NUR Hand-in Exercise 2

Angèl Pranger

March 18, 2025

Abstract

This document shows my solutions to hand-in exercise 2 of numerical methods in astrophysics.

1 Satellite galaxies around a massive central

General code used in this exercise is:

```python
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt

# Define constants
Nsat=100
a=2.4
b=0.25
c=1.6
xmin, xmax = 10**-4, 5

class n_class:
    def __init__(self,A,Nsat,a,b,c):
        """Class for functions n and p, such that the variable information does not
        have to be passed to them with every call."""
        self.A = A
        self.Nsat = Nsat
        self.a = a
        self.b = b
        self.c = c
        pass

    def n(self, x):
        """Returns the value of the function n at x."""
        return self.A*self.Nsat*(x/self.b)**(self.a-3)*np.exp(-(x/self.b)**self.c)

    def p(self, x):
        """Returns the value of the probability p=N/Nsat at x."""
        return 4*np.pi*x**2*self.A*(x/self.b)**(self.a-3)*np.exp(-(x/self.b)**self.c)

class RNG_class:
    def __init__(self, seed):
        """Class for random number generator such that the seed is passed once and the
    current state is remembered."""
        if (seed < 0):
            print("The seed for the RNG is invalid.")
        self.state1 = np.uint64(seed)
        self.m = np.uint64(4930622455819)
        self.a = np.uint64(3741260)
        self.fraction = 1/(self.m-1) # period is m-1
        pass

    def random(self):
        """Returns a uniformly random float between (0,1)."""
        # 64-bit XOR-shift method
        self.state1 = self.state1^(self.state1>>np.uint64(21))
        self.state1 = self.state1^(self.state1<<np.uint64(35))
```

```
        self.state1 = self.state1^(self.state1>>np.uint64(4))
        # Multiple linear congruential generator
        random = (self.a*self.state1) % self.m
        # Convert to interval (0,1)
        random = random * self.fraction
        return random
```

Q1.py

1.1 a

We have that

$$\begin{aligned}\int\int\int_V n(x)\mathrm{d}V &= \int\int\int n(x)x^2\sin(\phi)\mathrm{d}\phi\mathrm{d}\theta\mathrm{d}x \\ &= 4\pi\int n(x)x^2\mathrm{d}x \\ &= 4\pi\int x^2 A\langle N_{sat}\rangle(\frac{x}{b})^{a-3}\exp-(\frac{x}{b})^c\mathrm{d}x \qquad = \langle N_{sat}\rangle.\end{aligned}$$

To solve for A, we find that

$$\begin{aligned}4\pi A\int x^2(\frac{x}{b})^{a-3}\exp-(\frac{x}{b})^c\mathrm{d}x = 1 \\ A = \frac{1}{4\pi\int x^2(\frac{x}{b})^{a-3}\exp-(\frac{x}{b})^c\mathrm{d}x}.\end{aligned}$$

We are told to integrate from $x_{min} = 0$ to $x_{max} = 5$. There is an integrable singularity at the limit $x = 0$ (as dividing the integrand by zero is impossible), so an open method must be used to integrate this. We choose to use the extended midpoint Romberg method (which is Richardson extrapolation applied to the midpoint rule).

The code used for this exercise is given here:

```
def integrand(x,a=a,b=b,c=c):
    """Returns the integrand that is used to compute A."""
    return x**2*((x/b)**(a-3))*np.exp(-(x/b)**c)

def extended_midpoint_Romberg(f, a, b, m=8):
    """Computes the integral of f between a and b (with a<b) using the open method.
    Returns the best estimate value of the integral and an estimate for the error."""
    one_third = 1/3
    # Set initial step size
    h = b-a
    # Initialize array of estimates r of size m to zero
    r = np.zeros(m)
    # Calculate initial estimate using midpoint rule for N=1
    r[0] = h*f(0.5*(a+b))
    # Computing estimates for N=3, N=9, etc
    Np = 2
    for i in range(1, m):
        h *= one_third
        x = a + h*0.5
        # Add midpoint rule for new points
        j = 1 # Tracker of step size between new points (either h or 2h)
        for _ in range(Np):
            r[i] += h*f(x)
            x += (j+1)*h # j+1 alternates between 1 and 2
            j = (j+1)%2 # j alternates between 0 and 1
        # Add values for already calculated points
        r[i] += one_third*r[i-1]
        Np *= 3
    # Do weighted combinations
    Np = 1
    for i in range(1, m):
        Np *= 9
```

```
        factor = 1/(Np-1)
        for j in range(m-i):
            r[j] = (Np*r[j+1]-r[j])*factor
    return r[0], np.absolute(r[0]-r[1])

# Compute and print the found value for A
A = 1/(4*np.pi*extended_midpoint_Romberg(integrand, 0, xmax)[0])
print(f"A is {A}")
```

Q1.py

The resulting value for A is:

```
A is 9.194862922469179
```

output_Q1.txt

1.2 b

We have made a random number generator consisting of a 64-bit XOR-shift generator followed by a multiple linear congruential generator, as mentioned in class. As parameters, we use $a_1 = 21$, $a_2 = 35$ and $a_3 = 4$ for the XOR shift, which are the parameters advised in class and the book. For the MLCP we use $a = 3741260$ and $m = 4930622455819$, which are also numbers which are advised in the book. This results in a generator with a period of $m - 1$, because of which we can normalize the generated random integers to floats in the range (0,1) by dividing by $m - 1$. Testing the random number generator by having it generate 10000 random numbers results in the histogram shown in Figure 1. As seen in the plot, the random numbers indeed result in quite a uniform distribution of numbers between 0 and 1.

The code for this exercise is given here:

```
# Make object of class for n using the constants including the previously found A
n1 = n_class(A, Nsat, a, b, c)

# Seed random number generator
RNG1 = RNG_class(seed = 123456789)

# Testing random number generator
randoms = np.zeros(10000)
for i in range(len(randoms)):
    randoms[i] = RNG1.random()
plt.hist(randoms)
plt.title("Test: 10000 random numbers within (0,1)")
plt.xlabel("Random numbers")
plt.ylabel("Number")
plt.savefig('hist.png', dpi=600)

def rejection_sampling(p, xmin, xmax, num):
    """Takes a probability distribution p, interval bound by xmin and xmax. Samples num
    random
    draws from p within range (xmin, xmax) using rejection sampling.
    Returns array with num samples."""
    samples = np.zeros(num)
    i = 0
    while (i < num):
        # Draw a random x between xmin and xmax
        x = xmin + (xmax-xmin)*RNG1.random()
        # Draw random probability between (0,1) and check whether p(x) greater than this
        if (RNG1.random() < p(x)):
            samples[i] = x
            i += 1
    return samples

# Histogram in log-log space
samples = rejection_sampling(n1.p, xmin, xmax, 10000)
# 21 edges of 20 bins in log-space
edges = 10**np.linspace(np.log10(xmin), np.log10(xmax), 21)
hist = np.histogram(samples, bins=edges)[0]
# Correcting for bin width and normalization offset 10000/Nsat=100
```

```
hist_scaled = hist / (edges[1:]-edges[:-1]) * 0.01
# Getting analytical solution of N(x)/Nsat for values between xmin and xmax in log space
relative_radius = 10**np.linspace(np.log10(xmin), np.log10(xmax), 100)
analytical_function = n1.p(relative_radius)*Nsat # N(x)

# Plotting histogram and analytical solution
fig1b, ax = plt.subplots()
ax.stairs(hist_scaled, edges=edges, fill=True, label='Satellite galaxies')
plt.plot(relative_radius, analytical_function, 'r-', label='Analytical solution')
ax.set(xlim=(xmin, xmax), ylim=(10**(-3), 10**3), yscale='log', xscale='log',
       xlabel='Relative radius', ylabel='Number of galaxies')
ax.legend(loc='upper left')
plt.savefig('my_solution_1b.png', dpi=600)
```

Q1.py

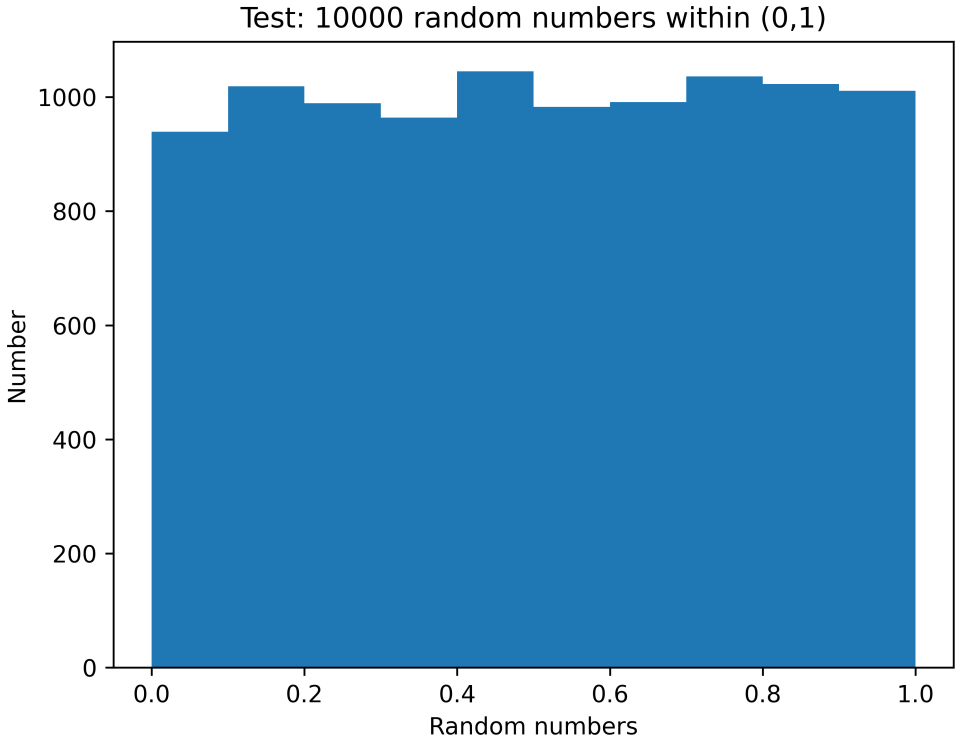

Figure 1: A test of 10000 randomly generator numbers is shown. The distribution looks quite uniform, indicating that the RNG works as expected.

We have that $N(x)\mathrm{d}x$ is the number of galaxies in the infinitesimal range $[x, x+\mathrm{d}x)$. To convert the number density $n(x)$ to $N(x)$, we have to integrate over ϕ and θ. This gives that $N(x) = 4\pi x^2 n(x)$.

To sample from the probability distribution $p(x) = N(x)/N_{sat}$, we have implemented a rejection sampling algorithm. We chose this algorithm, because it is relatively easy to implement and it turns out to run quite quickly (even though it is a relatively slow method as it takes two random numbers and return less than 1 per draw). The result of this sampling is shown in Figure 2.

1.3 c

We select 100 random satellite galaxies from the 10000 random samples from (b) with a method that selects every galaxy with equal probability, does not draw the same galaxy twice and does not reject any draw, by iteratively drawing one galaxy from the array, adding this to the new samples and removing it from the original array, until we have 100 draws. Removing the drawn galaxies from the original array

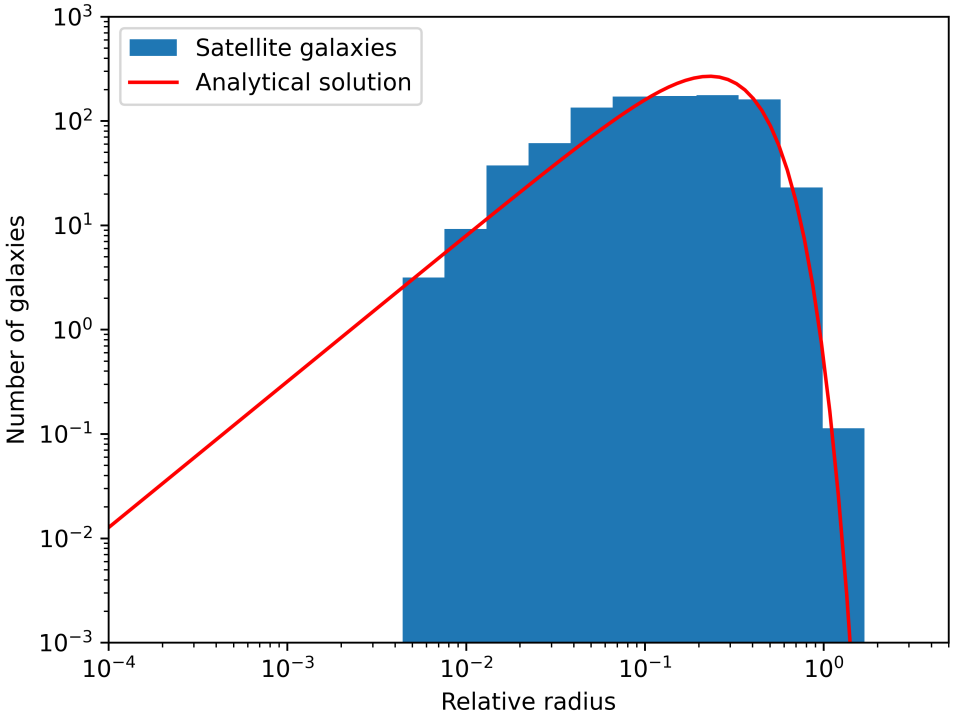

Figure 2: The histogram of 10000 sampled points from the distribution $p(x)$ are shown together with the analytical result for $N(x)$. The histogram is normalized both by the offset $10000/N_{sat} = 100$ and by the width of the bins, in order to compare the shape with the analytical result. We see that the histogram follows the analytical shape very nicely for large x. However, for x smaller than a certain threshold (about 0.005), the bins are all empty. This can probably be explained because our random number generator does not produce values which are as small.

ensures we do not draw the same galaxy twice, such that we do not have to reject any draw. We draw a discrete index between 0 and the "length of the array-1=N-1" by sampling between (0,1) and scaling multiplying by N to scale to (0, N), after which we take the floor such that the number is a random integer in [0, N-1].

We continue to sort the drawn galaxies using quicksort, as this method is the quickest method for the average typical case. The result is shown in Figure 3, where the number of galaxies within a radius is plotted.

The code for this exercise is given here:

```
152 def quicksort_recursive(array, low, high):
153     """Recursively sort array from low to high using quicksort algorithm. Alters the
    passed array."""
154     # Set pivot to middle element
155     pivot = np.int64(np.ceil((low+high)/2))
156     x_pivot = array[pivot]
157     # Looping to sort elements with respect to the pivot
158     i = low
159     j = high
160     while True:
161         while (array[i] < x_pivot):
162             i += 1
163         while (array[j] > x_pivot):
164             j -= 1
165         if (j<=i):
166             break
```

```
        else:
            mem = array[i]
            array[i] = array[j]
            array[j] = mem
            # If pivot is swapped, change location of pivot to new location
            # Let complementary indexer continue to prevent infinite looping in case array[i]=array[j]=array[pivot]
            if (i == pivot):
                pivot = j
                i += 1
            elif (j == pivot):
                pivot = i
                j -= 1
    # Apply algorithm recursively to subarrays left and right of the pivot
    if (low < pivot-1):
        array = quicksort_recursive(array, low, pivot-1)
    if (high > pivot+1):
        array = quicksort_recursive(array, pivot+1, high)
    return array

def quicksort(array):
    """Sort array using quicksort algorithm. Returns the sorted array."""
    # Sort first, last and middle element as pre-step
    ar = np.array([array[0], array[-1], array[(array.shape[0])>>1]])
    low = np.min(ar)
    middle = np.median(ar)
    high = np.max(ar)
    array[0] = low
    array[-1] = high
    array[(array.shape[0])>>1] = middle
    # Apply quicksort algorithm to the array
    quicksort_recursive(array, 0, array.shape[0]-1)
    return array

def random_samples_from_array(array, num):
    """Draw num random samples from array according to the following rules:
    1. Selects every item with equal probability.
    2. Does not draw any item twice.
    3. Does not reject any draw.
    Returns an array with the num drawn samples."""
    samples = np.zeros(num)
    for i in range(num):
        # Draw random integer (index) in [0, N-1] with N the current size of array
        idx = np.int64(np.floor(array.shape[0]*RNG1.random()))
        # Add this item to the samples
        samples[i] = array[idx]
        # Remove this item from the original list so as to not draw it again
        array = np.delete(array, idx)
    return samples

# Select 100 random samples from the previous 10000 samples from (b)
samples_100 = random_samples_from_array(samples, 100)
samples_100 = quicksort(samples_100)

# Cumulative plot of the chosen galaxies
fig1c, ax = plt.subplots()
ax.plot(samples_100, np.arange(100))
ax.set(xscale='log', xlabel='Relative radius',
       ylabel='Cumulative number of galaxies',
       xlim=(xmin, xmax), ylim=(0, 100))
plt.savefig('my_solution_1c.png', dpi=600)
```

Q1.py

1.4 d

To compute the derivative of n at $x = 1$ numerically we use Ridder's differentiation. The analytical result is given by taking the analytical derivative of n.

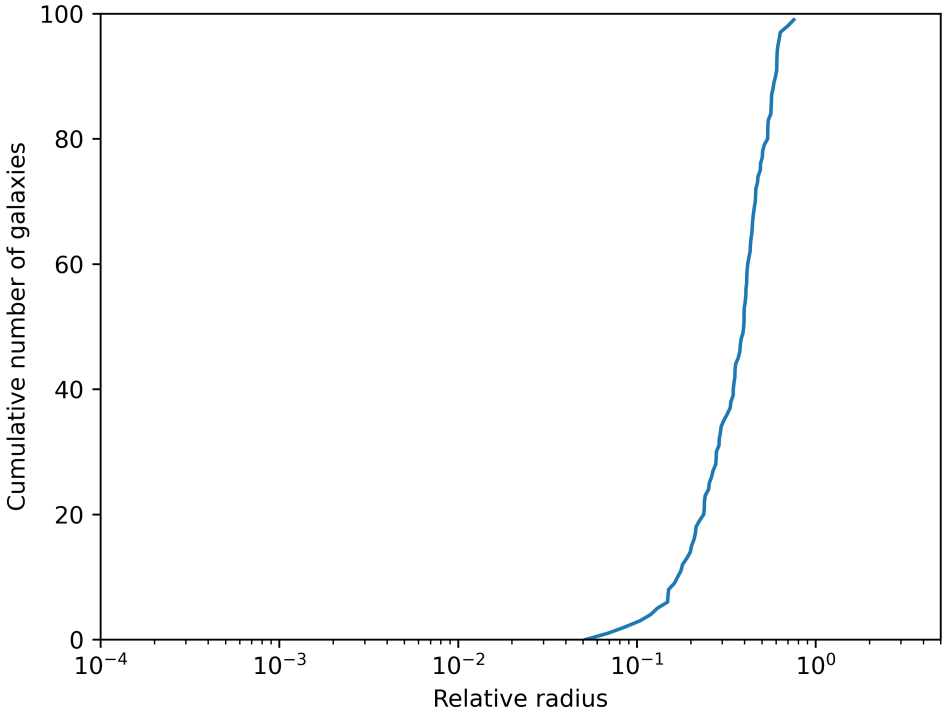

Figure 3: The number of galaxies within a radius is plotted (a cumulative distribution). It indeed looks like a cumulative distribution, which indicates that the samples are sorted correctly.

The used code is given here:

```
def analytical_derivative_n(x,A,Nsat,a,b,c):
    """Returns the value of the analytical derivative of n at the point x."""
    fraction = 1/b
    return A*Nsat*fraction*((a-3)*(x*fraction)**(a-4)-c*(x*fraction)**(a+c-4))*np.exp(-(
    x*fraction)**c)

def Ridders_differentiation(f, x, m=5, h=0.1, d=2, target_error=1e-13):
    """Applies Ridders differentiation on the function f at the point x.
    Returns the numerical derivative of f at x together with an estimate of the error.
    """
    # Calculate first approximations to f'(x) using central differences
    approx = np.zeros(m)
    d_inverse = 1/d
    error_old = np.inf
    for i in range(m):
        approx[i] = (f(x+h)-f(x-h)) * 0.5 / h
        h *= d_inverse
    # Combine pairs of approximations
    for j in range(1, m):
        power = np.power(d, 2*j)
        factor = 1/(power-1)
        for i in range(m-j):
            approx[i] = (power*approx[i+1]-approx[i])*factor
        # Terminate when improvement over previous best approximation is smaller than
    target error
        error_new = np.absolute(approx[0]-approx[i+1])
        if (error_new < target_error):
            return approx[0], error_new
        # Terminate early if the error grows, return best approximation from before
        if (error_old < error_new):
```

```
            return approx[i+1], error_old
        error_old = error_new
    return approx[0], error_new

# Compute the analytical and numerical derivatives
print(f"The analytical derivative of n at x=1 is {analytical_derivative_n(1,A,Nsat,a,b,c)}")
print(f"The numerical derivative of n at x=1 is  {Ridders_differentiation(n1.n,1,m=15,h=0.1)[0]}")
```

Q1.py

The resulting values for the derivate are:

```
The analytical derivative of n at x=1 is  -0.6253290521635682
The numerical derivative of n at x=1 is   -0.6253290521635598
```

output_Q1.txt

The analytical and numerical derivative are the same up to 13 significant digits.

2 Heating and cooling in HII regions

The code of any shared modules for question 2 is given by:

```
#!/usr/bin/env python
import numpy as np
import time

k = 1.38e-16 # erg/K
aB = 2e-13 # cm^3 / s
A = 5e-10 # erg
xi = 1e-15# /s
Z = 0.015
psi = 0.929
Tc = 1e4 # K

# here no need for nH nor ne as they cancel out
def equilibrium1(T, Z=Z, Tc=Tc, psi=psi):
    """Returns value of the function of which the root must be found for 2a."""
    return psi*Tc*k - (0.684 - 0.0416 * np.log(T/(1e4 * Z*Z)))*T*k

class Equilibrium2:
    def __init__(self, nH):
        """Class for equilibrium2 such that nH does not have to be passed with every function call."""
        self.nH = nH
        pass

    def equilibrium2(self, T):
        """Returns value of the function of which the root must be found for 2b."""
        return (psi*Tc - (0.684 - 0.0416 * np.log(T/(1e4 * Z*Z)))*T - .54 * ( T/1e4 )**.37 * T)*k*self.nH*aB + A*xi + 8.9e-26 * (T/1e4)

def bisection_root_step(f, a, b):
    """Takes a function f and bracket (a,b).
    Returns the new bracket found by bisection."""
    # Find middle of a and b
    c = (a+b)*0.5
    # Check whether a or b forms bracket with c
    if (f(a)*f(c) < 0):
        b = c
    else:
        a = c
    return a, b

def false_position_method(f, a, b, max_iterations=100, target_abs=0.1, target_rel=1e-10, safeguards=False):
    """Finds the root of a function using the false position method.
```

```
    Stops after max_iterations or when the target accuracy is met.
    Returns interval a, b enclosing the root and the number of iterations used."""
    for steps in range(max_iterations):
        # Save value for f(a) as we need it again later
        f_a = f(a)
        # Linearly estimate root from 2 last guesses (a and b)
        c = b-(b-a)/(f(b)-f_a)*f(b)
        # Find the counterpoint
        if (f_a*f(c) < 0):
            # Apply bisection if the interval has not reduced by at least half
            if ( safeguards & (c > 0.5*(a+b)) ):
                a, b = bisection_root_step(f, a, b)
            else:
                b = c
        else:
            # Apply bisection if the interval has not reduced by at least half
            if ( safeguards & (c < 0.5*(a+b)) ):
                a, b = bisection_root_step(f, a, b)
            else:
                a = c
        if (((b-a) < np.absolute(target_rel*a)) | ((b-a) < target_abs)):
            break
    return a, b, steps+1

def root_from_interval(f, a, b):
    """Return the approximate root value a or b, choosing the one for which f(x) is
    closest to zero."""
    if (np.absolute(f(a)) < np.absolute(f(b))):
        return a
    else:
        return b
```

Q2.py

2.1 a

The code used is given by:

```
# Compute and time the equilibrium temperature (root) from equilibrium1
start = time.time()
a, b, iterations = false_position_method(equilibrium1, 1, 1e7, target_abs=0.1,
    safeguards=True)
print(f"The equilibrium temperature (root) is {root_from_interval(equilibrium1, a, b)} K
     with an error estimate of {b-a:.3} K.")
end = time.time()
print(f"The execution time is {(end-start)*10**3:.3} ms.")
print(f"The number of iterations used is {iterations}.")
```

Q2.py

We calculate the equilibrium temperature by finding the root of equilibrium1 (as given in the code). The functions are of order lower than quadratic, which is why we don't expect Brent's method to work optimally. Therefore, we choose the linear false position method combined with bisection to overcome slow convergence in very non-linear regions. This is implemented by using false position in general but switching to bisection whenever the bracket is not at least decreased by half in size, inspired by the safeguards used in Brent's method. The algorithm stops (in this case) when an absolute error less than 0.1 K is reached. The results are given here:

```
The equilibrium temperature (root) is 32539.14137090274 K with an error estimate of
    0.0683 K.
The execution time is 0.211 ms.
The number of iterations used is 18.
```

output_Q2.txt

2.2 b

The results for root finding of the function equilibrium2 (as given in the code) are given here. We find that the higher the density, the lower the temperature, which is as expected. We further note that the error on the higher temperature is larger than the error on the lower temperatures, which is because the algorithm stops at a relative target accuracy of 10^{-10}.

The code used is given by:

```
# Compute and time the equilibrium temperature (root) from equilibrium2 for different nH
print(f"The equilibrium temperature (root) and estimated error are given.")
print("n_e [cm$^{-3}$]  T_equilibrium [K]  estimated absolute error [K]  time [ms]  iterations")
for n_e in [1e-4, 1, 1e4]:
    start = time.time()
    func_class = Equilibrium2(n_e)
    a, b, iterations = false_position_method(func_class.equilibrium2, 1, 1e15, target_abs=1e-10, target_rel=1e-10, safeguards=True)
    end = time.time()
    print(f"{n_e}    {root_from_interval(func_class.equilibrium2, a, b)}    {b-a:.3}    {(end-start)*10**3:.3}    {iterations}")
```

Q2.py

The results:

```
The equilibrium temperature (root) and estimated error are given.
n_e [cm$^{-3}$]  T_equilibrium [K]  estimated absolute error [K]  time [ms]  iterations
0.0001    160647887536815.62    9.61e+03    0.515    36
1    33861.300235178554    6.55e-11    0.802    57
10000.0    10525.88601966122    1.82e-12    0.924    57
```

output_Q2.txt