Numerical Recipes: Hand-In 4

Rens Kievit (s1948415)

May 25, 2023

Abstract

In this report we present the problems, solutions and scripts for the exercises from the fourth handout for the course Numerical Recipes.

Plotting styles in this report are set using the following code

```
def set_styles():
    """For consistent plotting scheme"""
    plt.style.use('default')
    mpl.rcParams['axes.grid'] = True
    plt.style.use('seaborn-darkgrid')
    mpl.rcParams['font.family'] = 'serif'
    mpl.rcParams['lines.linewidth'] = 1.5
    mpl.rcParams['font.size'] = 14
```

Listing 1: Matplotlib Plotting Styles

1 Solar System Orbits

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.time import Time
from astropy.coordinates import solar_system_ephemeris
from astropy.coordinates import get_body_barycentric_posvel
from astropy import units as u
from astropy import constants as const
from plotting import set_styles

from algorithms import LeapFrog, RungeKutta4

# TODO update gravity
def grav_force(r, Rsun, Msun=const.M_sun.value,
               G=const.G.to_value((u.AU**3)*(u.kg**(-1))*(u.d**(-2)))):
    """Compute the N-D gravitational force acting on an object with mass m1
    from an object with mass m2 located at the origin"""
    r_diff = r - Rsun
    return - (G * Msun * r_diff) / (np.power(np.linalg.norm(r_diff), 3))

    #return (G * Msun / (r_diff*r_diff)) * (r_diff/np.sum(np.sqrt(r_diff*r_diff)))

def get_pos_vel(object_names,
                time="2021-12-07 10:00",
                plot=False):
    """Get the positions in AU and velocities in AU/d of all solar system objects at time."""
    t = Time(time)
    planets = {} # Dictionary in which we save our planet objects (and the sun)

    if plot:
        fig, [ax1, ax2] = plt.subplots(1,2,figsize=(12,6), tight_layout=True)

    for obj in object_names:
        # get the astropy data
        with solar_system_ephemeris.set('jpl'):
            obj_pos_vel = get_body_barycentric_posvel(obj, t)

        x0 = obj_pos_vel[0].xyz.to_value(u.AU)
        v0 = obj_pos_vel[1].xyz.to_value(u.AU/u.d)

        if obj == 'sun':
            r_sun = x0 # Position of the sun, which we take to remain cosntant

        # Feed r_sun into the grav_force function to get the correct coordinates
        acc_func = lambda x, v: grav_force(x, r_sun)
        planets[obj] = LeapFrog(x0, v0, acc_func)

        if plot:
            if obj == 'sun':
                m = '*'
            else:
                m = 'o'
```

```
52              ax1.scatter(x0[0], x0[1], marker=m)
53              ax2.scatter(x0[0], x0[2], label=obj.capitalize(),
    marker=m)
54
55      if plot:
56          ax1.set_xlabel(r'$X$ [AU]')
57          ax1.set_ylabel(r'$Y$ [AU]')
58          ax2.set_xlabel(r'$X$ [AU]')
59          ax2.set_ylabel(r'$Z$ [AU]')
60          plt.suptitle('Initial Star and Planet Positions')
61
62          plt.legend()
63          plt.savefig('results/initial_positions.png', bbox_inches='
    tight')
64
65      return planets
66
67  def make_orbits(planets, N, h, plot=False,
68                  compare_to_rk4=False):
69      """"""
70      if plot:
71          fig1, axs1 = plt.subplots(1,2,figsize=(15,6))
72          fig2, ax2 = plt.subplots(1,1,figsize=(12,6))
73          fig3, ax3 = plt.subplots(1,1,figsize=(12,6))
74          fig4, axs4 = plt.subplots(1,2,figsize=(15,6))
75      i = 1
76
77      for name, obj in planets.items():
78          if name == 'sun':
79              # Just place it in all corresponding plots
80              axs1[0].scatter(obj.x0[0], obj.x0[1], color='yellow', s
    =50, marker='*', label='Sun')
81              axs1[1].scatter(obj.x0[0], obj.x0[1], color='yellow', s
    =50, marker='*')
82              axs4[0].scatter(obj.x0[0], obj.x0[1], color='yellow', s
    =50, marker='*', label='Sun')
83              axs4[1].scatter(obj.x0[0], obj.x0[1], color='yellow', s
    =50, marker='*')
84              continue # do not simulate the sun to itself
85
86          print(name.capitalize())
87          # Apply Leapfrog
88          obj.simulate_motion(N, h)
89
90          if plot:
91              axs1[0].scatter(obj.x0[0], obj.x0[1], c=f'C{i}', label=
    name.capitalize())
92              axs1[1].scatter(obj.x0[0], obj.x0[1], c=f'C{i}')
93              axs1[0].plot(obj.x[:,0], obj.x[:,1], c=f'C{i}')
94              axs1[1].plot(obj.x[:,0], obj.x[:,1], c=f'C{i}')
95              ax2.plot(h*np.arange(obj.x.shape[0]), obj.x[:,2], c=f'C
    {i}')
96
97
98          # Code for the bonus question
99          if compare_to_rk4:
100             RK4 = RungeKutta4(obj.x0, obj.v0, obj.acc_func)
101             RK4.simulate_motion(N, h)
102
103             # Plot the difference in x-positions between RK4 and LF
104             ax3.plot(h*np.arange(obj.x.shape[0]), (RK4.x[:,0] - obj
    .x[:,0]), c=f'C{i}')
```

```
105
106             # Plot the orbits
107             axs4[0].scatter(RK4.x0[0], RK4.x0[1], c=f'C{i}', label=
    name.capitalize())
108             axs4[1].scatter(RK4.x0[0], RK4.x0[1], c=f'C{i}')
109             axs4[0].plot(RK4.x[:,0], RK4.x[:,1], c=f'C{i}')
110             axs4[1].plot(RK4.x[:,0], RK4.x[:,1], c=f'C{i}')
111
112         i += 1
113
114     if plot:
115         # FIGURE 1
116         for ax1 in axs1:
117             ax1.set_xlabel(r'$X$ [AU]')
118             ax1.set_ylabel(r'$Y$ [AU]')
119         fig1.suptitle('Star and Planet Orbits over 200 Years with
    LeapFrog')
120
121         # Right plot zoomed in on the rocky planets
122         axs1[1].set_xlim(-2, 2)
123         axs1[1].set_ylim(-2, 2)
124
125         # Place legend next to figures
126         fig1.subplots_adjust(right=0.85)
127         fig1.legend(loc='right')
128         fig1.savefig('results/orbits_lf.png', bbox_inches='tight')
129
130         # FIGURE 2
131         ax2.set_xlabel(r'Time in Days')
132         ax2.set_ylabel(r'$Z$ [AU]')
133         ax2.set_title('Planet z-positions over 200 Years with
    LeapFrog')
134         fig2.savefig('results/zplane.png', bbox_inches='tight')
135
136         # FIGURE 3
137         ax3.set_xlabel(r'Time in Days')
138         ax3.set_ylabel(r'$X_{\mathrm{RK4}} - X_{\mathrm{LF}}$ [AU]'
    )
139         #ax3.set_yscale('log')
140         ax3.set_title('Positional Differences')
141         fig3.savefig('results/rk4_lf_diff.png', bbox_inches='tight'
    )
142
143         # FIGURE 4
144         for ax4 in axs4:
145             ax4.set_xlabel(r'$X$ [AU]')
146             ax4.set_ylabel(r'$Y$ [AU]')
147         fig4.suptitle('Star and Planet Orbits over 200 Years with
    Runge-Kutta 4')
148
149         # Right plot zoomed in on the rocky planets
150         axs4[1].set_xlim(-2, 2)
151         axs4[1].set_ylim(-2, 2)
152
153         # Place legend next to figures
154         fig4.subplots_adjust(right=0.85)
155         fig4.legend(loc='right')
156         fig4.savefig('results/orbits_rk4.png', bbox_inches='tight')
157
158 def solar_system_sim():
159     set_styles()
```

```
    object_names = ['sun', 'mercury', 'venus', 'earth', 'mars', '
    jupiter', 'saturn', 'uranus', 'neptune']

    h = 0.5
    N = int((365.25 * 200)/h) # 200yrs in total
    print('N Steps = ', N)

    plot = True
    compare_to_rk4 = True
    ##################3

    planets = get_pos_vel(object_names, plot=plot)
    make_orbits(planets, N, h, plot=plot, compare_to_rk4=
    compare_to_rk4)

def main():
    solar_system_sim()

if __name__ == '__main__':
    main()
```

Listing 2: Code to collect the positions and velocities of solar system objects and apply the LeapFrog and Runge-Kutta 4 Algorithms.

In this section we will simulate the orbits of the planets in our Solar system. Here the forces between planets are negligible compared to the force between each planet and the sun. The orbit of each planet is therefore well defined by a single equation, which we can easily brute force with an ODE solver. The acceleration of the planet is given by the gravitational force, which is defined as

$$\vec{F_G} = \frac{-mMG}{||\vec{r}||^3}\vec{r}. \tag{1}$$

Where $M = 1.989 \times 10^{30}$ kg is the mass of the sun, m is the mass of the planet, $G = 6.6743 \times 10^{-11}$ m^3 kg^{-1} s^{-2} is the gravitational constant. $\vec{r}$ is the vector pointing from is the planet to the sun. The acceleration is then simply given through $F = ma$, so the planet mass cancels out.

We get the initial positions and velocities of all 8 solar system planets and the sun using `astropy` following the procedure given on the handin at 10:00 on 2021-12-07. We show these initial positions in Figure 1.

1.1 Leapfrog

```
class LeapFrog():
    """Class to store positions and velocities for any physical
    object
    whose motion we want to simulate using leapfrog"""
    def __init__(self, x0, v0, acc_func):
        self.x0 = x0
        self.v0 = v0
        self.acc_func = acc_func
        self.h = None # Only works for constant h at this moment
        self.dim = x0.shape[0]

    def _setup_sim(self, N, h, continue_from_last=False):
        """Simulate the motion under accelatrion func for N
    timesteps"""
        if (self.h is not None) and (self.h != h):
```

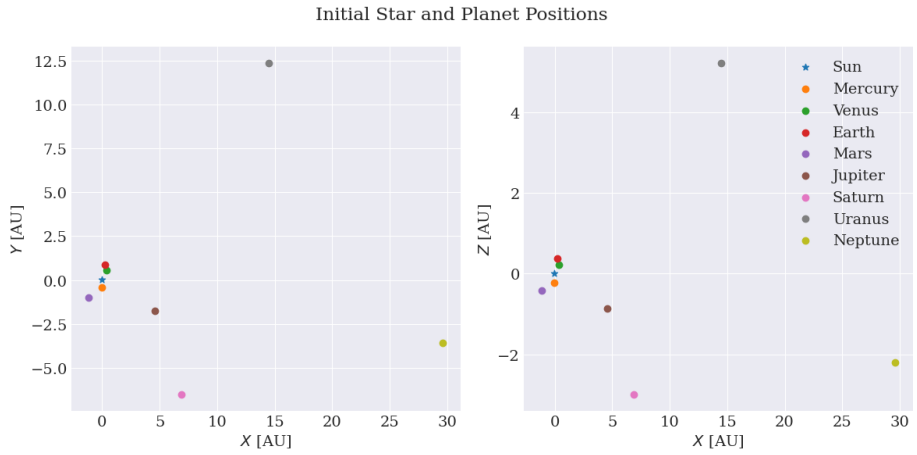

Figure 1: Initial positions of all 8 solar system planets and the sun at 10:00 on 2021-12-07 in the x-y plane (*left*) and the x-z plane (*right*)

.

```
            raise ValueError(f'Provided h {h} does not match existing h {self.h}')
        self.h = h
        if continue_from_last:
            self.index = self.x.shape[0]
            self.x = np.vstack([self.x, np.zeros([N, self.dim])])
            self.v = np.vstack([self.v, np.zeros([2*N, self.dim])])
            self.a = np.vstack([self.a. np.zeros([N, self.dim])])
        else:
            self.x = np.zeros([N, self.dim])
            self.v = np.zeros([2*N, self.dim]) # Two times so we can store v_i and v_i+0.5
            self.a = np.zeros([N, self.dim])

            self.x[0] = self.x0
            self.v[0] = self.v0
            self.v[1] = self._kickstart() # Get v at v_i+1/2 using RK4
            self.index = 1

    def _kickstart(self):
        """Given x0 and v0 and a function for acceleration, get the velocity
        at v_i+1/2. This is done standardized for RK4, but Euler also works"""
        h_rk4 = 0.5*self.h # take a half step
        # Approximated positions after 0.25 and 0.5 of a step using v0
        x_half = self.x[0] + h_rk4 * self.v[0]
        x_full = self.x[0] + h_rk4 * self.v[0]

        # Find v_1/2 using RK4 and approximated positions
        k1 = h_rk4 * self.acc_func(self.x[0], self.v[0])
        k2 = h_rk4 * self.acc_func(x_half, self.v[0] + 0.5*k1)
        k3 = h_rk4 * self.acc_func(x_half, self.v[0] + 0.5*k2)
        k4 = h_rk4 * self.acc_func(x_full, self.v[0] + k3)

```

```
45          one_sixth = 1./6.
46          one_third = 1./3.
47          return self.v[0] + one_sixth*k1 + one_third*k2 + one_third*
        k3 + one_sixth*k4
48
49      def simulate_motion(self, N, h, continue_from_last=False):
50          """Simulate motion under the given acceleration fucntion"""
51          self._setup_sim(N, h, continue_from_last)
52
53          # Leapfrog algorithm
54          for i in range(self.index, self.index + N - 1):
55              # index in the velocity array. Offset because we leave
        space for v_i
56              v_idx = (2*i) + 1
57              # currently can't use a = a(x, v) because we don't have
         v_i
58              self.a[i] = self.acc_func(self.x[i-1], self.v[i-2])
59              self.v[v_idx] = self.v[v_idx-2] + self.h * self.a[i]
60              self.x[i] = self.x[i-1] + self.h * self.v[v_idx]
```

Listing 3: LeapFrog Class

We start by applying the leapfrog algorithm to simulate the orbits. This is the proper method of solving ordinary differential equations for phsyical systems, because it is reversible and therefore able to conserve energy. This is an important factor in making sure the solutions do not diverge to unphysical states. We have implemented the algorithm in a class, given above. The leapfrog algorithm works by kicking the object from positions x_i with velocity $v_{i+1/2}$. However, we are only given x_0 and v_0. This means that we first have to use a regular ODE solver to find $v_{1/2}$. Here, this is done in the `_kickstart` function with a single iteration of Runge-Kutta 4.

We integrate the orbits with a step size $h = 0.5$ days for 200 years, this gives $N = 146100$ steps in total. We show the resulting orbits in Figure 2.

The orbits appear to behave very nicely. As mentioned previously, this algorithm conserves energy. This is visible in the fact that the orbits in the top plot of Figure 2 appear to be well defined lines that do not diverge away from or towards the Sun, which would be a violation of energy conservation. This same fact is visible in the bottom plot, the planets all appear to oscilate up and down in the z-plane, but the amplitude of the oscilations is constant. This also indicates the fact that energy conservation is not violated. Finally, this simulation shows that the orbit of Mercury is not entirely constant, but rather shifts around a little bit. This is not per se an inconsistent result, because its orbit remains closed.

1.2 Runge-Kutta 4

```
1  class RungeKutta4():
2      """Class to store positions and velocities for any physical
       object
3      whose motion we want to simulate using leapfrog"""
4      def __init__(self, x0, v0, acc_func):
5          self.x0 = x0
6          self.v0 = v0
7          self.acc_func = acc_func
8          self.h = None # Only works for constant h at this moment
9          self.dim = x0.shape[0]
```

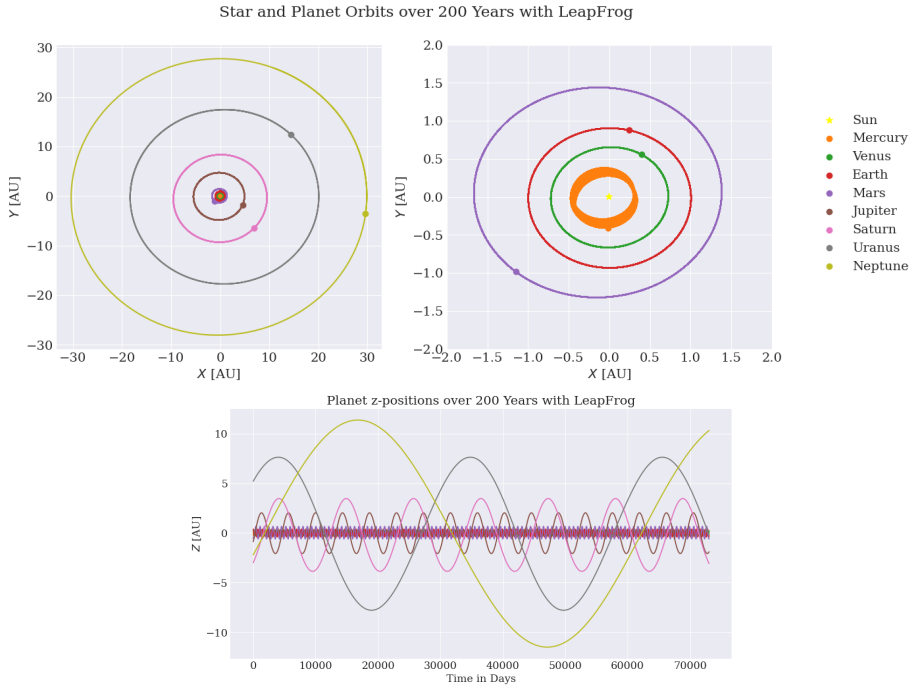

Figure 2: Integrated orbits of the 8 planets in the solar system over 200 years with a step size of 0.5 days using the LeapFrog algorithm and an initial kick with Runge-Kutta 4. *top*: Planetary orbits in the x-y plane, the plot in the right is zoomed in to better highlight the orbits of the rocky planets. *bottom*: Planetary positions on the z-axis as a function of time.

```
10
11      def _setup_sim(self, N, h, continue_from_last=False):
12          """Simulate the motion under accelatrion func for N
    timesteps"""
13          if (self.h is not None) and (self.h != h):
14              raise ValueError(f'Provided h {h} does not match
    existing h {self.h}')
15          self.h = h
16          if continue_from_last:
17              self.index = self.x.shape[0]
18              self.x = np.vstack([self.x, np.zeros([N, self.dim])])
19              self.v = np.vstack([self.v, np.zeros([N, self.dim])])
20              self.a = np.vstack([self.a. np.zeros([N, self.dim])])
21          else:
22              self.x = np.zeros([N, self.dim])
23              self.v = np.zeros([N, self.dim])
24              self.a = np.zeros([N, self.dim])
25
26              self.x[0] = self.x0
27              self.v[0] = self.v0
28              self.index = 0
29
30      def simulate_motion(self, N, h, continue_from_last=False):
31          """Simulate motion under the given acceleration fucntion"""
32          self._setup_sim(N, h, continue_from_last)
33
34          one_third = 1./3.
35          one_sixth = 1./6.
36
37          # Leapfrog algorithm
38          for i in range(self.index, self.index + N - 1):
39              k1v = self.h * self.acc_func(self.x[i], self.v[i])
40              k1x = self.h * self.v[i]
41
42              k2v = self.h * self.acc_func(self.x[i]+0.5*k1x, self.v[
    i]+0.5*k1v)
43              k2x = self.h * (self.v[i] + 0.5*k1v)
44
45              k3v = self.h * self.acc_func(self.x[i]+0.5*k2x, self.v[
    i]+0.5*k2v)
46              k3x = self.h * (self.v[i] + 0.5*k2v)
47
48              k4v = self.h * self.acc_func(self.x[i]+k3x, self.v[i]+
    k3v)
49              k4x = self.h * (self.v[i] + k3v)
50
51              self.v[i+1] = self.v[i] + one_sixth*k1v + one_third*k2v
     + one_third*k3v + one_sixth*k4v
52              self.x[i+1] = self.x[i] + one_sixth*k1x + one_third*k2x
     + one_third*k3x + one_sixth*k4x
```

Listing 4: Runge-Kutta 4 Class

As an experiment, we also integrate the orbits of the planets given the same initial conditions with a normal ODE solver that does not have the power to preserve energy. Similar to the LeapFrog method above, we also implement the RK-4 method using a class given above. However, this time we do not have to provide the velocities with an initial kick. We use the same step size and number of steps as before, and plot the resulting orbits in Figure 3

Visually, the orbits of all planets except Mercury appear exactly the same. The biggest difference lies in the fact that Mercury's orbit remains constant

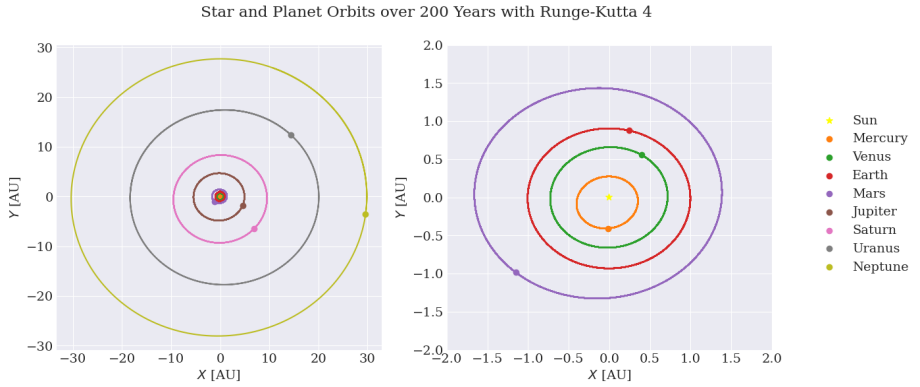

Figure 3: Integrated orbits in the x-y plane of the 8 planets in the solar system over 200 years with a step size of 0.5 days using the Runge-Kutta 4 method.

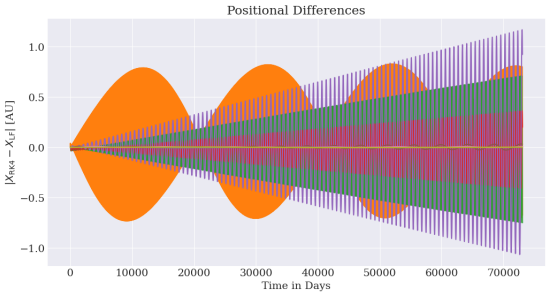

Figure 4: Difference in x-position between the orbit as integrated using the Runge-Kutta 4 method and the LeapFrog algorithm plotted as a function of time

over 200 years according to RK4, while it shifted around according to LeapFrog. However comparing two plots with each other visually is not the best method of investigating the performance of an algorithm. Therefore we present the difference in x-position between the two integrated orbits in Figure 4.

In that figure we see that the difference in x-position shows a somewhat periodic pattern that increases in amplitude over time. This indicates that the two solutions are slowly diverging from each other. The pattern we see corresponds to what we might have expected. Because RK-4 is a forward method, the velocity it predicts will always push the object slightly more away from the star than the true value. This small error accumulates over time leading to a larger orbit. This can physically be interpreted as increasing the amount of energy in the system, which is a violation of the laws of physics that LeapFrog was able to adhere to. Additionaly, we can clearly see that there is a large discrepancy between the two orbits of Mercury that occurs from a very early point.

2 Computing Forces with FFT

```
#!/usr/bin/env python3
def get_densities():
    """Code downloaded from Zorry Belcheva's personal strw page.
    Turned into a function for use in fourier.py"""
    import numpy as np
    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
```

Listing 5: Code for the Cloud-In-Cell method to distribute densities on the grid.

```
import numpy as np
import matplotlib.pyplot as plt
from plotting import set_styles
from cic import get_densities
from algorithms import fft_nd

def plot_at_zslices(data, savename, cb_label, suptitle,
                    z_slices=[4.5, 9.5, 11.5, 14.5]):

    fig, axs = plt.subplots(2,2,figsize=(10,8),sharex=True,sharey=
    True)

    for ax, z in zip(axs.flatten(), z_slices):
```

```
        # Integer z appear at the edges, but in our array we have the centers
        # so z=4.5 occurs at index 4
        im = ax.imshow(data[:,:,int(z)], cmap='jet', origin='lower')
        ax.set_title(rf'$Z$ = {z}')
        ax.grid(False)
    # Setup a colorbar for all
    # https://stackoverflow.com/questions/13784201/how-to-have-one-colorbar-for-all-subplots
    fig.subplots_adjust(right=0.8)
    cbar_ax = fig.add_axes([0.85, 0.11, 0.05, 0.77])
    cb = fig.colorbar(im, cax=cbar_ax)

    cb.set_label(cb_label)
    for ax in axs[:,0]:
        ax.set_ylabel(r'$Y$')
    for ax in axs[-1,:]:
        ax.set_xlabel(r'$X$')
    fig.suptitle(suptitle)
    plt.savefig(f'results/{savename}.png', bbox_inches='tight')

def compute_forces(eps=1e-15):
    # Make density grid and compute the density contrasts
    mean_rho = 1024/(16**3)
    rho = get_densities()
    delta = (rho - mean_rho)/mean_rho # Density contrasts

    # Plot slices at various z
    plot_at_zslices(delta, savename='density_contrast_slices', cb_label='Density Contrast', suptitle='Grid Density Contrast')

    # Apply FFT to \delta to get k^2 \Phi~
    phi_fft = fft_nd(delta)

    # Compute all k = sqrt(k_x^2 + k_y^2 + k_z^2)
    # The below only works for data with equal sized vertices in 3D
    k_vals_sq = (np.arange(phi_fft.shape[0], dtype=np.float64))** 2
    # Using this notation we force each array along a distinct axis resulting in a cube
    k_cube_sq = k_vals_sq[None, None, :] + k_vals_sq[None, :, None] + k_vals_sq[:, None, None]
    k_cube_sq[k_cube_sq < eps] = eps # Fight divide by zero error by setting zero to ~eps_m

    potential = fft_nd(phi_fft/k_cube_sq, inverse=True)

    # Plot the log of the FFT potential
    plot_at_zslices(np.log10(np.abs(phi_fft)), savename='fft_potential', cb_label=r'$\log_{10}(|\tilde{\Phi}|)$', suptitle='FFT-Space Potential')
    # Plot the potential
    plot_at_zslices(np.real(potential), savename='potential_slices', cb_label='Potential', suptitle='Grid Potential')

def main():
    set_styles()
    compute_forces()

if __name__ == '__main__':
    main()
```

Listing 6: All code to apply the procedures outlined below to compute the

potential.

```
# FFT
def dft_recursive(x, inverse):
    """Function to be called recursively by the FFT algorithm to perform the DFT on
    subsets of the array following the Danielson-Lanczos lemma. For speed we make use
    of trigonometric recurrence, therefore we never have to compute a complex exponent."""
    N = len(x)
    if N > 2:
        even = dft_recursive(x[::2], inverse)
        odd = dft_recursive(x[1::2], inverse)
        x = np.append(even, odd)

    # If we want an iFFT, a -1 should appear in the exponent
    if inverse:
        inv_fac = -1.
    else:
        inv_fac = 1.

    # Define the trig. recurrence variables
    theta = 2.*np.pi/N
    alpha = 2.*(np.sin(theta/2)**2)
    beta = np.sin(theta)
    cos_k = 1. # We start with k = 0; cos(0) = 1
    sin_k = 0. #                      sin(0) = 1

    for k in range(0, N//2):
        k2 = k + N//2 # Index of the 'odd' number
        t = x[k]

        Wnk = cos_k + inv_fac*1j*sin_k #np.exp(inv_fac*2.j*np.pi*k/N)
        second_factor = Wnk * x[k2]

        # one step of the fourier transform
        x[k] = t + second_factor
        x[k2] = t - second_factor

        # Update trig.
        cos_k_new = cos_k - alpha * cos_k - beta*sin_k
        sin_k_new = sin_k - alpha * sin_k + beta*cos_k
        cos_k, sin_k = cos_k_new, sin_k_new

    return x

def fft(x, inverse=False):
    """Apply the FFT algorithm to samples x using the recursive Cooley-Tukey algorithm
    If the length of x is not a power of 2, zeros are appended up to the closest higher
    power of 2. This function returns a complex array.
    If inverse is set to True, a '-' sign is introduced in the exponent of W_N^k"""
        # Check the dimensionality of the incoming data
    if len(x.shape) > 1:
        return fft_nd(x, inverse)

    # Check if N is a power of 2
    N = len(x)
```

```
    if (np.log2(N)%1) > 0: # Check if it is not an integer
        diff = int(2**(np.ceil(np.log2(N))) - N) # amount of zeros
    to add to make N a power of 2
        x = np.append(x, np.zeros(diff))
        N = len(x)

    # Cast x into a complex array so we can store
    x = np.array(x, dtype=np.cdouble)
    x_fft = dft_recursive(x, inverse)

    if inverse:
        x_fft /= N

    return x_fft

def fft_nd(x, inverse=False):
    """Apply the Fourier transform to mulitdimensional data. This
    can easily be done by performing
    the FFT algorithm along each axis separately consecutively"""
    dim = len(x.shape)
    func = lambda x: fft(x, inverse)
    # Start with dim 0 and work up to the highest dimension
    for i in range(dim):
        x = np.apply_along_axis(func1d=func, axis=i, arr=x)
    return x
```

Listing 7: Implementation of the recursive Cooley-Tukey algorithm with trigonometric recurrence.

In the previous section we could brute force the computation of the motion of the planets relatively easy because there are only eight objects, and eight relevant forces. However if interparticle interactions were important this task would have already been a lot more difficult as there would have been 56 forces to be taken into account at each timestep. Expanding beyond $N = 8$ rapidly worsens this problem. In this section we will investigate one of the methods to transform such a problem from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log(N))$, namely the Fast Fourier Transform (FFT). We will test this by computing the gravitational potential on a grid for a volume of 16x16x16 units with 1024 particles, each with a mass of 1. The edges of the grid are connected such taht $x = 16 \equiv 0$. The mass of each particle is assigned to a grid point using the Cloud-In-Cell method written by Zorry Belcheva, given above. We start by filling out the grid, and present the density contrast $\delta = (\rho - \overline{\rho})/\overline{\rho}$ at 4 separate 2D slices at constant z Figure

To calculate the gravitational force, we first need the gravitational potential which we can compute using the Poisson equation

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

We can simplify this because we only need to solve the spatial dependence of this equation, i.e. $\nabla^2\Phi \propto \delta$. Using fourier transforms we can easily do this with by computing the foruier transform of the density contrast, using that to compute the transformed potential, and then applying an inverse fourier transformation to find the true potential. Mathematically, we can write this as

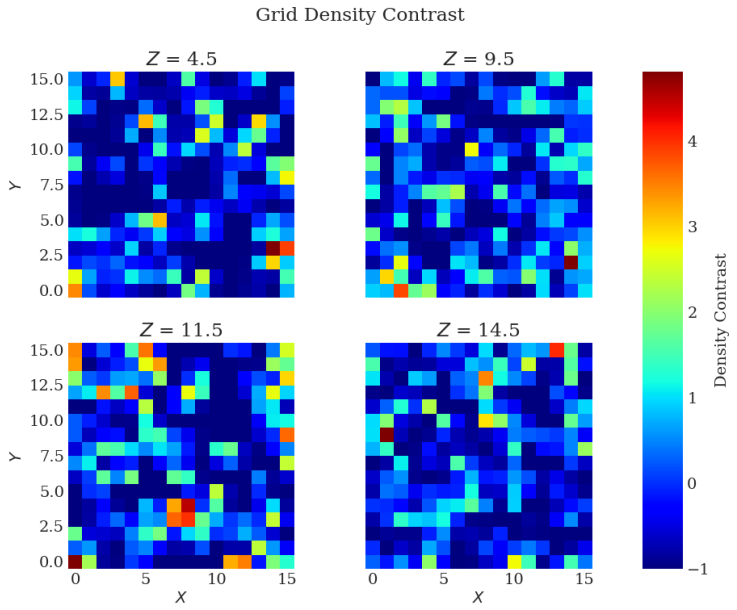

Figure 5: Density contrast δ in four separate 2D slices at constant z computed using the Cloud-In-Cell method on a 3D grid.

$$\delta \xrightarrow{\text{FFT}} \tilde{\delta} \propto k^2 \tilde{\Phi}$$

$$\tilde{\Phi} \propto \frac{\tilde{\delta}}{k^2}$$

$$\frac{\tilde{\delta}}{k^2} \xrightarrow{\text{IFFT}} \Phi$$

The only question that now remains is what k is. We can calculate this from the wavenumber vector as

$$k^2 = k_x^2 + k_y^2 + k_z^2. \tag{3}$$

Where k_x, k_y, and k_z are just equal to the grid coordinates and therefore take on integer values from 0 to 15.

The FFT algorithm in this work is implemented using the recursive Cooley-Tukey algorithm with trigonometric recurrence for additional speed up.

2.1 Results

We start with the first two steps from our potential calcuation procedure, i.e. apply a fourier transform to the density contrast, and divide it by k^2. In Figure 6 we present the log of the absolute value of $\tilde{\Phi}$. We take the absolute value because there is a complex component which we want to include in the figure.

We then apply the inverse Fourier Transform with the FFT algorithm to the data shown in Figure 6 to find the gravitational potential which we present in Figure 7.

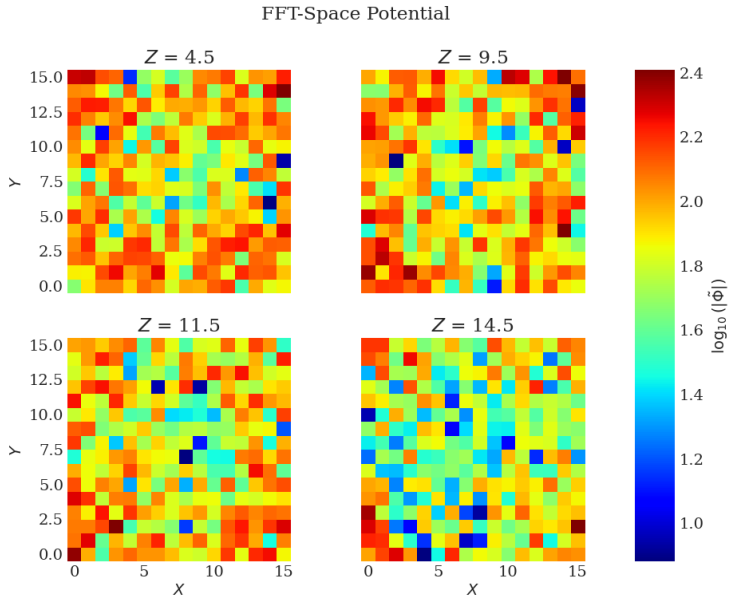

Figure 6: Logarithm of the absolute value of the fourier transformed gravitational potential, this is equal to the fourier transformed density contrast divided by k^2.

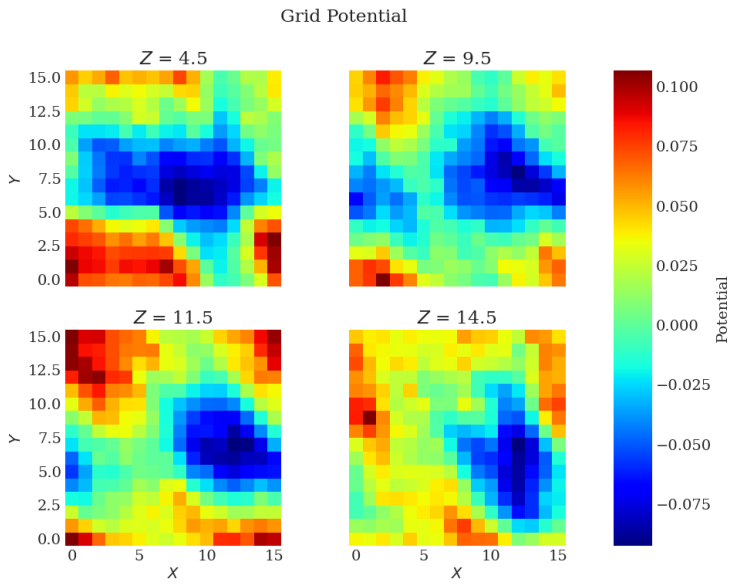

Figure 7: Gravitational potential of the density contrast grid shown in Fiugre 5, computed using fourier transformations as described in text.

We can see that the calculated potentials are quite smooth when compared to the more eratic distritubution of mass in Figure 5. However this makes sense as we can see from the Poisson equation that the density contrast influences only the second order derivative of the potential. Comparing the density contrast and potential figures, we can see that the regions of high and low density correspond well to the regions of high and low potential respectively, which ofcourse is exactly as expected for the gravitational potential. For example the low density region in the center of the $Z = 4.5$ slice corresponds to a potential well, and the high density 'square' in the $Z = 11.5$ slice at $Y = 3$ corresponds to a high potential arm reaching towards the center.

As a final note, a test run of the procedure outlined above with $k = 1$ resulted in a 'potential' that was exactly equal to the density contrast. While not physically relevant, this is a proof that the FFT algorithms properly worked because the inverse fourier transform of a fourier transformed dataset should be equal to the original data set.

3 Galaxy Classification

In this section we will attempt to implement a logistic regression algorithm to classify galaxies into two classes, ellipticals and spirals, using four parameters:

- κ_{co}, which indicates how much a galaxy is dominated by ordered rotation
- A color estimate (higher is redder, and therefore contains older stars)
- A measure of how extended the galaxy is
- Flux of an emission line tracing the star formation rate

We start by preprocessing all of the features by scaling them to have zero mean and unit standard deviation. This is implemented for each feature f_i separately as

$$f_{i,\text{scaled}} = \frac{f_i - \mu_i}{\sigma_i} \tag{4}$$

To get a sense of how each of our features is distributed, we plot their histograms in Figure 8, and present the first ten samples below.

```
1  -1.581862133006098681e+00  -6.612307828919693729e-03
       -3.475533387671284058e-02  4.149921715612928282e-03
2  1.574559067587914640e+00  -7.944354060046822097e-01
       2.031450234565139734e-01  1.605487869858183633e-02
3  1.531990475165863508e+00  8.573405014974248006e-01
       -2.079754178048100477e-01  4.166315770216809378e-02
4  1.297215471156623057e+00  1.226267345173265078e+00
       1.858865005815529270e-01  6.825469423304870997e-04
5  5.181259591131069930e-01  8.257714425578099871e-01
       -2.002589858418064583e-01  1.555766189074690373e-02
6  -1.647524013432828394e+00  -2.909640578232345343e-01
       -8.488083597139754743e-02  1.855013775209162245e-03
7  -1.312495960360358760e+00  -9.756536637063890627e-01
       6.454576379671385367e-02  6.772650820885613675e-03
8  9.418303994505743126e-02  -9.715157240517435788e-01
       -1.525183484703764025e-01  7.745013475795247196e-03
9  -7.308293662245395339e-01  -1.520148623547938893e+00
       -1.654403149798442940e-01  1.175440493844448313e-02
10 -1.198035514238125154e+00  -7.748995242729127542e-01
       -1.553888673156535449e-01  9.596892572192124507e-03
```

Listing 8: Scaled feature values for the first ten entries of the galaxy dataset

```
1  def hist(x, binmin, binmax, nbins, log=False, return_centers=False)
       :
2      """"""
3      if log:
4          bin_edges = np.logspace(np.log10(binmin), np.log10(binmax),
        nbins + 1)
5      else:
6          bin_edges = np.linspace(binmin, binmax, nbins + 1)
7      if return_centers:
8          bin_centers = np.zeros(nbins)
9
10     histogram = np.zeros(nbins)
11     for i in range(nbins):
12         bin_mask = (x >= bin_edges[i]) * (x < bin_edges[i + 1])
```

```
        if log:
            histogram[i] = len(x[bin_mask])
        else:
            histogram[i] = len(x[bin_mask])
        if return_centers:
            bin_centers[i] = bin_edges[i] + 0.5 * (bin_edges[i+1] -
 bin_edges[i])

    if return_centers:
        return histogram, bin_edges, bin_centers
    return histogram, bin_edges
```

Listing 9: Code to make the histograms in Figure 8

```
def preprocess_data(fname, plot=False, nbins=None):
    data = np.genfromtxt(fname)
    features = data[:,:-1]
    labels = data[:, -1]

    # Rescale the features
    for i in range(features.shape[1]):
        mean = np.mean(features[:,i])
        std = np.std(features[:,i])
        features[:,i] = (features[:,i] - mean)/std

    # Save to a text file
    np.savetxt('results/scaled_features.txt', features)

    if plot:
        fig, axs = plt.subplots(2,2,figsize=(8,8),sharex=False,
sharey=False,tight_layout=True)
        for i in range(features.shape[1]):
            ax = axs.flatten()[i]
            binmin = np.min(features[:,i])
            binmax = np.max(features[:,i])
            n, bin_edges, bin_centers = hist(features[:,i], binmin,
 binmax, nbins, return_centers=True)

            ax.step(bin_centers, n, where='mid', lw=3)
            ax.set_title(f'Feature {i}')
            ax.set_xlabel('Rescaled Values')

        for ax in axs[:,0]:
            ax.set_ylabel('Counts')
        plt.savefig('results/scaled_features_dist.png')

    return features, labels
```

Listing 10: Code for the feature preprocessing and plotting.

It is immediately noticable that while most values for features 2 and 3 lie around 0, there are outliers at extremely high values. This is less true for feature 0 which shows a peak at 0 and has a relatively constant distribution in its wings. Feature 1 appears to be nicely distributed close to a Gaussian. Due to the few outliers in features 2 and 3 we will limit ourselves to the 2% and 98% percentiles for plotting in the rest of this work to better highlight the variations in the regions where the bulk of the datapoints reside. Ofcourse we do not discard these points for the training process.

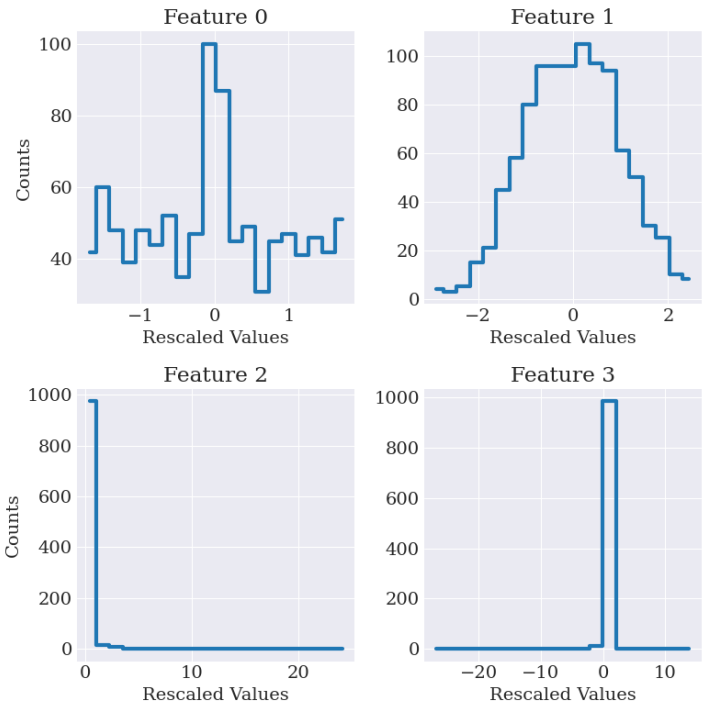

Figure 8: Distributions of the scaled features used for the classification algorithms in this section.

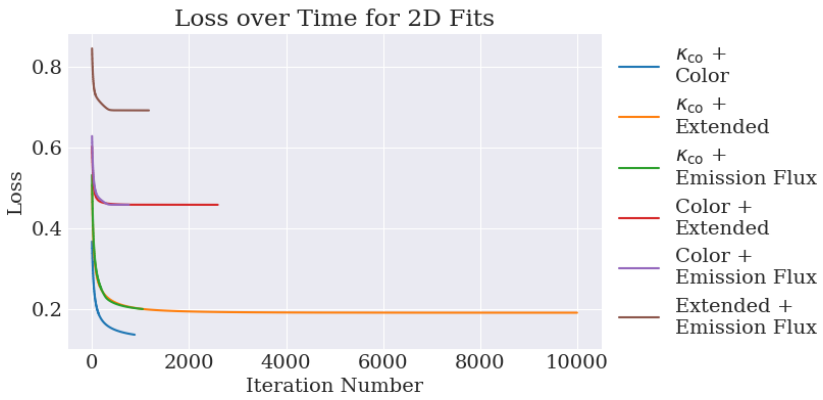

Figure 9: Loss value as a function of iteration number for each of the six possible combinations of two out of four features using logistic regression and a constant step size as minimization method.

3.1 Two Feature Combinations

We begin by investigating how well the logistic regression algorithm is able to predict the galaxy class if it is only provided with two of the four available features. We do this by running the algorithm by each of the six possible combinations of two features and investigating the resulting loss values. In each case we also include a bias feature consisting of only ones. As an extra test we also look at the differences between using a constant step size $\eta = 0.1$ and using line minimization using golden section search to discover the minimum. We show the resulting loss curves, the evolution of the loss value as a function of iteration, in Figure 9 for the learning rate method and in Figure 10 for the line mininimization method.

We can see that under both minimization methods, all loss values converge to approximately the same values. However line minimization converges $\sim 100-1000$ times faster than using a learning rate of 0.1. From these plots we can gather that the combination of κ_{CO} and color estimate is the best combination of parameters to estimate galaxy class. Meanwhile, the extended measure and the flux of the emission line tracing star formation rate appear to be a very bad combination of predictors. To get a better grasp of the distribution of the features, and the prediction capablity of our models, we plot all six of the two dimensional combinations of distributions of the features coloured by their galaxy class together with the decision boundary as learned by the logistic regression model in Figure 11. We only plot the results of the line minimization method here.

In the last panel, corresponding to the combination with the highest loss, we can see that the two classes are very much intertwined making it very difficult for the algorithm to separate the two. The figure also shows us why the combination of color and κ_{CO} work well together to predict the galaxy class. The two classes appear quite clearly as two distinct clusters with only a little bit of overlap in the center. The model attempted to draw its decision boundary between these two clusters, although visually it looks like the true best decision boundary might have been placed slight more to the bottom left.

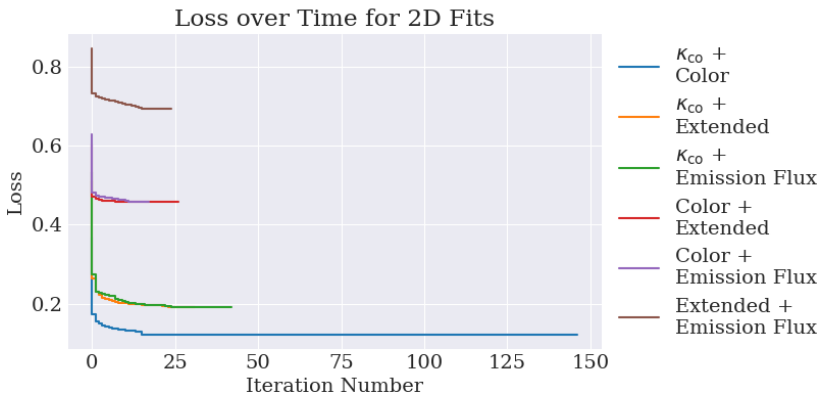

Figure 10: Loss value as a function of iteration number for each of the six possible combinations of two out of four features using logistic regression and line minimization as minimization method.

To investigate these results quantitatively we also compute the confusion matrix and the F1 score for each of these fits. The F1 score is defined as

$$\mathrm{F1} = \frac{2 \times \mathrm{precision} \times \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} \tag{5}$$

With

$$\mathrm{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{6}$$

$$\mathrm{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{7}$$

Both the precision and the recall are values between 0 and 1. Therefore the F1 score is also always a value between 0 and 1, where higher numbers correspond to better predictions. We calculate these values for all combinations of features, and present the results in Table 1.

Features	TN	TP	FN	FP	F1
κ_{co} + Color	476	472	28	24	0.9477911646586344
κ_{co} + Extended	439	440	60	61	0.8791208791208791
κ_{co} + Emission Flux	440	438	62	60	0.8777555110220441
Color + Extended	390	392	108	110	0.7824351297405189
Color + Emission Flux	389	393	107	111	0.7828685258964143
Extended + Emission Flux	370	91	409	130	0.2524271844660194

Table 1: Confusion matrix values laid out horizontally and the F1 score computed using these for each combination of two features.

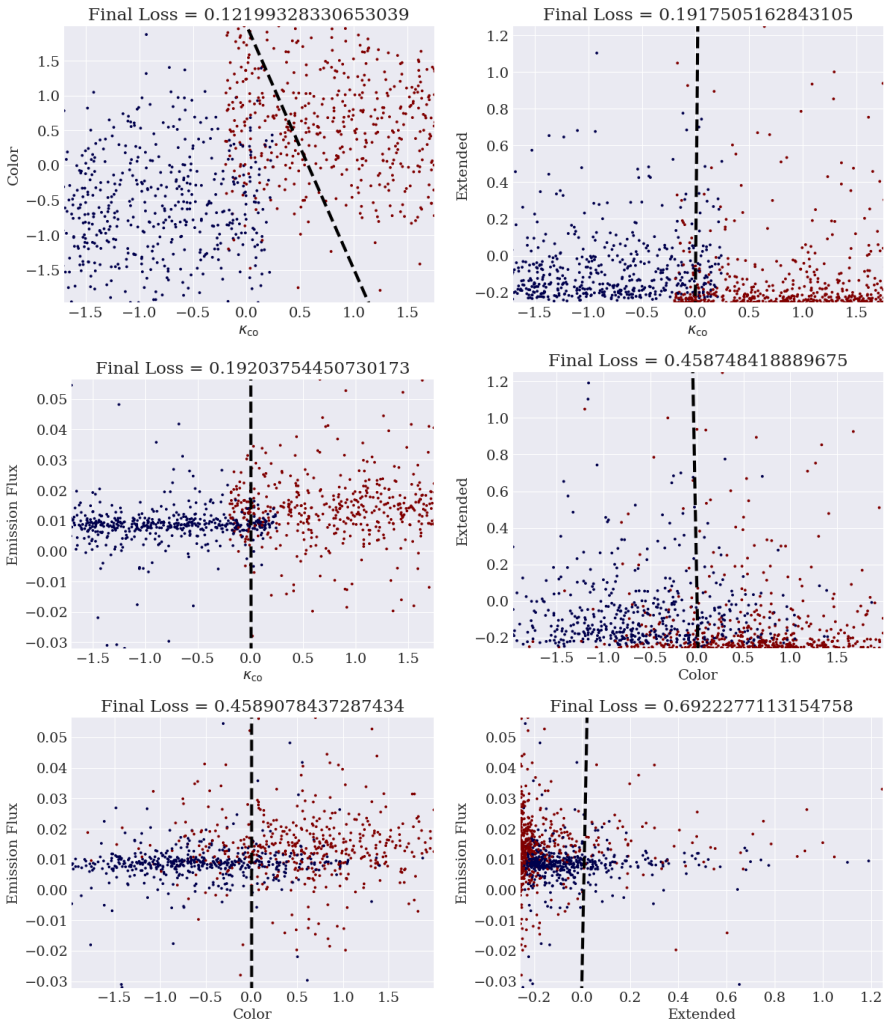

Figure 11: Distributions of each combination of features with the outer 2% cut out due to outliers. Black dashed lines indicate the decision boundary of a classification algorithm trained with logistic regression and line minimization using golden section search. The titles indicate the converged loss value.