Handin Assignment 2

Ioannis Koutalios s3365530

January 6, 2025

Abstract

Code and results for Handin assignment 2 for the course Numerical Recipes in Astrophysics.

1 Satellite galaxies around a massive central

For satellite galaxies from near the centre up to 5 virial radii, we can assume a density profile that follows the equation

$$n(x) = A\langle N_{sat}\rangle \left(\frac{x}{b}\right)^{a-3} \exp[-\left(\frac{x}{b}\right)^{c}] \tag{1}$$

where x is the radius relative to the virial radius($x = r/r_{vir}$) and a, b, c are free parameters which for our purposes we assume to be $a = 2.4$, $b = 0.25$, $c = 1.6$. $\langle N_{sat}\rangle$ is the average total number of satellites, which for this exercise is 100, and A is a normalization factor.

1.1 Numerical Integration

We want to calculate A. To do that we need to solve

$$\iiint_V n(x)dV = \langle N_{sat}\rangle \tag{2}$$

which can be written as

$$\int_{\phi=0}^{2\pi}\int_{\theta=0}^{\pi}\int_{x=0}^{5} n(x)x^2\sin(\theta)d\phi d\theta dx = 4\pi\int_{x=0}^{5} n(x)x^2dx = \langle N_{sat}\rangle \tag{3}$$

We want to numerically solve the final integral. To do that we define our function "f_n" which is equivalent to $4\pi n(x)x^2$. We then define the "trapezoid" function which uses the trapezoid rule to calculate the integral. We can directly integrate using this function but we will be more accurate if we integrate using Romberg's algorithm. The function "romberg" applies this algorithm. It initializes an array of integrals by calling the "trapezoid" function for an increasing number of points and then uses Richardson's extrapolation to combine these estimates and return a final value. The order of the extrapolation is m and can be specified when calling the function.

```
import numpy as np

SAT = 100      # number of satellite galaxies
A = 2.4        # power-law index
B = 0.25       # scale radius
C = 1.6        # exponential index

def Nx(x: float, norm: float) -> float:
    '''
    Distribution function
    '''
    return norm*SAT*((x/B)**(A-3))*np.exp(-(x/B)**C)

def Fx(x: float, norm: float) -> float:
    '''
```

```
    Function to be integrated
    '''
    return 4*np.pi*norm*SAT*((1/B)**(A-3))*np.exp(-(x/B)**C)*x**(A-3+2)

def FxNorm(x: float) -> float:
    '''
    Wrapper function to include the normalization factor
    '''
    return Fx(x, norm)

def Trapezoid(func: callable, low: float, up: float, n: int) -> float:
    '''
    Integrate using trapezoid algorithm
    func: function
    low: lower limit
    up: upper limit
    n: number of points used for calculations
    '''
    x = np.linspace(low, up, n)
    h = x[1]-x[0]
    integ = h*(func(x[0])/2+np.sum(func(x[1:-1]))+func(x[-1])/2)
    return integ

def Romberg(func: callable, low: float, up: float, n: int, m: int) -> float:
    '''
    Integrate using Romberg's algorithm
    func: function
    low: lower limit
    up: upper limit
    n: number of points used for initial calculations
    m: order of extrapolation
    '''
    ints = np.zeros(m)
    for i in range(m):
        ints[i] = Trapezoid(func, low, up, n)
        n *= 2
    for j in range(m):
        for i in range(0, m-j-1):
            ints[i] = (4**(j + 1)*ints[i+1]-ints[i])/(4**(j+1)-1)
    return ints[0]

if __name__ == '__main__':
    # Define the range of integration
    low, up = 0, 5

    norm = 1  # Initial guess for normalization
    norm = 100 / Romberg(FxNorm, low, up, n=100, m=5)
    print(f'NORM = {norm:.10}')
```

open_int.py

In the main part of the script, we calculate the integral using Romberg's algorithm for an initial number of points $n = 10\,000$ and an extrapolation order of $m = 5$. We then calculate A by dividing 100 by the value we calculated and print A using 10 significant digits. The output of the script can be seen below.

```
NORM = 9.194856403
```

output/open_int.txt

1.2 Distribution

For the next task, we have to sample galaxies using the distribution. The probability distribution of the relative radii should be $p(x)dx = N(x)dx/\langle N_{sat}\rangle$, where $N(x)$ is:

$$N(x) = \int_{\phi=0}^{2\pi}\int_{\theta=0}^{\pi} n(x)x^2\sin(\theta)d\phi d\theta = 4\pi n(x)x^2 \tag{4}$$

We also want to uniformly sample the angle ϕ, θ in their respective range.

To do that we first import from our previous script the function "n" with the defined parameters. We also read the calculated value of A from the output of our previous script. We then define "f_N" to be our $N(x)$ function from above.

We create a function "lcg" for generating random numbers using the algorithm for a Linear Congruential Generator. We normalize the numbers we return so they are uniformly distributed between 0 and 1. The values we selected for the multiplier, increment and modulus come from previous implementations of the same algorithm in many scientific libraries. The "uniform" function utilizes the "lcg" to uniformly generate values in any given range.

Our next function "sampling" uses rejection sampling to generate values for a given distribution. We use two seeds one for the numbers we generate and one for the testing values to perform the rejection process. We want to sample N values so we generate more to account for the rejection process. The final array of generated values is of length N.

```
import numpy as np
import matplotlib.pyplot as plt
from open_int import Nx

# import the value of A that was calculated in the previous script
with open('output/open_int.txt') as f:
        lines = f.readlines()
        NORM = float(lines[0].split('=')[1])

def Distribution(x: float) -> float:
    '''
    Distribution function for the number of galaxies
    4*pi*n(x)*x^2
    '''
    return 4*np.pi*Nx(x,NORM)*(x**2)

def LCG(seed: int = 0, n: int = 10, a: int = 1103515245, c: int = 12345, m: int = 2**31)
    -> np.ndarray:
    '''
    random number generator
    uses Linear Congruential Generators
    generates uniform distribution between 0 and 1
    seed: used for reproducible results
    n: number of points
    a: multiplier
    c: increment
    m: modulus
    '''
    numb = np.zeros(n)
    for i in range(n):
        seed = (a * seed + c) % m
        numb[i] = seed / m
    return numb

def Uniform(a: float = 0, b: float = 1, n: int = 1, seed: int = 42) -> np.ndarray:
    '''
    uniform number generator
    a: lower limit
    b: upper limit
    n: number of points
    seed: used for reproducible results
    '''
    return a+(b-a)*LCG(seed,n=n)

def Sampling(f: callable, a: float, b: float, N: int, seed1: int = 24, seed2: int = 42,
    mult: int = 100) -> np.ndarray:
    '''
    sampling using rejection sampling
    f: distribution
    a: low limit
    b: upper limit
    seed1, seed2: used for reproducible results
    '''
```

```
    n = mult*N

    x = Uniform(a,b,n=n,seed=seed1)
    y = Uniform(n=n,seed=seed2)

    max_val = np.amax(f(x))
    samp=np.zeros(N)
    m = 0
    i = 0

    while m<N:
        if i == n:
            break

        if y[i]<f(x[i])/max_val:
            samp[m] = x[i]
            m+=1
        i+=1

    return np.array(samp)


if __name__ == '__main__':

    N = 10000
    sample = np.array([Sampling(Distribution,10**-4,5,N,1312,1234),Uniform(0,2*np.pi,N),
    Uniform(0,np.pi,N)])
    np.save('output/sample',sample)

    x = np.linspace(10**-4,5,10000)
    y = Distribution(x)

    plt.hist(sample[0],bins=np.logspace(np.log10(10**-4),np.log10(5), 20),density=True,
        color='yellow',edgecolor='black',label='Sattelite Galaxies')
    plt.plot(x,y/100,color='red',label=r'$N(x)/ 100$')
    plt.xscale('log')
    plt.yscale('log')
    plt.legend()
    plt.xlabel('Relative Radius')
    plt.ylabel('Number of galaxies')
    plt.savefig('plots/distr.png', dpi=300)
    plt.close()

    plt.hist(sample[0],bins=np.logspace(np.log10(10**-4),np.log10(5), 20),density=True,
        color='yellow',edgecolor='black',label='Sattelite Galaxies')
    plt.plot(x,y/100,color='red',label=r'$N(x)/ 100$')
    plt.xscale('log')
    plt.yscale('log')
    plt.ylim(1e-3,10)
    plt.xlim(1e-3,10)
    plt.legend()
    plt.xlabel('Relative Radius')
    plt.ylabel('Number of galaxies')
    plt.savefig('plots/distr_zoom.png', dpi=300)
    plt.close()
```

distr.py

In the main part of our script, we generate our sampled galaxies. We create an array which has a shape of $(3 \times 10\,000)$ to store for each galaxy the relative radius, and the two angles. We then store it for the next task and create the histogram plots. Along with the histogram plots, we also plot $N(x)/100$. The correction of $N(x)$ is to account for the normalization offset of $10\,000/\langle N_{sat} \rangle = 100$

As we can see in Figure 1 there is a great match between the normalized distribution and our sample of generated galaxies. On the right panel, we can see that the line crosses every bin at its top, as we would expect.

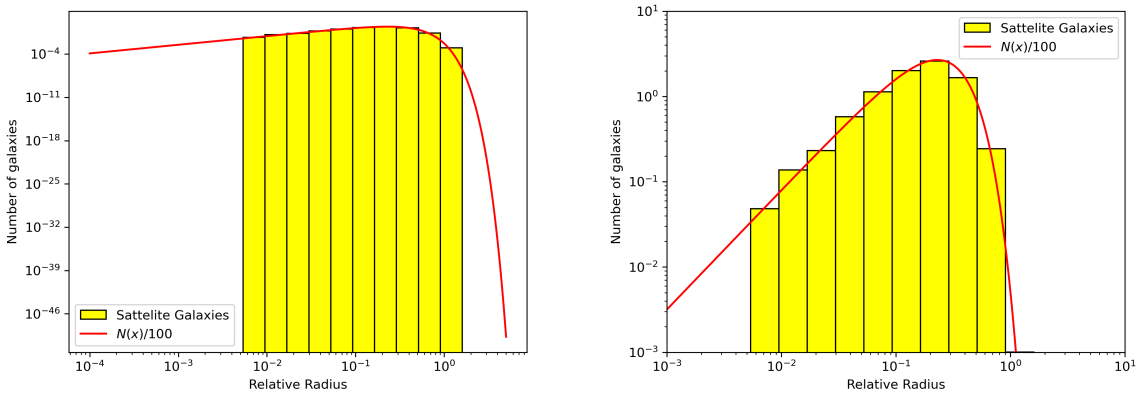

Figure 1: A histogram of the radii of the generated galaxies. Both the x and y axis, are in logarithmic scale. On the left panel, we can see the whole range that has a big empty space because the probability of the distribution is extremely low. On the right panel, we focus on the populated area of the distribution to see that our predictions match the given distribution.

1.3 Sorting 100 random galaxies

We now want to select 100 galaxies from our sample. The process has to be random with every galaxy having an equal probability of being selected. We should have no duplicate galaxies in our selection and the process has to be done without rejecting any selection. We then want to sort the selected galaxies based on their relative radius and create a plot of galaxies within a radius.

To implement all that we first create a "random_selection" function that uses the Fisher-Yates algorithm to completely randomize the array. We can then select the first N elements. This process follows all the principles we described, as every galaxy has an equal chance of getting selected, the elements are unique and we use no rejection. The Fisher-Yates algorithm works by utilizing our "lcg" generator. We first select N random numbers to be used as seeds. We then go through the array backwards and every time we generate a random integer in the range $[0, i]$. We then swap the elements. The final array is in random order. We can then choose the first N elements.

To sort the selected galaxies we use the function "selection_sort" that implements the known algorithm. Selection sort uses two loops. The second loop starts from the index of the first one, scans the unsorted part of the array to find the smallest element and then swaps it to the top of the sorted part of the array. When we finish the loops we will have an array that goes from the smallest to the biggest value.

```python
import numpy as np
import matplotlib.pyplot as plt
from distr import Uniform

def RandomSelection(ar: np.ndarray, N: int = 100, seed: int = 12) -> np.ndarray:
    '''
    uses Fisher-Yates algorithm to randomize the array
    then selects N first elements
    ar: array to select from
    N: number of values to be selected
    seed: used for reproducible results
    '''
    array = np.copy(ar)
    leng = len(array)
    random_seed = Uniform(n=leng, seed=seed)

    for i in range(leng - 1, 0, -1):
        rand = Uniform(0,i,seed=random_seed[i])
        j = int(rand)
        array[i], array[j] = array[j], array[i]

    return array[:N]
```

```python

def SelectionSort(a: np.ndarray, inplace: bool = True) -> np.ndarray:
    '''
    sorting by using the selection sort algorithm
    a: array to be sorted
    inplace: if False then original array remains unsorted
    '''
    if inplace:
        ar = a
    else:
        ar = np.copy(a)

    for i in range(len(ar)-1):
        i_min = i
        for j in range(i+1,len(ar)):
            if ar[j]<ar[i_min]:
                i_min = j
        if i_min != i:
            ar[i] , ar[i_min] = ar[i_min] , ar[i]


    return ar

if __name__ == '__main__':

    samp = np.load('output/sample.npy')
    samp = samp[0]
    rand_samp = RandomSelection(samp, N=100, seed=2)

    rand_samp = SelectionSort(rand_samp)

    plt.plot(rand_samp,np.arange(1,101,1),color='crimson',label='Sattelite Galaxies')
    plt.xscale('log')
    plt.xlim(10**-4,5)
    plt.legend()
    plt.ylabel('Number of galaxies within radius')
    plt.xlabel('Relative Radius')
    plt.savefig('plots/rand_sort.png', dpi=300)
```

sort.py

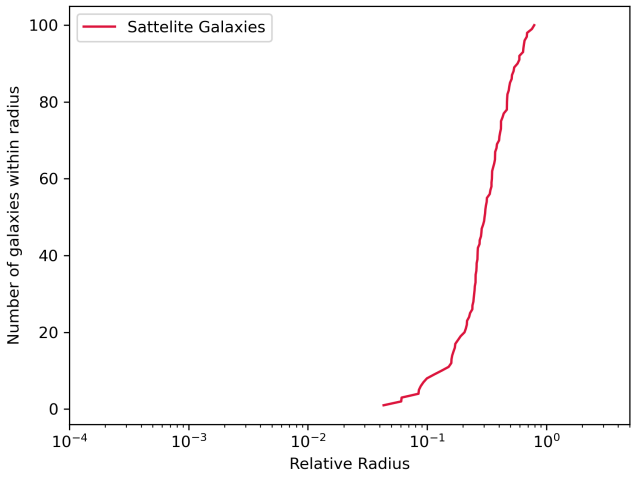

Figure 2: The number of galaxies within a radius. We plot the 100 selected galaxies from our distribution and we plot how many galaxies we find within each value of the relative radius. The x-axis is in logarithmic scale.

In the main part of our script, we load the sampled galaxies from our previous script. We then randomly select 100 elements which we then sort based on the value of the radius. We then want to plot the number of galaxies within a radius. We utilize the fact that we have the array sorted to use the values of radius as the x-values of our plot. For the y-values, we just need to increase by 1 every time a new element is found, which is done with the "np.arange" function.

As we can see in Figure 2 the sorting process worked at the plot is monotonically increasing within its range.

1.4 Numerical Differentiation

In this part, we want to calculate the derivative of $n(x)$ at $x = 1$. We aim for a precision of 12 significant digits or higher. To make sure of that we will compare our calculated value with the analytical solution of the derivative that we implemented in the function "analyt_der".

In our script, we import the function "n" with its parameters and also read the value of A from the text file. The function "f_n" acts as a wrapper, allowing us to define all the non-changing parameters. We then define the "central_dif" function that calculates the derivative using the central difference algorithm. This function will be used inside the "ridder" function that uses Ridder's algorithm. The algorithm calculates the derivative multiple times using central differences for a decreasing step h and then uses Richardson's extrapolation to combine the values. In our implementation the extrapolation stops when the estimated error increases, which allows us to give big values of m, knowing that it will not negatively affect our result.

```
import numpy as np
from open_int import Nx, SAT, A, B, C
from distr import NORM

def NxNorm(x: float) -> float:
    '''
    Wrapper function to include the normalization factor
    '''
    return(Nx(x,NORM))

def AnalytDeriv(x: float) -> float:
    '''
    function for the analytical derivative of n
    '''
    return NORM*SAT*((((A-3)/(B**(A-3)))*(x**(A-4))*np.exp(-((x/B)**C)))
        -(C/(B**C))*(((x/B)**(A-3))*(x**(C-1))*np.exp(-((x/B)**C))))

def CentralDif(func: callable, x: float, step: float = .1) -> float:
    '''
    derivative using central differences
    step: step for the calculation of the derivative
    '''
    return (func(x+step)-func(x-step))/(2*step)

def Ridder(func: callable, x: float, step: float = .1, d: int = 2, m: int = 5) -> float:
    '''
    derivative using ridder's algorithm
    func: function
    x: point to calculate the derivate at
    step: step for initial central difference derivative
    d: decrease factor for step
    m: maximum number of extrapolations
    '''
    der = np.zeros(m)

    for i in range(m):
        der[i] = CentralDif(func,x,step)
        step = step/d

    prev_der = der[0]
    prev_er = 100.0
    error = 10

```

```
    for j in range(m):
        for i in range(0,m-j-1):
            der[i] = der[i+1] + (der[i+1]-der[i])/(2**(j+1) -1)

        error = np.abs(der[0]-prev_der)

        if error-prev_er > 0:
            return prev_der # early stopping

        prev_er = error
        prev_der = der[0]

    return der[0]

if __name__ == '__main__':

    der = Ridder(NxNorm,1,step=.01,m=10)
    print(f'numerical dn/dx = {der:.13}')

    analyt = AnalytDeriv(1)
    print(f'analytic dn/dx  = {analyt:.13}')

    print(f'abs difference  = {np.abs(analyt-der):.1}')

```

deriv.py

In the main part of our script, we calculate the derivative using the "ridder" function. Notice that while we pass $m = 10$, the function doesn't necessarily extrapolate to the 10th order as it can terminate before that if the estimated error increases.

We also calculate the derivative by using the analytic function and print both with 14 significant digits. We also print the absolute difference between the two values.

```
numerical dn/dx = -0.6253286087841
analytic dn/dx  = -0.6253286087841
abs difference  = 2e-14
```

output/deriv.txt

As we can see the derivative we calculated matches the analytic one in all 14 significant digits. In fact, the absolute difference between the two values is 10^{-14}.

2 Heating and cooling in HII regions

In this section, we will numerically solve equations describing the process of heating and cooling HII regions. All the values and coefficients are given in c.g.s. units.

2.1 First equation

In our first approach, we only consider one heating and one cooling mechanism. The heating from photoionization can be described by:

$$\Gamma_{pe} = \alpha_B n_H n_e \psi k_B T_c \tag{5}$$

where $\alpha_B = 2 \cdot 10^{-13}$ is the B recombination coefficient, n_H and n_e are the densities of hydrogens and electrons, $\psi = 0.929$ is a numerical value, and $T_c = 10^4$ is the stellar temperature.

The cooling mechanism in our scenario is by radiative recombination which follows the equation

$$\Lambda_{rr} = \alpha_B n_H n_e \langle E_r r \rangle \tag{6}$$

where

$$\langle E_r r \rangle = \left[0.684 - 0.0416 \ln \left(\frac{T}{10^4 Z^2} \right) \right] k_B T \tag{7}$$

To find the equilibrium temperature we need

$$\alpha_B n_H n_e \psi k_B T_c = \alpha_B n_H n_e \left[0.684 - 0.0416 \ln\left(\frac{T}{10^4 Z^2}\right)\right] k_B T \tag{8}$$

or equivalently:

$$\psi T_c - \left[0.684 - 0.0416 \ln\left(\frac{T}{10^4 Z^2}\right)\right] T = 0 \tag{9}$$

In our script, we define the equation we want to solve and use a wrapper to define all the values except the temperature. We then define two functions for root finding, "false_position" and "newton_raphson".

The false-position algorithm uses a bracket $[a, b]$ and calculates the new point at $c = \frac{af_b - bf_a}{f_b - f_a}$. It then finds in which of the brackets $[a, c]$ or $[c, b]$ we have a change of sign to limit the original bracket. The process is repeated until we have a convergence to a predefined margin or the maximum number of iterations has been completed, in which case an error is raised.

In the Newton-Raphson algorithm, we have a single value which is being updated, using the value of the derivative. To implement that we use the central difference algorithm from the previous task. The new point is calculated by: $a_{new} = a - \frac{f_a}{f'_a s}$.

The estimated error in both of these algorithms, which is being used to determine the convergence, is the relative error, which means that we divide it by the estimated true value.

```
import numpy as np
from deriv import CentralDif

# here no need for nH nor ne as they cancel out, we also cancel out k
def Equilibrium1(T: float, Z: float, Tc: float, psi: float) -> float:
    '''
    Equilibrium equation 1
    '''
    return psi*Tc - (0.684 - 0.0416 * np.log(T/(1e4 * Z*Z)))*T

def F1(x: float) -> float:
    '''
    wrapper for equilibrium1
    '''
    return Equilibrium1(x, Z=.015, Tc=10**4, psi=.929)

def FalsePosition(f: callable, a: float, b: float, delta: float = 1e-11, maxiter: int =
    10000) -> tuple:
    '''
    root finding using false-position algorithm
    f: function
    a: low bracket
    b: upper bracket
    delta: maximum relative accuracy
    maxiter: max number of iterations before stopping
    '''

    fa = f(a)
    fb = f(b)

    if fa * fb > 0:
        raise ValueError("Same sign at both ends of the interval.")

    for i in range(maxiter):
        c = (a * fb - b * fa) / (fb - fa)
        fc = f(c)

        if fa * fc < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc

        if (b-a)/c < delta:
            return c, (b-a)/c, i
```

```
46
47      raise RuntimeError("False Position method did not converge.")
48
49  def NewtonRaphson(f: callable, a: float, delta: float = 1e-11, step: float = .1, maxiter
        : int = 10000) -> tuple:
50      '''
51      root finding using Newton-Raphson algorithm
52      f: function
53      a: starting point
54      delta: maximum relative accuracy
55      step: step for the central difference algorithm
56      maxiter: max number of iterations before stopping
57      '''
58      for i in range(maxiter):
59          fa = f(a)
60          der = CentralDif(f,a,step=step)
61          dx = fa / der
62          a -= dx
63          if abs(dx/a) < delta:
64              return a, abs(dx/a), i
65      raise RuntimeError("Newton-Raphson algorithm did not converge.")
66
67
68  if __name__ == '__main__':
69
70      low, up = 1, 10**7
71      print('Using False-Position\n')
72      T_fal, error_fal ,iterations_fal = FalsePosition(F1,low,up)
73
74      print(f'T = {T_fal:.12}')
75      print(f'Accuracy = {error_fal:.1}')
76      print(f'number of iterations = {iterations_fal}')
77
78
79      print('\n\nUsing Newton-Raphson\n')
80
81      a = 2e5
82      print(f'Starting from a = {a:.1e}')
83      T_new_5, error_new_5 ,iterations_new_5 = NewtonRaphson(F1,a)
84
85      print(f'T = {T_new_5:.12}')
86      print(f'Accuracy = {error_new_5:.1}')
87      print(f'number of iterations = {iterations_new_5}')
88
89      a = 1e4
90      print(f'\nStarting from a = {a:.1e}')
91      T_new_4, error_new_4 ,iterations_new_4 = NewtonRaphson(F1,a)
92
93      print(f'T = {T_new_4:.12}')
94      print(f'Accuracy = {error_new_4:.1}')
95      print(f'number of iterations = {iterations_new_4}')
```

root1.py

We used both of our functions to numerically solve our equation. False-position worked in the general margin that was set by the problem $[1, 10^7]$. In Newton-Raphson, we had to pick a starting point. When we chose the middle value $5 \cdot 10^6$ the algorithm did not converge. When we instead used smaller initial values we had a fast convergence, much faster than when using the false-position.

For each of the algorithms, the predefined accuracy that had to be reached for them to converge was set to 10^{-11}. The final relative error estimates are even smaller as we can see in the output of the script.

```
1  Using False-Position
2
3  T = 32539.1267375
4  Accuracy = 2e-16
5  number of iterations = 20
6
7
8  Using Newton-Raphson
```

```
9
10 Starting from a = 2.0e+05
11 T = 32539.1267375
12 Accuracy = 2e-15
13 number of iterations = 5
14
15 Starting from a = 1.0e+04
16 T = 32539.1267375
17 Accuracy = 2e-16
18 number of iterations = 4
```

output/root1.txt

2.2 Second equation

We now want to add additional heating and cooling mechanisms and solve them. One such mechanism is the free-free emission cooling which is described by

$$\Lambda_{FF} = 0.54 \left(\frac{T}{10^4}\right)^{0.37} \alpha_B n_H n_e k_B T \tag{10}$$

The heating by cosmic rays can be calculated using

$$\Gamma_{CR} = A n_e \xi_{CR} \tag{11}$$

where $A = 5 \cdot 10^{-10}$ is a constant and $\xi_{CR} = 10^{-15}$

One last heating mechanism comes from MHD waves

$$\Gamma_{MHD} = 8.9 \cdot 10^{-26} n_H \frac{T}{10^4} \tag{12}$$

Combining all that and assuming $n_H = n_e$ (total ionization) we can derive an equation which can be solved numerically:

$$(\psi T_c - \left[0.684 - 0.0416 \ln\left(\frac{T}{10^4 Z^2}\right)\right] T - 0.54 \left(\frac{T}{10^4}\right)^{0.37}) k n_H \alpha_B + A\xi_{CR} + 8.9 \cdot 10^{-26} \frac{T}{10^4} = 0 \tag{13}$$

We will solve the above equation for three different values of $n_H = n_e$, which are $10^{-4}, 1, 10^4$. All values are again in c.g.s. units.

For the root finding, we will use again the "newton_raphson" algorithm from our previous script. We also define a new function "secant", which uses the secant algorithm for root finding. In this algorithm, we have two initial points (that can act as a bracket). The new point is calculated using: $c = b - f_b \frac{b-a}{f_b - f_a}$. We then set $a = b$ and $b = c$ and repeat the process until the relative difference between the two values reaches the desired accuracy.

```
1  import numpy as np
2  from root1 import NewtonRaphson
3
4  K=1.38e-16  # erg/K
5  AB = 2e-13  # cm^3 / s
6
7  def Equilibrium2(T: float, Z: float = .015, Tc: float = 10**4, psi: float = .929,
8                   nH: float = 10**0, A: float = 5*10**-10, xi: float = 10**-15) -> float:
9      return (psi*Tc - (0.684 - 0.0416 * np.log(T/(1e4 * Z*Z)))*T - .54 * ( T/1e4 )**.37 *
       T)*K*nH*AB + A*xi + 8.9e-26 * (T/1e4)
10
11 def F2(x: float, n: float) -> float:
12     '''
13     wrapper for equilibrium2
14     '''
15     return Equilibrium2(x, Z=.015, Tc=10**4, psi=.929, nH=n, A=5*10**-10, xi=10**-15)
16
17 def Secant(f: callable, a: float, b: float, delta: float = 1e-11, maxiter: int = 10000)
       -> tuple:
```

```
    '''
    root finding using secant algorithm
    f: function
    a: low bracket
    b: upper bracket
    delta: maximum relative accuracy
    maxiter: max number of iterations before stopping
    '''
    fa = f(a)
    fb = f(b)

    for i in range(maxiter):

        dif = ((b-a)/(fb-fa))*fb
        a = b
        b = b - dif
        fa = fb
        fb = f(b)

        if abs((b-a)/b) < delta:
            return b, abs((b-a)/b), i

    raise RuntimeError("Secant algorithm did not converge.")

if __name__ == '__main__':

    print('Using Secant Algorithm\n')

    T_0_sec, error_0_sec, iterations_0_sec = Secant(lambda x: F2(x, n=10**0),1,10**15)

    T_pos4_sec, error_pos4_sec, iterations_pos4_sec = Secant(lambda x: F2(x, n=10**4)
    ,1,10**15)

    print(f'for nH = 10^0, T = {T_0_sec:.12}, rel error = {error_0_sec:.1}, num of iter
    = {iterations_0_sec}')
    print(f'for nH = 10^4, T = {T_pos4_sec:.12}, rel error = {error_pos4_sec:.1}, num of
     iter = {iterations_pos4_sec}')

    print('\n\nUsing Newton-Raphson Algorithm\n')

    T_neg4, error_neg4, iterations_neg4 = NewtonRaphson(lambda x: F2(x, n=10**-4), 5e14,
     step=1)
    T_0, error_0, iterations_0 = NewtonRaphson(lambda x: F2(x, n=10**0), 5e14, step=1)
    T_pos4, error_pos4, iterations_pos4 = NewtonRaphson(lambda x: F2(x, n=10**4), 5e14,
    step=1)

    print(f'for nH = 10^-4, T = {T_neg4:.12}, rel error = {error_neg4:.1}, num of iter =
     {iterations_neg4}')
    print(f'for nH = 10^0, T = {T_0:.12}, rel error = {error_0:.1}, num of iter = {
    iterations_0}')
    print(f'for nH = 10^4, T = {T_pos4:.12}, rel error = {error_pos4:.1}, num of iter =
    {iterations_pos4}')
```

root2.py

In the main part of our script, we find the roots for the different values of n_H using both algorithms. The secant algorithm did not converge for $n_H = 10^{-4}$. For the other two values of n_H, it managed to find the correct value of the temperature in the least number of iterations (9 and 8). All the equations were solved in the range $[1, 10^{15}]$

The Newton-Raphson algorithm managed to solve the equation for all values of n_H. The number of iterations needed was 22 for the two higher values of n_H, while for the smallest density, it only needed 9 iterations. Note that in this method we use only one starting point which we set at the middle of our range $(5 \cdot 10^{14})$

For all these implementations we stopped when the relative accuracy reached was below 10^{-11}. As we can see in the output they achieved relative accuracies ranging from 10^{-13} to 10^{-17}

```
Using Secant Algorithm

```

```
for nH = 10^0, T = 33861.3002352, rel error = 8e-14, num of iter = 9
for nH = 10^4, T = 10525.8860197, rel error = 3e-15, num of iter = 8


Using Newton-Raphson Algorithm

for nH = 10^-4, T = 1.60647887537e+14, rel error = 5e-13, num of iter = 9
for nH = 10^0, T = 33861.3002352, rel error = 9e-17, num of iter = 22
for nH = 10^4, T = 10525.8860197, rel error = 2e-15, num of iter = 22
```

output/root2.txt

Lastly, one thing that becomes clear from the values of the temperature for the equilibrium, is that it greatly depends on the density of the hydrogen (which is equal to the electron). The low-density gas had a much higher temperature of equilibrium (10^{14}) when compared with the higher-density ones (10^4).

2.3 Timing executions

In this last part, we want to time the functions we used to solve the previous equations. For each one, we select the implementation that used the least number of iterations and we time it for 10^5 executions and take the average time.

For equation 1 we calculated the average time using the "false_position" and the "newton_raphson". For the 2nd equation, we calculate the average time using the "newton_raphson" for $n_H = 10^{-4}$ and the "secant" for the other two values of the density.

```
import os , sys
import timeit
import numpy as np

if __name__ == "__main__":

    eq1_new = timeit.timeit("T_new_4, error_new_4 ,iterations_new_4 = NewtonRaphson(F1,1e4)",
        "from root1 import NewtonRaphson,F1",number=10**5)

    eq1_fal = timeit.timeit("T_new_4, error_new_4 ,iterations_new_4 = FalsePosition(F1,1,1e7)",
        "from root1 import FalsePosition,F1",number=10**5)

    eq2_neg4 = timeit.timeit("T_neg4, error_neg4, iterations_neg4 = \
        NewtonRaphson(lambda x: F2(x, n=10**-4), 5e14, step=1)",
        "from root1 import NewtonRaphson; from root2 import F2",number=10**5)

    eq2_0 = timeit.timeit("T_0_sec, error_0_sec, iterations_0_sec = Secant(lambda x: F2(x, n=10**0),1,10**15)",
        "from root2 import Secant,F2",number=10**5)

    eq2_pos4 = timeit.timeit("T_0_sec, error_0_sec, iterations_0_sec = Secant(lambda x: F2(x, n=10**4),1,10**15)",
        "from root2 import Secant,F2",number=10**5)

    print(f'For equation 1')
    print(f'Average time for Newton-Raphson = {eq1_new/1e5:.2e}')
    print(f'Average time for False-Position = {eq1_fal/1e5:.2e}')

    print(f'\nFor equation 2')
    print(f'Average time for nH=1e-4 = {eq2_neg4/1e5:.2e}')
    print(f'Average time for nH=1    = {eq2_0/1e5:.2e}')
    print(f'Average time for nH=1e4  = {eq2_pos4/1e5:.2e}')
```

time_2.py

We can see that for all the implementations the average time is at the order of 10^{-5} s. Newton-Raphson is the fastest for equation 1 (it also used the least number of iterations), while for the different density values of equation 2, the secant algorithm is the fastest when it is able to converge, which is not the case for the low density. For this case, the best we can do is use Newton-Raphson which is relatively slower.

```
For equation 1
Average time for Newton-Raphson = 2.91e-05
Average time for False-Position = 5.41e-05

For equation 2
Average time for nH=1e-4 = 9.42e-05
Average time for nH=1    = 4.15e-05
Average time for nH=1e4  = 3.75e-05
```

output/time_2.txt