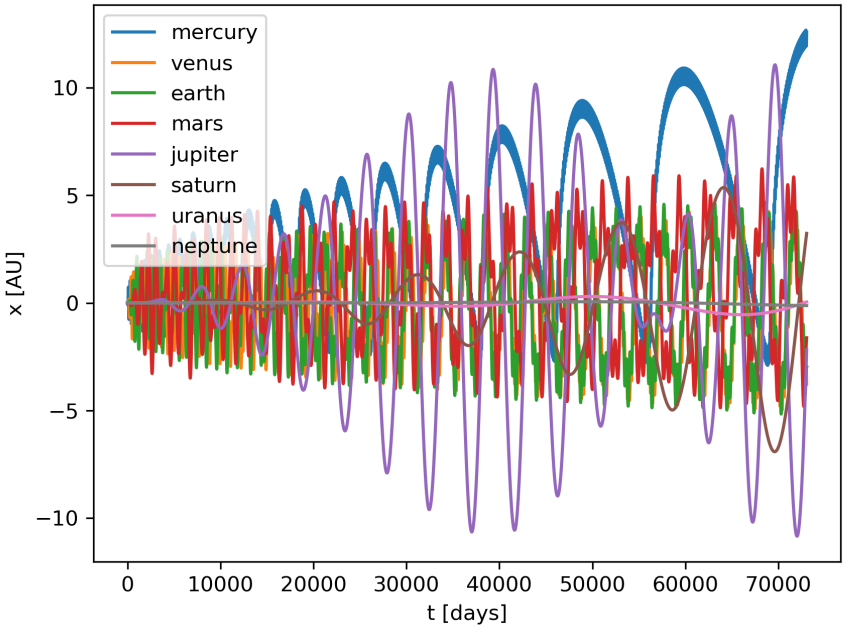

Figure 11: The difference in x positions of all the 8 planets when integrating using the Leapfrog minus when integrating with Euler's algorithm. We can also see the divergence of Euler's algorithm at the inner planets in this plot.

2 Calculating forces with the FFT

In this section, we will calculate the gravitational forces on a grid of 1024 particles with 1 mass unit each. The volume that we are interested in is periodic, which means that the edges connect. We will perform the calculations in the frequency space and then reconvert it to normal space.

2.1 Generate and convert

We first want to generate the volume with all the densities. We do that with the function "generate" which creates a grid $[16 \times 16 \times 16]$ with all the interpolated densities of the particles and also returns the coordinates in a 1D array.

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

def Generate() -> tuple:
    '''
    generates the interpolated densities of 1024 particles
    in a 3D 16x16x16 grid each with a mass of 1 unit
    returns:
    densities: 3D array with the interpolated densities
    grid: 1D array with the coordinates
    '''
    np.random.seed(121)
    n_mesh = 16
    n_part = 1024
    positions = np.random.uniform(low=0, high=n_mesh, size=(3, n_part))
    grid = np.arange(n_mesh) + 0.5
    densities = np.zeros(shape=(n_mesh, n_mesh, n_mesh))
    cellvol = 1.
    for p in range(n_part):
        cellind = np.zeros(shape=(3, 2))
```

```
        dist = np.zeros(shape=(3, 2))
        for i in range(3):
            cellind[i] = np.where((abs(positions[i, p] - grid) < 1) |
                                  (abs(positions[i, p] - grid - 16) < 1) |
                                  (abs(positions[i, p] - grid + 16) < 1))[0]
            dist[i] = abs(positions[i, p] - grid[cellind[i].astype(int)])
        cellind = cellind.astype(int)
        for (x, dx) in zip(cellind[0], dist[0]):
            for (y, dy) in zip(cellind[1], dist[1]):
                for (z, dz) in zip(cellind[2], dist[2]):
                    if dx > 15: dx = abs(dx - 16)
                    if dy > 15: dy = abs(dy - 16)
                    if dz > 15: dz = abs(dz - 16)
                    densities[x, y, z] += (1 - dx)*(1 - dy)*(1 - dz) / cellvol
    return densities, grid

if __name__ == '__main__':
    # generate the densities and calculate the contrast
    densities, grid = Generate()
    rho_aver = 1024/16**3
    contrast = (densities-rho_aver)/rho_aver

    # create the plots
    plot_ind = [4,9,11,14]
    X, Y = np.meshgrid(grid,grid)
    for i in plot_ind:
        fig = plt.figure()
        plt.pcolormesh(X,Y,contrast[:,:,i],cmap='inferno')
        plt.colorbar()
        plt.savefig(f'plots/2d_slice_{i+.5}.png', dpi=300)
        plt.close()

```

generate.py

In the main part of the script we generate the grid and calculate the contrast: $\delta = \frac{\rho-\bar{\rho}}{\bar{\rho}}$. We then create colormap plots for 4 slices which are shown in Figures 12 to 15

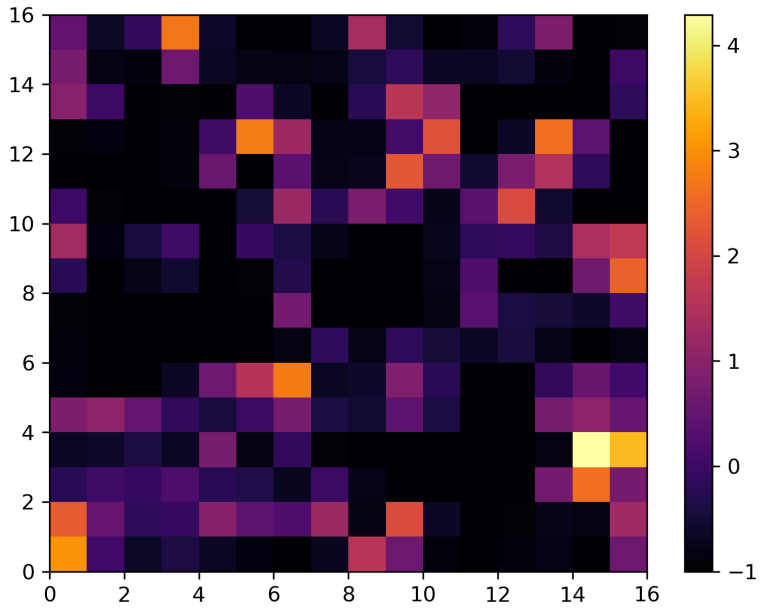

Figure 12: A 2D slice of the grid at z=4.5

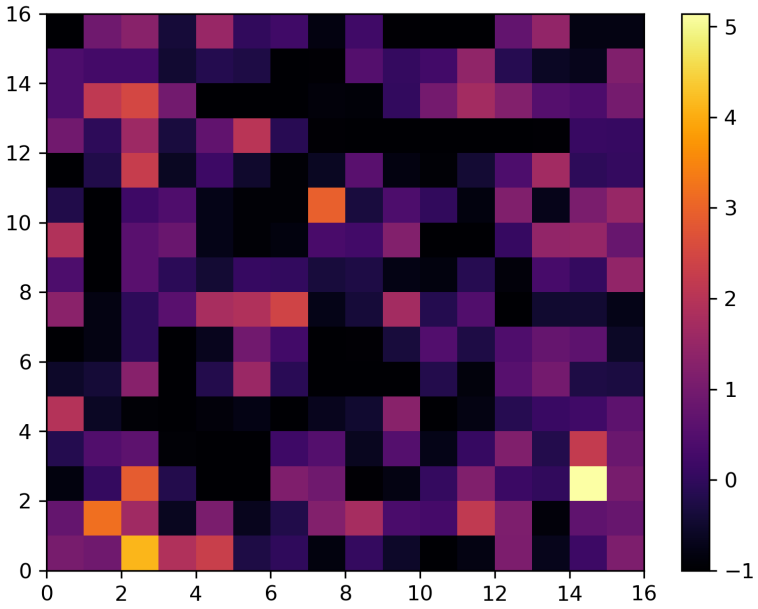

Figure 13: A 2D slice of the grid at z=9.5

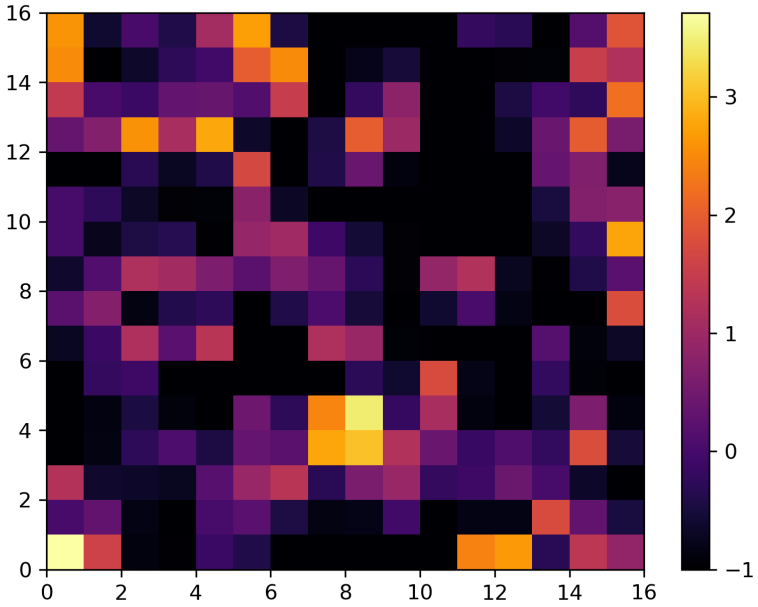

Figure 14: A 2D slice of the grid at z=11.5

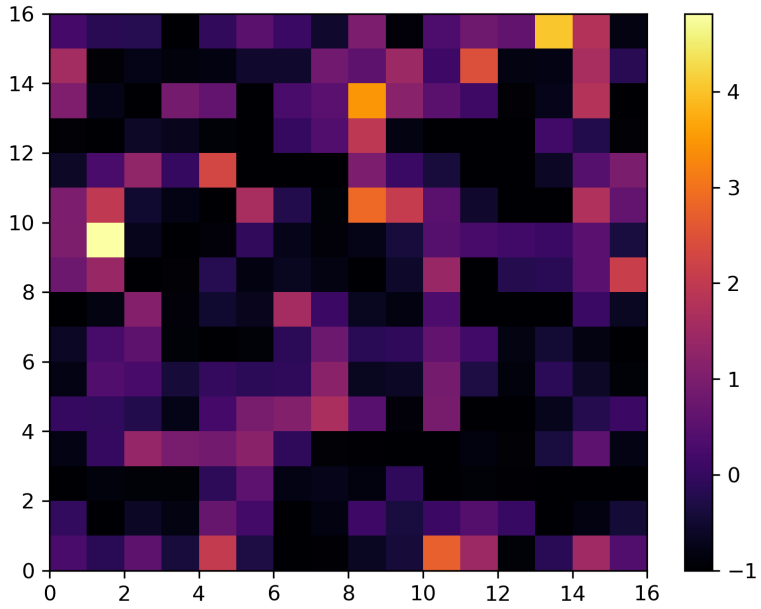

Figure 15: A 2D slice of the grid at z=14.5

2.2 Calculate gravitational forces

We now want to calculate the gravitational potential at each grid point. The Poisson equation is:

$$\nabla^2\Phi = 4\pi G\bar{\rho}(1+\delta) \propto \delta \tag{2}$$

We can calculate the potential in Fourier space

$$-k^2\tilde{\Phi} \propto \tilde{\delta} \tag{3}$$

The negative sign ensures that high-mass regions will have a smaller potential as is the convention for all gravitational potentials. By doing this we can find the Fourier-transformed potential ($\tilde{\Phi}$) and then with the inverse Fourier-transformation find the real potential (Φ).

To perform the calculations we create an FFT routine using the Cooley-Tukey algorithm. Our function "cooley_tukey_fft" calls itself recursively on a slice of the input array with all the even elements and another recursive call on all odd elements. When the recursive calls terminate we loop over k to calculate the Fourier transformation. From all the different definitions for the Fourier, we use

$$\tilde{x}[k] = \sum_{n=0}^{N-1} x[n]e^{-i2\pi k\frac{n}{N}} \tag{4}$$

where we use a summation because we have a discrete Fourier transformation.

We then create "cooley_tukey_ifft" in a similar way to perform the inverse FFT calculations. The difference here is the sign at the calculations of the exponential. We also have to normalize the results, by dividing by 2. The reason for that is that the FFT operation gives a symmetrical result for positive and negative frequencies, which makes our inverse FFT have double the original amplitude if we don't normalize it.

The grid we want to use FFT on is in 3 dimensions. Our functions only operate in one dimension so we create 2 similar functions "fft_3d" and "ifft_3d" that utilize the previous functions to perform a 3D Fourier and inverse Fourier transformation. The way they operate is that they slice the 3D grid into multiple 2D slices, and they perform a 2D Fourier transformation. The 2D Fourier transformation is

simple as we can just take the FFT of all the rows followed by the FFT of all the columns. After having the FFT of all the slices we can take the 1D FFT of all the columns in the 3rd dimension. The final result is the 3D FFT of the whole grid.

```python
import numpy as np
import matplotlib.pyplot as plt
from generate import Generate

def CooleyTukeyFFT(inp: np.ndarray) -> np.ndarray:
    '''
    calculates the fft of an input signal
    using the Cooley-Tukey algorithm
    Uses recursion to calculate the fft
    '''
    x = np.copy(inp)
    n = len(x)
    if n == 1:
        return x
    even = CooleyTukeyFFT(x[::2])
    odd = CooleyTukeyFFT(x[1::2])
    fft = np.zeros(n, dtype=np.complex128)
    for k in range(n // 2):
        fft[k] = even[k] + np.exp(-2j * np.pi * k / n) * odd[k]
        fft[k + n // 2] = even[k] - np.exp(-2j * np.pi * k / n) * odd[k]
    return fft

def CooleyTukeyIFFT(inp: np.ndarray) -> np.ndarray:
    '''
    calculates the inverse fft of an input
    signal using the Cooley-Tukey algorithm
    Uses recursion to calculate the ifft
    '''
    x = np.copy(inp)
    n = len(x)
    if n == 1:
        return x
    even = CooleyTukeyIFFT(x[::2])
    odd = CooleyTukeyIFFT(x[1::2])
    ifft = np.zeros(n, dtype=np.complex128)
    for k in range(n // 2):
        ifft[k] = even[k] + np.exp(+2j * np.pi * k / n) * odd[k]
        ifft[k + n // 2] = even[k] - np.exp(+2j * np.pi * k / n) * odd[k]
    return ifft / 2

def FFT3D(values: np.ndarray) -> np.ndarray:
    '''
    calculates the 3D fft of an input using
    the Cooley-Tukey algorithm
    Use the CooleyTukeyFFT function to calculate the fft
    '''
    f1 = np.zeros_like(values,dtype=np.complex128)
    f2 = np.zeros_like(values,dtype=np.complex128)
    for i in range(len(values)):
        flat = np.array(values[:,:,i],dtype=np.complex128)
        for j in range(len(values)):
            flat[j,:] = CooleyTukeyFFT(flat[j,:])
        for j in range(len(values)):
            flat[:,j] = CooleyTukeyFFT(flat[:,j])
        f1[:,:,i] = flat
    for i in range(len(values)):
        for j in range(len(values)):
            f2[i,j,:] = CooleyTukeyFFT(f1[i,j,:])
    return f2

def IFFT3D(values: np.ndarray) -> np.ndarray:
    '''
    calculates the 3D inverse fft of an input
    using the Cooley-Tukey algorithm
    Use the CooleyTukeyIFFT function to calculate the ifft
    '''
```

```
    f1 = np.zeros_like(values,dtype=np.complex128)
    f2 = np.zeros_like(values,dtype=np.complex128)
    for i in range(len(values)):
        flat = np.array(values[:,:,i],dtype=np.complex128)
        for j in range(len(values)):
            flat[j,:] = CooleyTukeyIFFT(flat[j,:])
        for j in range(len(values)):
            flat[:,j] = CooleyTukeyIFFT(flat[:,j])
        f1[:,:,i] = flat
    for i in range(len(values)):
        for j in range(len(values)):
            f2[i,j,:] = CooleyTukeyIFFT(f1[i,j,:])
    return f2

if __name__ == '__main__':
    # generate the densities and calculate the contrast
    densities, grid = Generate()
    rho_aver = 1024/16**3
    contrast = (densities-rho_aver)/(rho_aver)

    # fft of contrast densities
    fourier = FFT3D(contrast)

    # getting k
    N = 16
    kx = np.concatenate((np.arange(0, N // 2), np.arange(-N // 2, 0)))
    ky = np.concatenate((np.arange(0, N // 2), np.arange(-N // 2, 0)))
    kz = np.concatenate((np.arange(0, N // 2), np.arange(-N // 2, 0)))
    KX, KY, KZ = np.meshgrid(kx, ky,kz)

    # calculate the fourier potential
    small_value = 1e-10 # avoid division by 0
    fourier = -fourier/(KX**2 + KY**2+KZ**2+small_value)

    # calculate the potential with inverse fft
    potential = np.real(IFFT3D(fourier))

    # create the plots
    plot_ind = [4,9,11,14]
    X, Y = np.meshgrid(grid,grid)
    for i in plot_ind:
        fig = plt.figure()
        plt.pcolormesh(X,Y,np.log10(np.abs(fourier[:,:,i])),cmap='inferno')
        plt.colorbar()
        plt.savefig(f'plots/fourier_{i+.5}.png', dpi=300)
        plt.close()

        fig = plt.figure()
        plt.pcolormesh(X,Y,potential[:,:,i],cmap='inferno_r')
        plt.colorbar()
        plt.savefig(f'plots/potential_{i+.5}.png', dpi=300)
        plt.close()


```

fft.py

In the main part of the script, we calculate the contrast in the same way we did at Section 2.1. We then take the 3D FFT of that result. We now want to calculate the Fourier-transformed potential ($\tilde{\Phi}$) using Equation (3). For that, we need to calculate the wavenumber k for each position in the Fourier-transformed density grid. Because of symmetries in our FFT routine, the results are returned with first the term for k=0 followed by all the positive terms of k. At $\tilde{x}[n/2]$ we have our greatest positive frequency term and then we begin with the negative ones starting from the smallest value. We create 3 arrays for all the 3 dimensions of our grid using this definition and then create a grid for each of the individual components (KX, KY, KZ). Because we have to divide our Fourier transformed densities with the square of the norm of the wavenumber vector we will come across some 0 terms. To avoid any errors we add a sufficiently small value.

We then take the inverse FFT of the Fourier-transformed potential and take the real part. The new

grid array now contains the full potential of the original grid. We create plots for the same 2D slices we used in Section 2.1 for the potential (Figures 20 to 23) and for the logarithm of the absolute value of the Fourier-transformed potential (Figures 16 to 19).

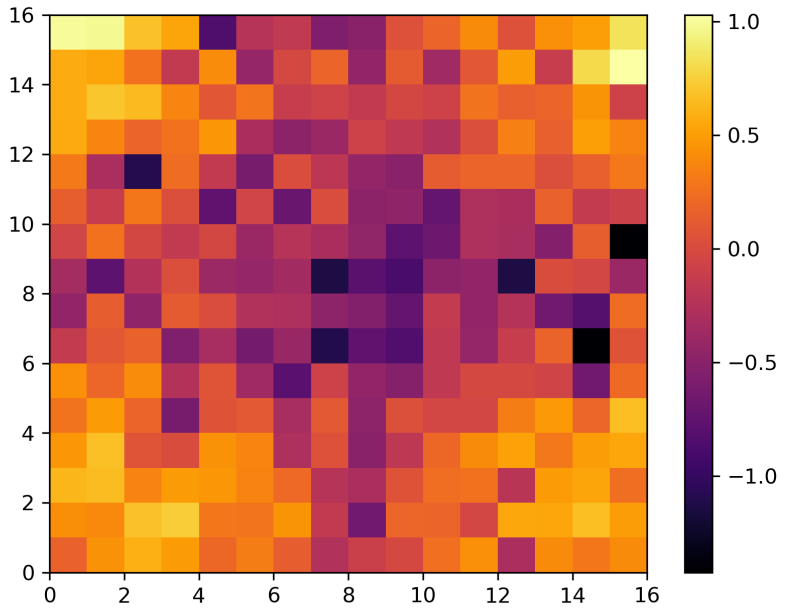

Figure 16: A 2D slice of the logarithm of the absolute value of the Fourier-transformed potential at z=4.5

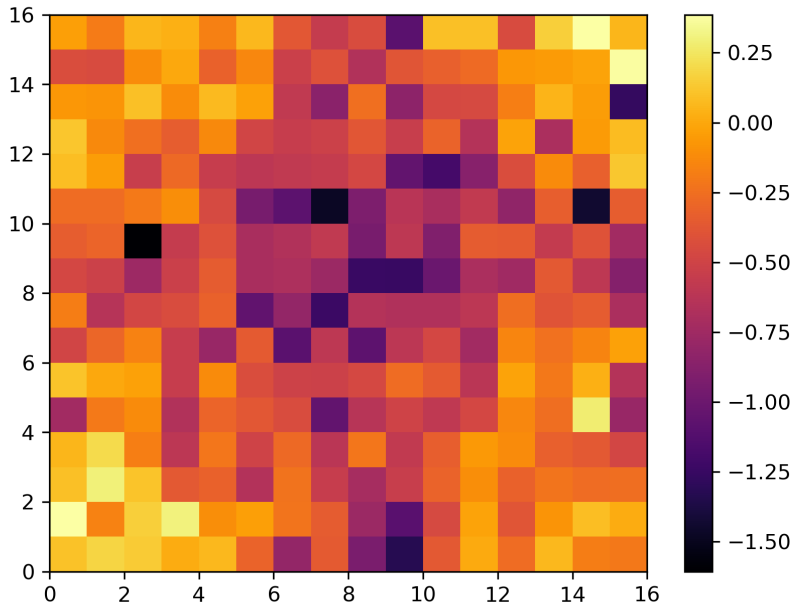

Figure 17: A 2D slice of the logarithm of the absolute value of the Fourier-transformed potential at z=9.5

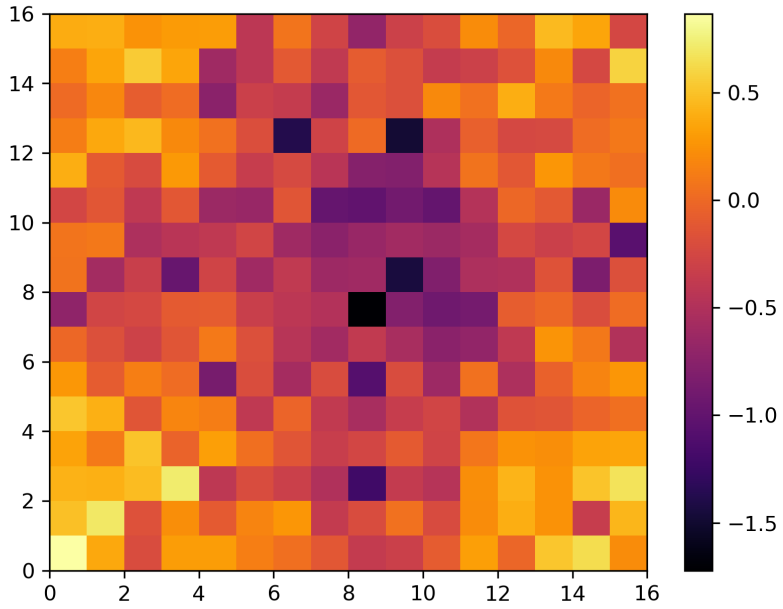

Figure 18: A 2D slice of the logarithm of the absolute value of the Fourier-transformed potential at z=11.5

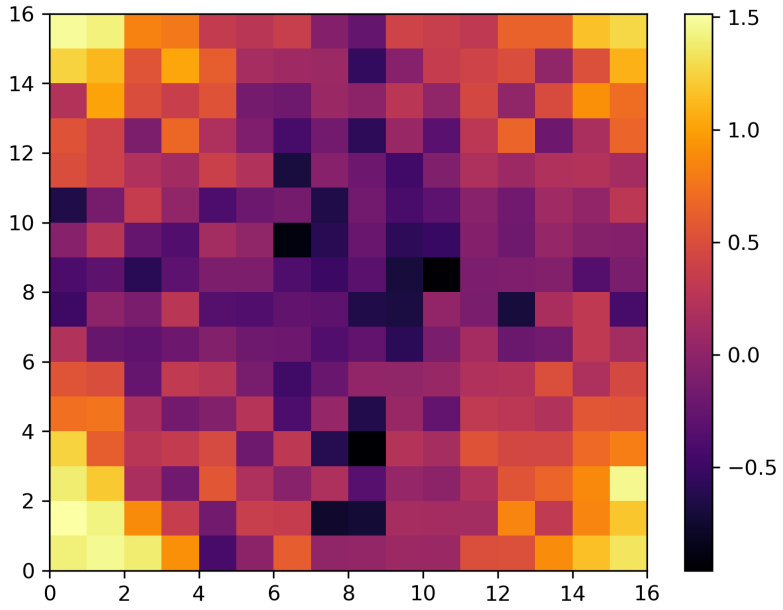

Figure 19: A 2D slice of the logarithm of the absolute value of the Fourier-transformed potential at z=14.5

As we can see our results for the potential (Figures 20 to 23) make intuitive sense when we compare them with the density slices in Figures 12 to 15. We see that high-density regions lead to a smaller

gravitational potential because we used the minus sign in Equation (3). We use an inverse colormap to highlight those regions.

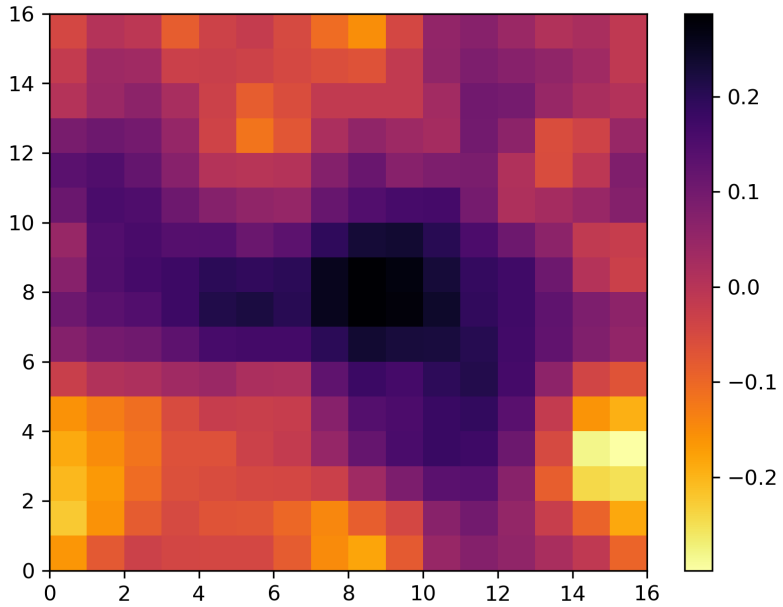

Figure 20: A 2D slice of the potential at z=4.5

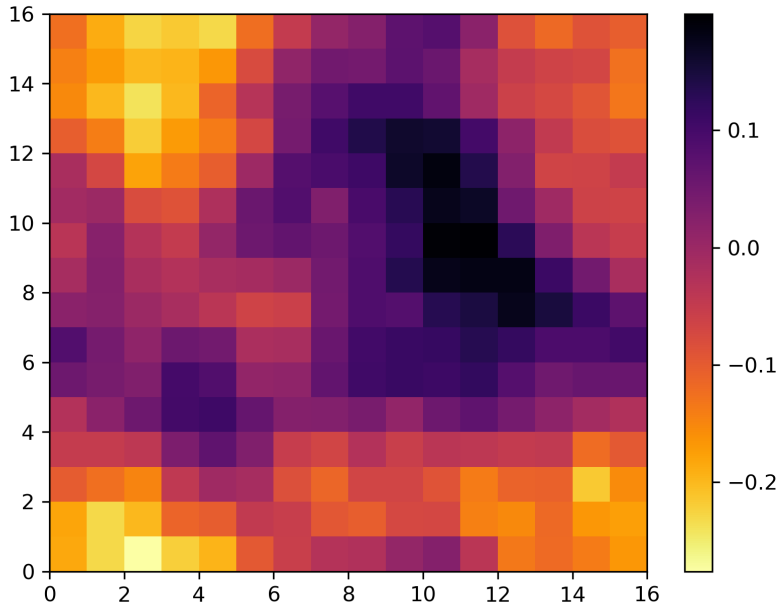

Figure 21: A 2D slice of the potential at z=9.5

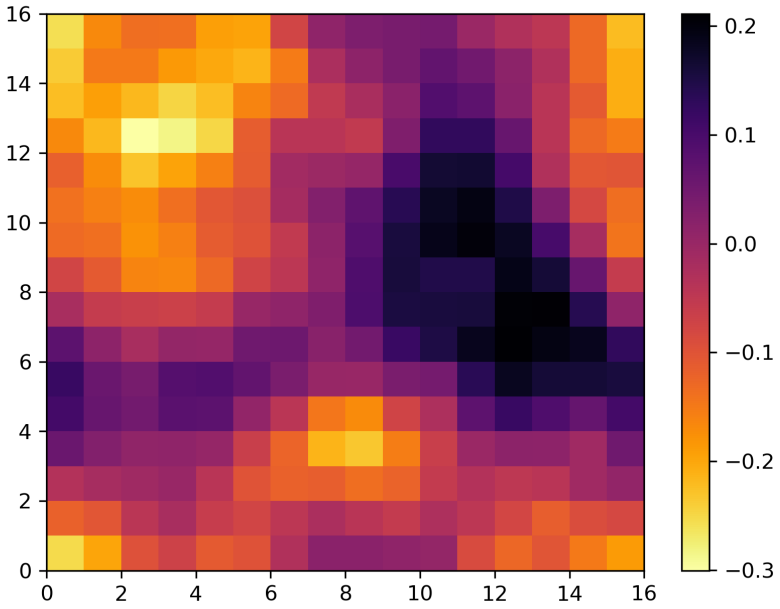

Figure 22: A 2D slice of the potential at z=11.5

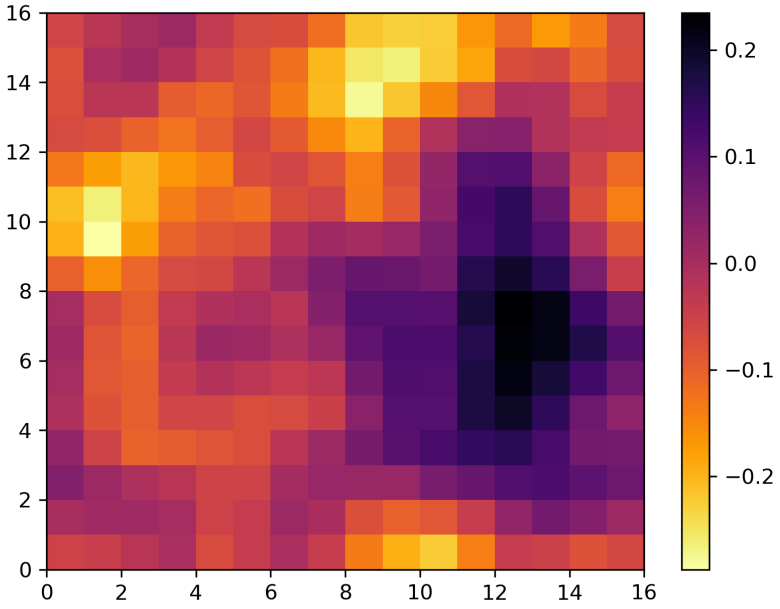

Figure 23: A 2D slice of the potential at z=14.5

3 Spiral and elliptical galaxies

In this section, we will use Logistic Regression to classify galaxies in either spirals or ellipticals using different sets of features.

3.1 Preparing the dataset

Our dataset contains 5 columns. The last column consists of the labels with a flag that indicates whether it is a spiral (1) or elliptical (0). The first four columns contain different parameters of each galaxy. First, we have a parameter indicating how much the galaxy is dominated by ordered rotation, then an estimate of their color followed by a measure of how extended each galaxy is, and lastly the flux of an emission line used to measure the star formation rate of galaxies.

To prepare the dataset we want to create an input array with shape $[m \times n]$ where m is the length of the dataset and n is the number of features (in our case 4). We want to normalize each feature to have a mean of 0 and a standard deviation of 1. We do that with the following formula:

$$x \leftarrow \frac{x - \text{mean}(x)}{\text{std}(x)} \tag{5}$$

```python
if __name__ == '__main__':
    import numpy as np
    import matplotlib.pyplot as plt

    # get data from file
    data = np.loadtxt('data/galaxy_data.txt').T
    x = data[:-1]       # inputs
    y = data[-1]        # labels

    # feature scaling (mean=0, std=1)
    # plus create plots
    for i in range(len(data)-1):
        x[i] = (x[i] - x[i].mean()) / x[i].std()

        fig = plt.figure()
        plt.hist(x[i],bins=20,color='crimson',edgecolor='black')
        plt.yscale('log')
        plt.xlabel('Scaled Feature')
        plt.ylabel('N')
        plt.savefig(f'plots/distr_{i}.png')
        plt.close()

    # reverse it for shape [m x n]
    x = x.T

    # save in .txt file
    np.savetxt('output/input.txt',x)
```

prepare.py

We save the input array and we show the first 10 lines of the text file. We also create histogram plots of the 4 distributions in Figures 24 to 27. We use a logarithmic scale to better indicate the outliers and regions with small values of N.

```
-1.581862133006098681e+00 -6.612307828919693729e-03 -3.475533387671284058e-02
    4.149921715612928282e-03
1.574559067587914640e+00 -7.944354060046822097e-01 2.031450234565139734e-01
    1.605487869858183633e-02
1.531990475165863508e+00 8.573405014974248006e-01 -2.079754178048100477e-01
    4.166315770216809378e-02
1.297215471156623057e+00 1.226267345173265078e+00 1.858865005815529270e-01
    6.825469423304870997e-04
5.181259591131069930e-01 8.257714425578099871e-01 -2.002589858418064583e-01
    1.555766189074690373e-02
-1.647524013432828394e+00 -2.909640578232345343e-01 -8.488083597139754743e-02
    1.855013775209162245e-03
```

```
−1.31249596036035876e+00  −9.75653663706389063e−01  6.45457637967138537e−02
    6.77265082088561367e−03
9.41830399450574313e−02  −9.71515724051743579e−01  −1.52518348470376403e−01
    7.74501347579524719e−03
−7.30829366224539534e−01  −1.52014862354793889e+00  −1.65440314979844294e−01
    1.17544049384444831e−02
−1.19803551423812515e+00  −7.74899524272912754e−01  −1.55388867315653545e−01
    9.59689257219212451e−03
```

output/input.txt

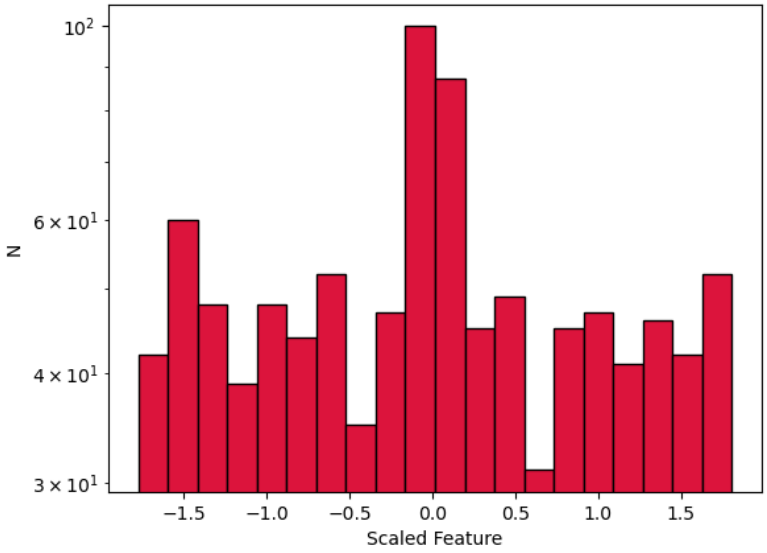

Figure 24: The distribution of the 1st column after applying feature scaling.

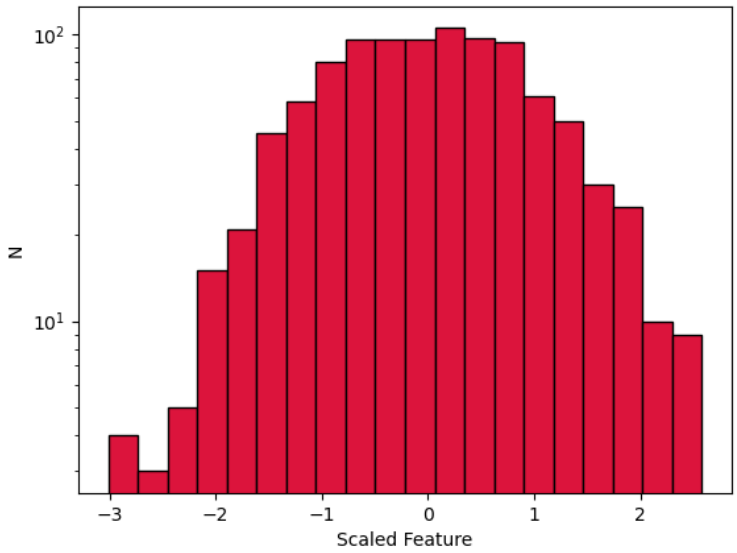

Figure 25: The distribution of the 2nd column after applying feature scaling.

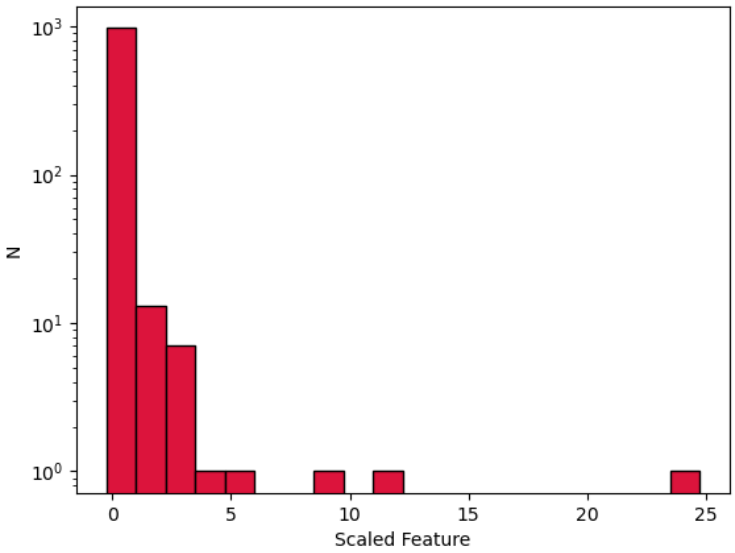

Figure 26: The distribution of the 3rd column after applying feature scaling.

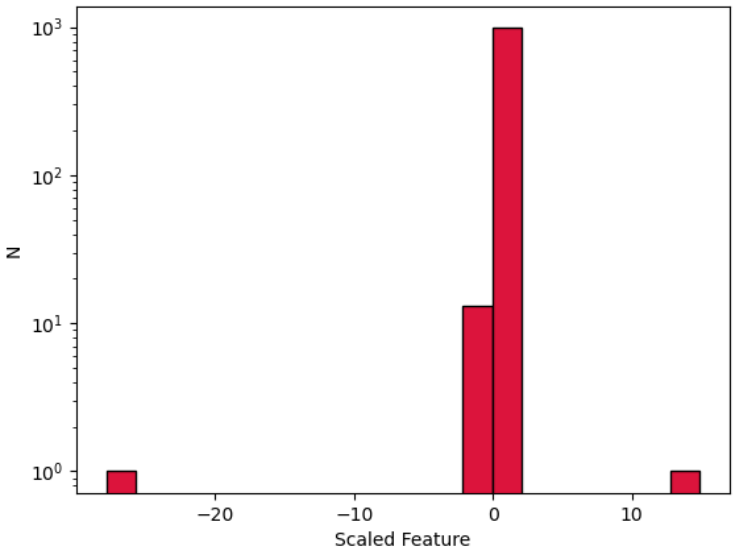

Figure 27: The distribution of the 4th column after applying feature scaling.

3.2 Logistic Regression

We now want to perform a Logistic Regression to classify the galaxies in spirals and ellipticals. The first step is to have a minimization algorithm to find the best weights. We use the "bfgs" function that utilizes the Quasi-Newton method of BFGS. The implementation that we used follows the pseudocode and equations from ? (?) (pages 136-140). For the line minimization that is required, we use the "golden_search" function that utilizes the golden search algorithm for function minimization. The implementation is the same as we did in previous assignments with minor changes for compatibility with "bfgs".

Now that we have our minimization routine we will do the logistic regression. We define the "sigmoid"

function to calculate the sigmoid and define the loss function of the logistic regression in "loss_fun":

$$J(\theta) = -y \log(sig(X \cdot \theta)) - (1 - y) \log(1 - sig(X \cdot \theta)) \tag{6}$$

where X is the input, θ are the model parameters and y the true labels. We also define the gradient of the loss function in "grad_loss":

$$J'(\theta) = \frac{1}{m} X^T \cdot [sig(X \cdot \theta) - y] \tag{7}$$

In the "fit" function we perform the training of our model by initializing the weights at 1 and then using the minimization function to find the best values for the weights by minimizing the loss function. We also define a function for the plotting of the loss function.

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

def GoldenSearch(func: callable, init1: float, init2: float, tol: float = 1e-5, maxiter:
    int = 1000) -> float:
    '''
    minimization routine that utilizes the golden search
    algorithm
    input:
    func: function to find local minimum
    init1, init2: initial points
    tol: relative precision of local minimum
    maxiter: maximum number of iterations
    returns:
    x-value of a local minimum
    '''
    phi = (1 + math.sqrt(5))/2
    R = 1/phi
    C=1-R
    a = init1
    d = init2
    b = (a + d)/2
    if abs(d-b) > abs(b-a):
        c=b+C*(d-b)
    else:
        c=b
        b=b+C*(a-b)
    fb = func(b)
    fc= func(c)
    for i in range(maxiter):
        if abs(d-a)<tol*(abs(b)+abs(c)):
            if fb<fc:
                return b
            else:
                return c
        if fc<fb:
            a = b
            b = c
            c = R*c+C*d
            fb = fc
            fc = func(c)
        else:
            d = c
            c = b
            b = R*b+C*a
            fc = fb
            fb = func(b)
    return (b+c)/2

def BFGS(f: callable, grad_f: callable, x0: np.ndarray, tol: float = 1e-6, max_iter: int
    = 1000) -> np.ndarray:
    '''
    minimization routine that utilizes the Quasi-Newton
    BFGS algorithm
    input:
```

```
    f: function to find local minimum
    grad_f: analytical derivative of f
    x0: initial point
    tol: relative precision of local minimum
    maxiter: maximum number of iterations
    returns:
    x-value of a local minimum
    '''
    n = len(x0)
    x = x0
    H = np.eye(n)
    cost_fun = []
    for i in range(max_iter):
        cost_fun.append(f(x))
        grad = grad_f(x)
        pk = -np.dot(H, grad)
        alpha = GoldenSearch(lambda alpha: f(x+alpha*pk), 0.0, 1.0, tol=1e-3)
        x_new = x + alpha * pk
        s = x_new - x
        if np.sqrt(np.sum((s)**2))<tol:
            return x_new, cost_fun
        y = grad_f(x_new) - grad
        rho = 1.0 / np.dot(s, y)
        A1 = np.eye(n) - rho * np.outer(s, y)
        A2 = np.eye(n) - rho * np.outer(y, s)
        H = np.dot(A1, np.dot(H, A2)) + rho * np.outer(s, s)
        x = x_new
    raise ValueError("Maximum number of iterations exceeded.")

def Sigmoid(z: np.ndarray) -> np.ndarray:
    '''
    sigmoid function
    '''
    return 1 / (1 + np.exp(-z))

def LossFun(weights: np.ndarray, X: np.ndarray, y: np.ndarray) -> float:
    '''
    loss function for logistic regression
    '''
    theta = weights
    linear = np.dot(X,theta)
    sig = Sigmoid(linear)
    eps = 1e-25 # avoid log(0)
    j = -y*np.log(sig+eps) - (1-y)*np.log(1-sig+eps)
    return np.mean(j)

def GradLoss(weights: np.ndarray, X: np.ndarray, y: np.ndarray) -> np.ndarray:
    '''
    analytic derivative of the loss function for logistic regression
    '''
    dw = (1/X.shape[0])*np.dot(X.T,(Sigmoid(np.dot(X,weights))-y))
    return dw

def Fit(X: np.ndarray, y: np.ndarray, max_iter: int = 1000) -> tuple:
    '''
    logistic regression fitting
    '''
    n_features = X.shape[1]
    weights = np.ones(n_features)
    f = lambda x: LossFun(x,X,y)
    grad_f = lambda x: GradLoss(x,X,y)
    weights, cost_fun = BFGS(f,grad_f,x0=weights,max_iter=max_iter)
    return weights, cost_fun


def TrainingPlot(y: np.ndarray, name: str = 'training', fig: str = 'png', dpi: int =
    300):
    '''
    function to create cost function convergance plot
    '''
```

```
    plt.figure()
    plt.plot(np.arange(len(y[0])),y[0],label='col 1-2')
    plt.plot(np.arange(len(y[1])),y[1],label='col 1-3')
    plt.plot(np.arange(len(y[2])),y[2],label='col 1-4')
    plt.plot(np.arange(len(y[3])),y[3],label='col 2-3')
    plt.plot(np.arange(len(y[4])),y[4],label='col 2-4')
    plt.plot(np.arange(len(y[5])),y[5],label='col 3-4')
    plt.plot(np.arange(len(y[6])),y[6],label='col 1-2-3')
    plt.plot(np.arange(len(y[7])),y[7],label='col 1-2-4')
    plt.plot(np.arange(len(y[8])),y[8],label='col 1-3-4')
    plt.plot(np.arange(len(y[9])),y[9],label='col 2-3-4')
    plt.plot(np.arange(len(y[10])),y[10],label='col 1-2-3-4')
    plt.ylabel('Cost Function')
    plt.xlabel('Number of iterations')
    plt.legend()
    plt.savefig(f'plots/{name}.{fig}',dpi=dpi)
    plt.close()


if __name__ == '__main__':
    #load data
    data = np.loadtxt('data/galaxy_data.txt').T
    y = data[-1]
    x = np.loadtxt('output/input.txt')

    # Training

    # combinations of 2 features
    Xx = x[:,0:2]
    weights_0, cost_fun_0=Fit(Xx,y)

    Xx = np.array([x[:,0],x[:,2]]).T
    weights_1, cost_fun_1=Fit(Xx,y)

    Xx = np.array([x[:,0],x[:,3]]).T
    weights_2, cost_fun_2=Fit(Xx,y)

    Xx = x[:,1:3]
    weights_3, cost_fun_3=Fit(Xx,y)

    Xx = np.array([x[:,1],x[:,3]]).T
    weights_4, cost_fun_4=Fit(Xx,y)

    Xx = x[:,2:4]
    weights_5, cost_fun_5=Fit(Xx,y)

    # combinations of 3 features
    Xx = x[:,0:3]
    weights_6, cost_fun_6=Fit(Xx,y)

    Xx = np.array([x[:,0],x[:,1],x[:,3]]).T
    weights_7, cost_fun_7=Fit(Xx,y)

    Xx = np.array([x[:,0],x[:,2],x[:,3]]).T
    weights_8, cost_fun_8=Fit(Xx,y)

    Xx = x[:,1:4]
    weights_9, cost_fun_9=Fit(Xx,y)

    # all 4 features
    Xx = x[:,:]
    weights_10, cost_fun_10=Fit(Xx,y)

    # create cost function convergance plot
    TrainingPlot([cost_fun_0,cost_fun_1,cost_fun_2,cost_fun_3,cost_fun_4
        ,cost_fun_5,cost_fun_6,cost_fun_7,cost_fun_8,cost_fun_9,cost_fun_10])

    # save weights for other scripts
    np.save('output/weights-2',np.array([weights_0,weights_1,weights_2,weights_3,
    weights_4,weights_5]))
```

```
193     np.save('output/weights-3',np.array([weights_6,weights_7,weights_8,weights_9]))
194     np.save('output/weights-4',weights_10)
```

classification.py

In the main part of the script, we load the data, using the rescaled features we calculated before. We then perform the fitting for all possible combinations of different features. In Figure 28 we can see the value of the loss function during the training. As we can see including different features is allowing us to reach even lower values.

The different distinct groups that appear in Figure 28 are because of the different combinations of features. The best performance is when we include the 1st and 2nd features. The worst performance is when neither is present. Including the 1st without the 2nd lead to better performance than the opposite. The reason these two features give such a performance improvement is that they have a better distribution as was seen in Section 3.1, and they are also better indicators of whether a galaxy is spiral or elliptical.

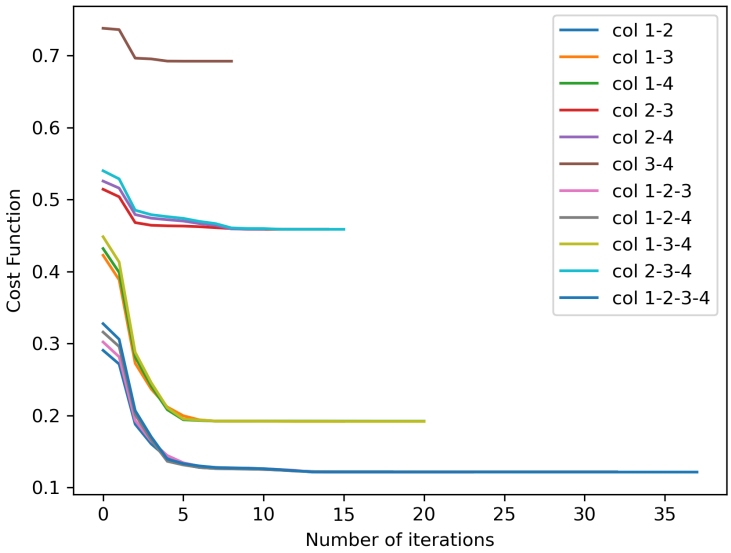

Figure 28:

3.3 Evaluate

We now want to evaluate our models. We create the "predict" function to generate predictions with the weights of our model and the input features. We define the "evaluate" function that utilizes the predictions to calculate the number of true/false positives/negatives. The "f_score" function calculates the F_β value. We then create two functions to plot two features against each other with the decision boundary.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from classification import Sigmoid
4
5  def Predict(weights: np.ndarray, X: np.ndarray) -> np.ndarray:
6      '''
7      generate predictions for logistic regression models
8      input:
9      weights: the values of the weights of the model
10     X: the input array
11     returns:
12     predicted labels of the model
13     '''
```

```
    theta = weights
    linear = np.dot(X,theta)
    predicted_probabilities = Sigmoid(linear)
    predicted_labels = (predicted_probabilities >= 0.5).astype(int)
    return predicted_labels

def Evaluate(weights: np.ndarray, X: np.ndarray, y: np.ndarray) -> tuple:
    '''
    calculates the true positives, true
    negatives, false positives and false
    negatives of the model
    '''
    predictions = Predict(weights, X)
    TP = TN = FP = FN = 0
    for i,pred in enumerate(predictions):
        if pred == y[i]:
            if pred == 1:
                TP +=1
            else:
                TN +=1
        else:
            if pred == 1:
                FP +=1
            else:
                FN +=1
    return TP, TN, FP, FN

def FScore(TP: int, FP: int, FN: int, beta: float = 1.) -> float:
    '''
    calculate the F score
    by default beta = 1
    '''
    prec = TP/(TP+FP)
    rec = TP/(TP+FN)
    f_score = ((1+beta**2)*prec*rec)/(beta**2 * prec + rec)
    return f_score

def Boundary(X: np.ndarray, weights: np.ndarray) -> tuple:
    '''
    creates the decision boundaries of
    logistic regression
    '''
    x = X[:,0]
    a = weights[0]
    b = weights[1]
    return x,(-a*x)/b

def BoundaryPlot(weights: np.ndarray, X: np.ndarray, y: np.ndarray, xlim: tuple = None,
                 ylim: tuple = None, name: str = 'test', fig: str = 'png', dpi: int =
    300):
    '''
    creates a plot of 2 features with the
    decision boundary and the labels
    '''
    x_b, y_b = Boundary(X,weights)
    scatter = plt.scatter(X[:,0],X[:,1],c=y,marker='.')
    plt.legend(handles=scatter.legend_elements()[0], labels=['0','1'])
    plt.plot(x_b,y_b,color='red')
    if xlim == None:
        plt.xlim(np.amin(X[:,0]),np.amax(X[:,0]))
    else:
        plt.xlim(xlim[0],xlim[1])
    if ylim == None:
        plt.ylim(np.amin(X[:,1]),np.amax(X[:,1]))
    else:
        plt.ylim(ylim[0],ylim[1])
    plt.xlabel('Scaled Feature')
    plt.ylabel('Scaled Feature')
    plt.savefig(f'plots/{name}.{fig}',dpi=dpi)
    plt.close()
```

```
83
84
85  if __name__ == '__main__':
86      # load data
87      data = np.loadtxt('data/galaxy_data.txt').T
88      y = data[-1]
89      x = np.loadtxt('output/input.txt')
90
91      # load weights
92      weights_2 = np.load('output/weights-2.npy')
93      weights_3 = np.load('output/weights-3.npy')
94      weights_4 = np.load('output/weights-4.npy')
95
96      # evaluations and boundary plots
97
98      # combinations of 2 features
99      Xx = x[:,0:2]
100     BoundaryPlot(weights_2[0],Xx,y,name='boundary_1-2')
101     tp_0, tn_0, FP_0, FN_0 = Evaluate(weights_2[0],Xx,y)
102     acc_0 = (tp_0+tn_0)/len(y)
103     f_0 = FScore(tp_0,FP_0,FN_0)
104
105     Xx = np.array([x[:,0],x[:,2]]).T
106     BoundaryPlot(weights_2[1],Xx,y,ylim=[-.5,1.],name='boundary_1-3')
107     tp_1, tn_1, FP_1, FN_1 = Evaluate(weights_2[1],Xx,y)
108     acc_1 = (tp_1+tn_1)/len(y)
109     f_1 = FScore(tp_1,FP_1,FN_1)
110
111     Xx = np.array([x[:,0],x[:,3]]).T
112     BoundaryPlot(weights_2[2],Xx,y,ylim=[-.1,.1],name='boundary_1-4')
113     tp_2, tn_2, FP_2, FN_2 = Evaluate(weights_2[2],Xx,y)
114     acc_2 = (tp_2+tn_2)/len(y)
115     f_2 = FScore(tp_2,FP_2,FN_2)
116
117     Xx = x[:,1:3]
118     BoundaryPlot(weights_2[3],Xx,y,ylim=[-.5,1.],name='boundary_2-3')
119     tp_3, tn_3, FP_3, FN_3 = Evaluate(weights_2[3],Xx,y)
120     acc_3 = (tp_3+tn_3)/len(y)
121     f_3 = FScore(tp_3,FP_3,FN_3)
122
123     Xx = np.array([x[:,1],x[:,3]]).T
124     BoundaryPlot(weights_2[4],Xx,y,ylim=[-.1,.1],name='boundary_2-4')
125     tp_4, tn_4, FP_4, FN_4 = Evaluate(weights_2[4],Xx,y)
126     acc_4 = (tp_4+tn_4)/len(y)
127     f_4 = FScore(tp_4,FP_4,FN_4)
128
129     Xx = x[:,2:4]
130     BoundaryPlot(weights_2[5],Xx,y,xlim=[-.5,1.],ylim=[-.1,.1],name='boundary_3-4')
131     tp_5, tn_5, FP_5, FN_5 = Evaluate(weights_2[5],Xx,y)
132     acc_5 = (tp_5+tn_5)/len(y)
133     f_5 = FScore(tp_5,FP_5,FN_5)
134
135     # combinations of 3 features
136     Xx = x[:,0:3]
137     tp_6, tn_6, FP_6, FN_6 = Evaluate(weights_3[0],Xx,y)
138     acc_6 = (tp_6+tn_6)/len(y)
139     f_6 = FScore(tp_6,FP_6,FN_6)
140
141     Xx = np.array([x[:,0],x[:,1],x[:,3]]).T
142     tp_7, tn_7, FP_7, FN_7 = Evaluate(weights_3[1],Xx,y)
143     acc_7 = (tp_7+tn_7)/len(y)
144     f_7 = FScore(tp_7,FP_7,FN_7)
145
146     Xx = np.array([x[:,0],x[:,2],x[:,3]]).T
147     tp_8, tn_8, FP_8, FN_8 = Evaluate(weights_3[2],Xx,y)
148     acc_8 = (tp_8+tn_8)/len(y)
149     f_8 = FScore(tp_8,FP_8,FN_8)
150
151     Xx = x[:,1:4]
152     tp_9, tn_9, FP_9, FN_9 = Evaluate(weights_3[3],Xx,y)
```

```
    acc_9 = (tp_9+tn_9)/len(y)
    f_9 = FScore(tp_9,FP_9,FN_9)

    # all 4 features
    Xx = x[:,:]
    tp_10, tn_10, FP_10, FN_10 = Evaluate(weights_4,Xx,y)
    acc_10 = (tp_10+tn_10)/len(y)
    f_10 = FScore(tp_10,FP_10,FN_10)

    # print results
    print(f'col  1 , 2 |TP:{tp_0:^3}|TN:{tn_0:^3}|FP:{FP_0:^3}|FN:{FN_0:^3}|accuaracy:{acc_0:^5}|F_1:{f_0:.3}')
    print(f'col  1 , 3 |TP:{tp_1:^3}|TN:{tn_1:^3}|FP:{FP_1:^3}|FN:{FN_1:^3}|accuaracy:{acc_1:^5}|F_1:{f_1:.3}')
    print(f'col  1 , 4 |TP:{tp_2:^3}|TN:{tn_2:^3}|FP:{FP_2:^3}|FN:{FN_2:^3}|accuaracy:{acc_2:^5}|F_1:{f_2:.3}')
    print(f'col  2 , 3 |TP:{tp_3:^3}|TN:{tn_3:^3}|FP:{FP_3:^3}|FN:{FN_3:^3}|accuaracy:{acc_3:^5}|F_1:{f_3:.3}')
    print(f'col  2 , 4 |TP:{tp_4:^3}|TN:{tn_4:^3}|FP:{FP_4:^3}|FN:{FN_4:^3}|accuaracy:{acc_4:^5}|F_1:{f_4:.3}')
    print(f'col  3 , 4 |TP:{tp_5:^3}|TN:{tn_5:^3}|FP:{FP_5:^3}|FN:{FN_5:^3}|accuaracy:{acc_5:^5}|F_1:{f_5:.3}')
    print(f'col  1,2,3 |TP:{tp_6:^3}|TN:{tn_6:^3}|FP:{FP_6:^3}|FN:{FN_6:^3}|accuaracy:{acc_6:^5}|F_1:{f_6:.3}')
    print(f'col  1,2,4 |TP:{tp_7:^3}|TN:{tn_7:^3}|FP:{FP_7:^3}|FN:{FN_7:^3}|accuaracy:{acc_7:^5}|F_1:{f_7:.3}')
    print(f'col  1,3,4 |TP:{tp_8:^3}|TN:{tn_8:^3}|FP:{FP_8:^3}|FN:{FN_8:^3}|accuaracy:{acc_8:^5}|F_1:{f_8:.3}')
    print(f'col  2,3,4 |TP:{tp_9:^3}|TN:{tn_9:^3}|FP:{FP_9:^3}|FN:{FN_9:^3}|accuaracy:{acc_9:^5}|F_1:{f_9:.3}')
    print(f'col 1,2,3,4|TP:{tp_10:^3}|TN:{tn_10:^3}|FP:{FP_10:^3}|FN:{FN_10:^3}|accuaracy:{acc_10:^5}|F_1:{f_10:.3}')
```

evaluation.py

In the main part of the script, we load the data and the weights we calculated in the previous script. We go again for each possible combination of features and calculate the true/false positives/negatives of the model. We also calculate accuracy and the F_1 score. For the combinations of 2 features, we also plot them with the decision boundary.

```
col  1 , 2 |TP:472|TN:476|FP:24 |FN:28 |accuaracy:0.948|F_1:0.948
col  1 , 3 |TP:435|TN:445|FP:55 |FN:65 |accuaracy:0.88 |F_1:0.879
col  1 , 4 |TP:434|TN:445|FP:55 |FN:66 |accuaracy:0.879|F_1:0.878
col  2 , 3 |TP:392|TN:390|FP:110|FN:108|accuaracy:0.782|F_1:0.782
col  2 , 4 |TP:393|TN:389|FP:111|FN:107|accuaracy:0.782|F_1:0.783
col  3 , 4 |TP:91 |TN:382|FP:118|FN:409|accuaracy:0.473|F_1:0.257
col  1,2,3 |TP:472|TN:476|FP:24 |FN:28 |accuaracy:0.948|F_1:0.948
col  1,2,4 |TP:472|TN:476|FP:24 |FN:28 |accuaracy:0.948|F_1:0.948
col  1,3,4 |TP:435|TN:445|FP:55 |FN:65 |accuaracy:0.88 |F_1:0.879
col  2,3,4 |TP:392|TN:390|FP:110|FN:108|accuaracy:0.782|F_1:0.782
col 1,2,3,4|TP:472|TN:476|FP:24 |FN:28 |accuaracy:0.948|F_1:0.948
```

output/evaluation.txt

As we can see our results confirm the ones we got in Figure 28. The combinations that include the first two columns give an accuracy and F_1 score of 0.948 meaning that out of 1000 cases, only 52 were mislabeled. Other combinations vary in performance. The worst performance comes for the combination of the 3rd and 4th columns which as we already mentioned doesn't include either the 1st or 2nd column and is unable to distinguish between the two having an accuracy of less than 0.5 and a very low F_1 score.

In Figures 29 to 34 we can find the plots of each combination of features with the decision boundary. We can see how great the boundary line manages to distinguish between the spiral and elliptical galaxies in Figure 29 where we have the first two features. The results are more mixed for the rest of them and for Figure 34 we can see that the boundary line is unable to distinguish the galaxies, as we already mentioned that columns 3 and 4 are not able to correctly classify the galaxies by themselves.

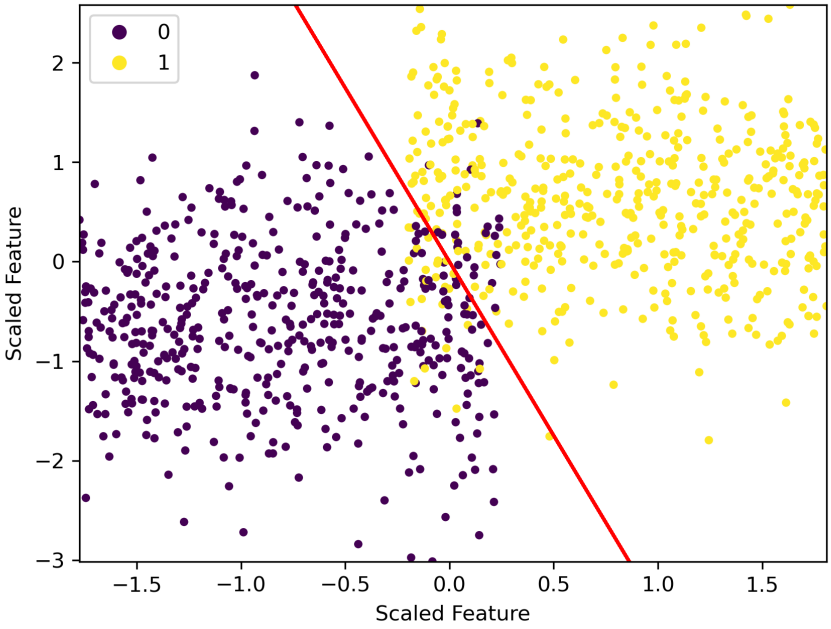

Figure 29: The 1st (x-axis) and 2nd (y-axis) rescaled features with the decision boundary. With different colors, we indicate the true labels of each point. This is the best-performing combination of features.

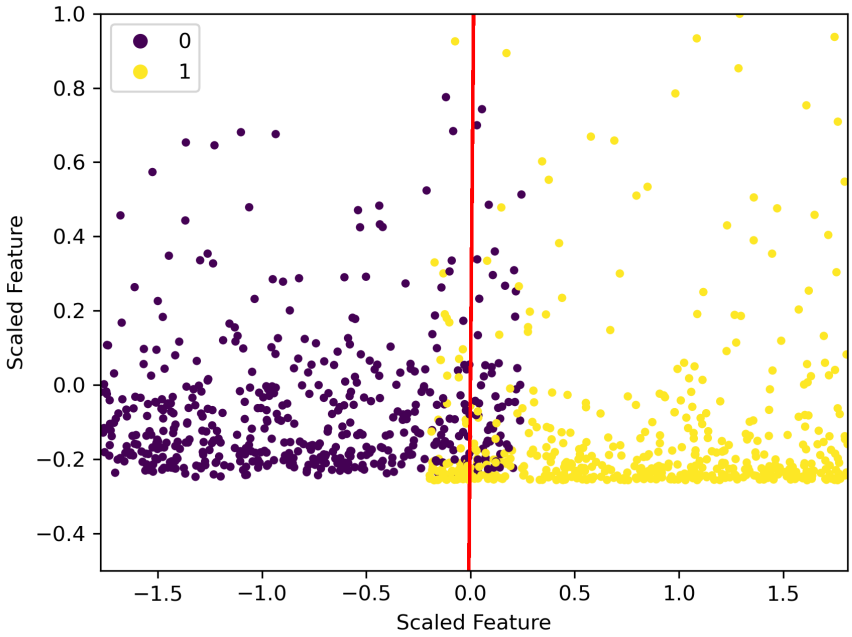

Figure 30: The 1st (x-axis) and 2nd (y-axis) rescaled features with the decision boundary. With different colors, we indicate the true labels of each point.

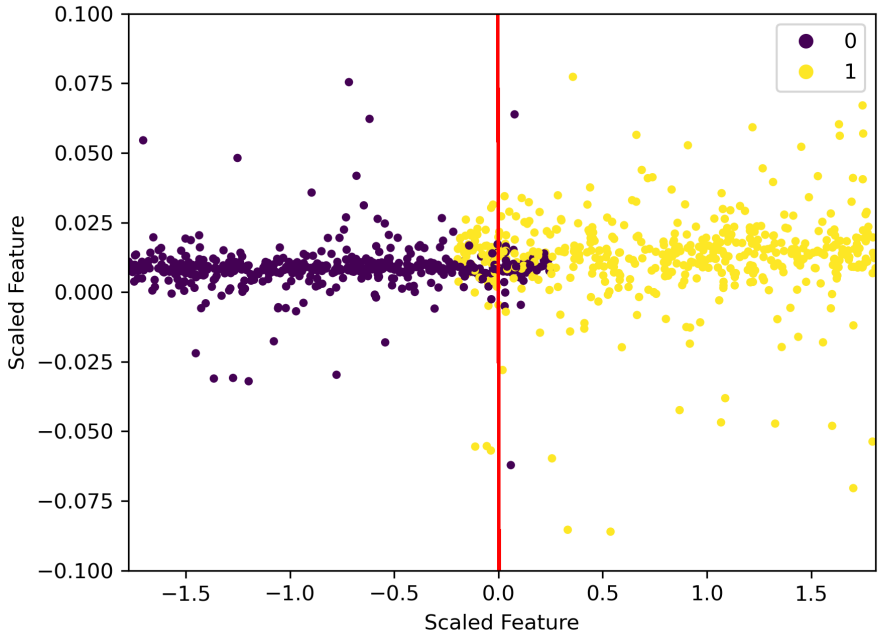

Figure 31: The 1st (x-axis) and 2nd (y-axis) rescaled features with the decision boundary. With different colors, we indicate the true labels of each point.

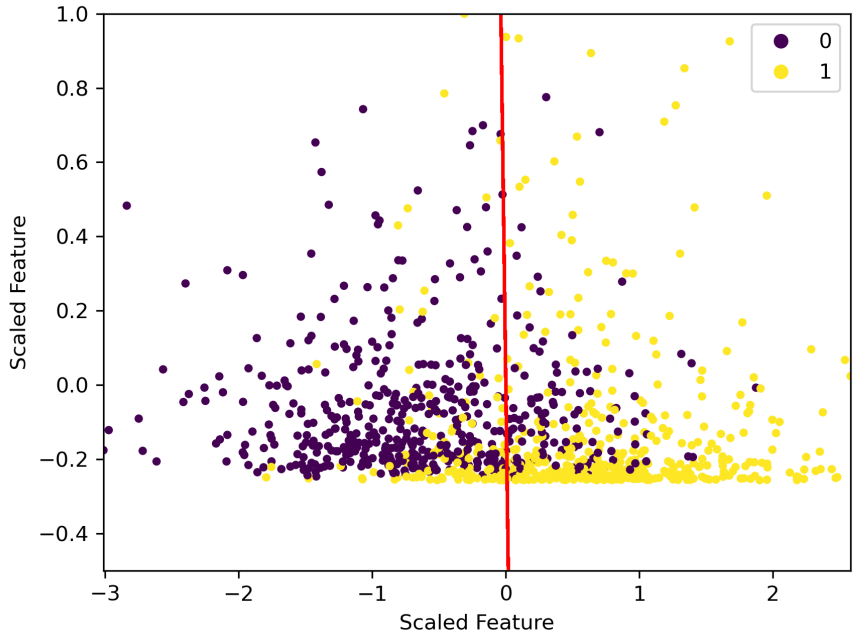

Figure 32: The 1st (x-axis) and 2nd (y-axis) rescaled features with the decision boundary. With different colors, we indicate the true labels of each point.

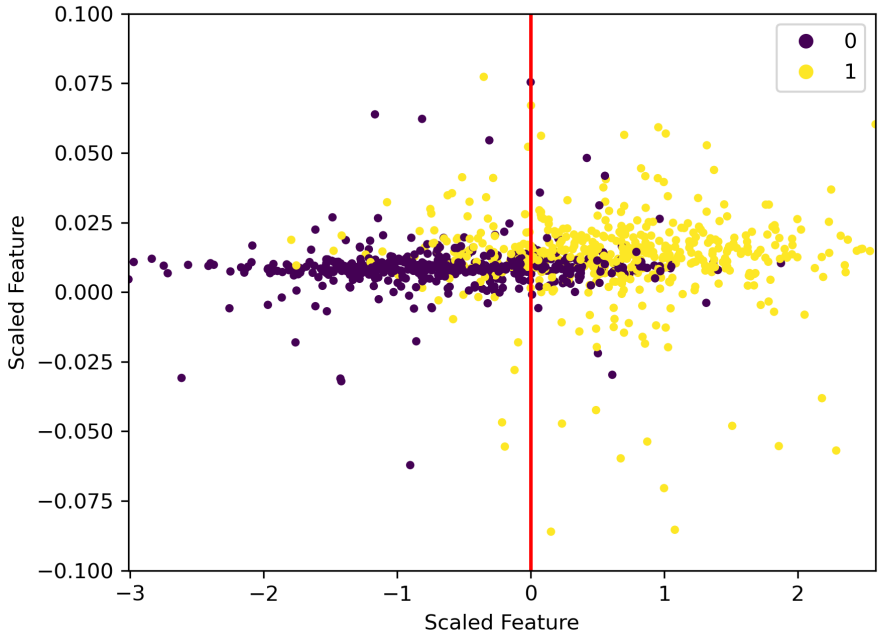

Figure 33: The 1st (x-axis) and 2nd (y-axis) rescaled features with the decision boundary. With different colors, we indicate the true labels of each point.

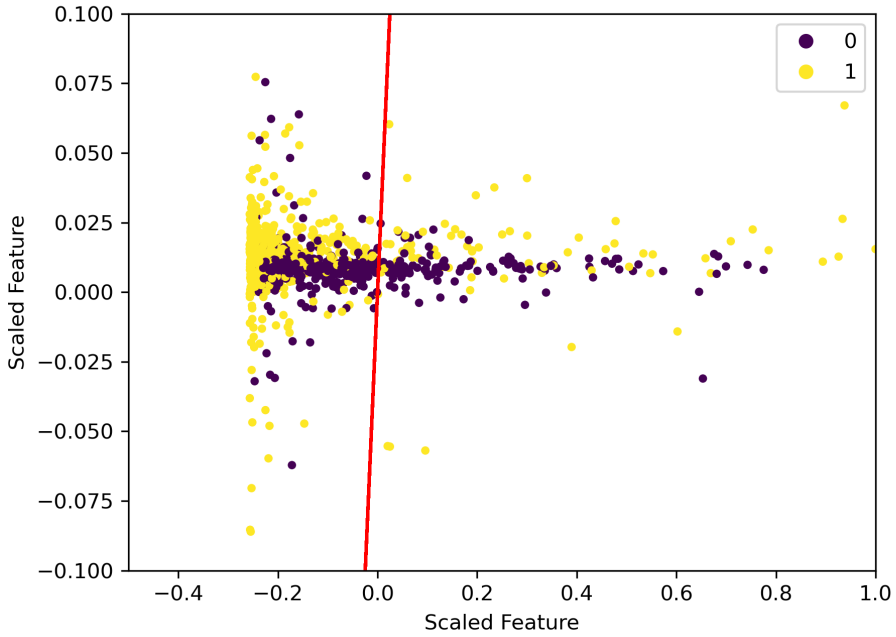

Figure 34: The 1st (x-axis) and 2nd (y-axis) rescaled features with the decision boundary. With different colors, we indicate the true labels of each point. This is the worst-performing combination of features.