# Numerical Recipes in Astrophysics 2020
# Third homework set

Zorry Belcheva
s2418797

November 30, 2020

## 1 Exercise 1

For part a) of the first exercise, we're minimising $\chi^2$ for a model. We have the number density of satellites as a function of $x \equiv r/r_{\rm vir}$:

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

The model parameters are $a, b, c$, $\langle N_{\rm sat}\rangle$ is the mean number of satellites per halo, and $A$ is a normalisation constant. First we read the data using the Read_data class. We go through each line, taking into account some useful things about the data format: the 4th line contains the number of haloes in this mass bin, and after a hashtag we have the positions of all the satellites of a halo (if any), given in $x, \phi$ and $\theta$. We'll only use $x$ here. $\langle N_{\rm sat}\rangle$ is just the average of the number of satellites per halo (line 32). To proceed, we should bin the data. Looking at equation (1), it makes sense to bin the satellites *logarithmically* in x. We know that the maximum value of x is 5; to me, a suitable lower bound is $10^{-4}$, as it's low enough but not too low, and I'd expect most of the satellites, if not all, to reside at $r > 10^{-4} r_{\rm vir}$. Therefore I chose 20 logarithmically spaced bins between $x \in [10^{-4}, 5]$, again thinking that 20 is a suitable number to divide this x-range into. Then, we obtain the mean number of satellites per halo in each bin, $N_i$, as the counts in each bin divided by the number of haloes in this mass bin.

For the $\chi^2$ minimisation, I chose to generalise my downhill simplex method from hand-in 2 (which was in 2-dimensions) to an $n$-dimension one. For a function varying in 3 parameters, we need to give 4 initial starting points that form a tetrahedron in parameter-function space. My initial guesses are x0 to x3 (lines 248-251). The N-dimensional simplex is implemented in the `simplex_nd()` function. I've set the maximal iterations limit to 8 and the default accuracy to $10^{-3}$ for the reason that it was very slow, and I am exceeding the maximal allowed time, but ideally I'd set a better accuracy. While minimising the function, we must be careful to re-evaluate the integration constant $A$ any time we change the parameters $a, b, c$, and then also calculate the $\tilde{N}_i = 4\pi \int_{x_i}^{x_{i+1}} (z) x^2 dx$ - the model mean and variance for bin $i$. So the class Get_chi_squared() first calculates $A$ as in assignment 1 (by integrating), then finds $\tilde{N}_i$ for the bins, and gets the $\chi^2 = \Sigma (x-\mu)^2/\sigma^2$ for the set of model parameters $(a, b, c)$. Finally, we are ready to minimise the function, in this case I used the complementary function chi_minimise(point), lines 96-101.

The simplex converges after a different number of iterations for each data file, and I find slight differences in the best-fit parameters $(a, b, c)$. They are quoted in the file output, along with the average satellite number, and the minimum $\chi^2$. Plots of the binned data as well as the best fit profiles are shown after the output.

For part b), we want to minimise a Poisson log-likelihood, which, from the lecture slides, is given by:

$$\mathcal{L}(p) = \prod_{i=0}^{N-1} \frac{\mu(x_i|p)^{y_i} \exp(-\mu(x_i|p))}{y_i!}. \tag{2}$$

Taking the negative log of this expression, we have

$$-\ln \mathcal{L}(p) = -\sum_{i=0}^{N-1} (y_i \ln(\mu(x_i|p)) - \mu(x_i|p) - \ln(y_i!)). \tag{3}$$

We have $y_i = N_i$, $\mu(x_i|p) = \tilde{N}_i$. Then, out log-likelihood is:

$$-\ln \mathcal{L}(a, b, c) = -\sum_{i=0}^{N-1} (N_i \ln(\tilde{N}_i) - \tilde{N}_i - \ln(N_i!)). \tag{4}$$

We can further expand the factorial in the last term as the sum of logs of the numbers from $N_i$ to 1. We take the sum over all bins used in part a). The procedure to do the optimisation is the same as in part a), except this time we minimise the complementary function likelihood_minimise(point), which calls `Get_chi_squared().get_poisson_likelihood()`, because we still need to do the same steps and calculate $A$ and $\tilde{N}_i$ for any set of parameters we explore. Again, it takes a different amount of iterations for the simplex to converge for the 5 data files. The results are quoted immediately after the results for part a) in the output file. The plots bellow show the two fits. The code runs in an incredibly long time, and I really don't know why (also, it's twice as fast on my own laptop...). If I run it for all datafiles, unfortunately I severely exceed the time limit on my office desktop (`minstroom`), so I have to save a fraction of the output to a text file, and only run ex. 1 for the first two files. The rest of the output I read in from a text file, as well as the plots - I'm sorry for this, but I really couldn't find the bottleneck on time.

Personally I'm not entirely happy with how the plots look, I was expecting that the 2nd method yields noticeably better results, but that's not the case - the 2 methods give a similar best-fit histogram. Moreover, they don't match the data as well as I had expected. The result success also differs for the 5 mass bins, because for central haloes of higher mass we have fewer satellites. This is visible especially closer to the central, where both best-fits predict much more data should be seen. Also, my guess is the success of any optimisation algorithm depends on the initial guesses, which were harder to pick in 3D parameter space. I also think the simplex is not the numerically optimal algorithm choice, to me it appears it is pretty slow in this case, but it might be due to an error from my side. Of course, it would be best to do a Levenberg-Marquardt routine, perhaps by implementing the analytical derivatives of the model with the parameters, as these are known exactly.

For part c), we're comparing the two methods. I've implemented both a G test and a KS test. The G statistic is given by

$$G = 2 \sum O_i \ln \frac{O_i}{E_i}, \tag{5}$$

where $O$ and $E$ denote observed and expected data value. This is implemented in the function Gtest(observed, expected). The significance of the G statistic I took from the lecture slides to be equal to:

$$Q = 1 - \frac{\gamma(k/2, x/2)}{\Gamma(k/2)}, \tag{6}$$

where $\gamma$ and $\Gamma$ denote the incomplete and complete Gamma functions. This is the CDF of a $\chi^2$ distribution, and is calculated in the functon Q(k, x), where k are the degrees of freedom, and x is the G statistic. The KS test implementation can be seen in the ks_test(observed, expected) function, following the slides. It calculates the statistic D and its significance Q and returns both.

Both the statistic and significance for the G test and the KS test for the best-fits are shown in the output after the result report. Regarding the degrees of freedom: we have the 3 fixed model parameters, $a, b, c$, but we also have the fixed total satellite number and number of haloes, so the average number of satellites is also a fixed parameter. Our free degrees are the bin observations, in this case 20 numbers. Therefore, the degrees of freedom I take to be equal to $n_{\text{bins}} - n_{\text{params}} - 1 = 16$. Regarding the results: for what I see, for the G test, although the statistic varies a bit for the two models, its Q value is very close for both, and is at almost all cases equal to 1, which makes me think this particular statistic doesn't distinguish both fit

models - as I would have expected by seeing the plots. According to my eye, there's little difference between the two fits, and they're both equally 'far' from the data; the G test statistic seems to agree. Of course, here we don't comment on the actual success of my fits - perhaps the best fit parameters are just wrong. The KS statistic, on the other hand, gives different results for both models. For the 3 higher mass bins, the KS Q-value is higher for the log-likelihood minimisation, so I think we could naively say it favours this model? But it's the opposite case for the two lower mass bins. In reality, I wouldn't use either of these tests to draw a definitive conclusion.

```
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import math
from scipy.special import gammainc
import time


# Read data from datafile:
# - bins_low = lower histogram boundary;     defailt = 1e-4
# - bins_high = higher histogram boundary;   default = 5
# - nbins = number of bins to use;           default = 20
class Read_data:
    def __init__(self, datafile, bins_low=1e-4, bins_high=5, nbins=20):
        file = open(datafile, 'r')
        lines = file.readlines()
        lines = [lines[i][:-1] for i in range(len(lines))]  # remove \n at the end of each line
        file.close()

        self.nhaloes = int(lines[3])  # extract the no. of haloes - 3rd line
        self.nsats = np.zeros(self.nhaloes)  # to hold the no. of satellites of each halo

        i = 0
        # Count the number of satellites per halo:
        for l in lines[5:]:
            if l == '#':  # new halo encountered
                i += 1
            else:  # new satellite encountered: +1 to halo satellite count
                self.nsats[i] += 1

        self.Nsat = np.average(self.nsats)  # this is <Nsat>

        # read in the satellite coordinates:
        self.coords = np.genfromtxt(datafile, skip_header=4)
        # Make the histogram:
        self.bins = np.logspace(np.log10(bins_low), np.log10(bins_high), nbins)
        self.counts, self.bin_edges = np.histogram(self.coords[:, 0], bins=self.bins)
        self.Ni = self.counts / self.nhaloes    # average no. of satellites per halo in each bin
        self.Ni_poisson = []  # to hold the Poisson equivalent of Ni
        self.nbins = nbins
        self.bins_low = bins_low
        self.bins_high = bins_high
        self.A = []     # Integral normalisation: to be recomputed at every (a, b, c) evaluation
        self.chi = []


# Given the data 'data' and parameters (a, b, c),
# 1) compute normalisation A
# 2) compute Ni_poisson = tilde(Ni) - Poisson expectation counts value
class Get_chi_squared():
    def __init__(self, data, a, b, c):
        args = [a, b, c]
```

```
54          data.A = 1/(4*np.pi*self.simpsons(self.n_not_normalised, data.bins_low, data.
    bins_high, 100000, args))
55          data.Ni_poisson = [4*np.pi*self.simpsons(self.n_normalised_poisson,
56                                                   a=data.bin_edges[i], b=data.bin_edges[i+1],
     N=10000,
57                                                   args=[a, b, c, data.Nsat, data.A]) for i in
     range(data.nbins-1)]
58
59      # Simpson's rule:
60      # integrate a function f in the interval from a to b, using N points
61      def simpsons(self, func, a, b, N, args=[]):
62          h = (b - a) / N
63          s = func(a, *args) + func(b, *args)
64          for i in range(1, N, 2):
65              s += 4 * func(a + h * i, *args)
66          for i in range(2, N - 1, 2):
67              s += 2 * func(a + h * i, *args)
68          return s * h / 3
69
70      # General chi squared
71      def chi_squared(self, x, mu, sigma):
72          return sum((x - mu) * (x - mu) / sigma / sigma)
73
74      # Expression to integrate, as function of x = r/r-vir
75      def n_not_normalised(self, x, *args):
76          [a, b, c] = args
77          return x ** (a - 1) * b ** (3 - a) * np.exp(-(x / b) ** c)
78
79      def n_normalised_poisson(self, x, a, b, c, nsat, A):
80          return A*nsat*(x/b)**(a-3)*np.exp(-(x/b)**c)*x*x
81
82      def get_chi(self):
83          return self.chi_squared(data.Ni, mu=data.Ni_poisson, sigma=data.Ni_poisson)
84
85      # Poisson log-likelihood for the data observation (Ni) and expectation (Ni_poisson)
86      def get_poisson_likelihood(self):
87          n = len(data.Ni)
88          s = np.zeros(n)
89          for i in range(n):
90              s[i] = data.Ni_poisson[i] - data.Ni[i]*np.log(data.Ni_poisson[i])
91              for j in range(int(data.Ni[i]))[::-1]:  # the factorial
92                  s[i] += np.log(j + 1)
93          return sum(s)
94
95
96  # Get chi squared value at a point in parameter space
97  def chi_minimise(point):
98      a = point[0]
99      b = point[1]
100     c = point[2]
101     return Get_chi_squared(data, a, b, c).get_chi()
102
103
104 # Get log-likelihood value at a point in parameter space
105 def likelihood_minimise(point):
106     a = point[0]
107     b = point[1]
108     c = point[2]
109     return Get_chi_squared(data, a, b, c).get_poisson_likelihood()
110
111
112 # Swap function
113 def swap(a, b):
114     swap = a
115     a = b
```

```
    b = swap
    return a, b


# Simple selection sort algorithm; return sorted indices
def selection(a):
    n = len(a)
    b = np.copy(a)
    ind_array = np.arange(n)
    for i in range(n):
        m = min(b[i:])
        # print(m)
        ind = np.argmin(b[i:])
        swap = b[i]
        b[i] = m
        b[ind + i] = swap
        # swap = i
        ind_array[i] = i + ind
        # ind_array[ind+i] = swap
        # print(a)
    return b, ind_array


# N-dimensional Downhill simplex
def simplex_nd(func, x, maxiter=8, accuracy=1e-3):
    iteration = 0
    n = len(x)
    f = np.empty(n)
    for i in range(n):
        f[i] = func(x[i])
    # ---------------------- Downhill simplex implementation: ---------------------
    while iteration <= maxiter:
        # sort the points:
        for i in range(n):
            f[i] = func(x[i])
        # print(f)
        f, ind = selection(f)
        # print(f, ind)
        x = x[ind]

        xbar = np.empty(x.shape[1])
        for i in range(int(x.shape[1])):
            xbar[i] = np.average(x[:, i])

        frac_range = abs(f[-1]-f[0])/abs(f[-1]+f[0])*2

        # Follow the algorithm until accuracy is met: for more algorithm details see e.g.
    the book, the slides
        if frac_range < accuracy:
            result = xbar
            print('\nAccuracy reached in ', iteration, '  iterations.')
            break
        else:
            xtry = 2*xbar - x[-1]
            ftry = func(xtry)

            if f[0] < ftry < f[-1]:
                # print('case0')
                x[-1] = xtry
            elif ftry < f[0]:
                # print('expand')
                xexp = 2*xtry - xbar
                fexp = func(xexp)
                if fexp < ftry:
                    x[-1] = xexp
```

```
                else:
                    x[-1] = xtry
            elif ftry >= f[-1]:
                # print('contract')
                xnew = 0.5*(xbar+x[-1])
                fnew = func(xnew)
                if fnew < f[-1]:
                    x[-1] = xnew
                else:
                    x = 0.5*(xbar+x)

        iteration += 1
        print(iteration, end='')
    if int(iteration) == int(maxiter):
        print("Max iterations reached (", str(maxiter), "), best guess is: ", xbar)
    return xbar, func(xbar)


# Standard G-test implementation, expression from the slides:
def Gtest(observed, expected):
    r = observed/expected
    a = observed*np.log(r)
    a = a[~np.isnan(a)]
    G = 2*sum(a)
    return G


# Significance Q of a variable following a chi-squared distribution;
# expression from the slides.
def Q(k, x):
    p = gammainc(k/2, x/2)/math.gamma(k/2)
    return 1 - p


# Function that performs KS test: is the sample 'observed' consistent
# with 'expected'? NOTE: assumes the data is binned in THE SAME bins
# Returns KS statistic D and probability P (significance Q).
# See report and 'Numerical Recipes' for more info.
def ks_test(observed, expected):
    # Numerically optimal implementation of P, the CDF:
    def Q(z):
        if z == 0:
            return 1
        elif z < 1.18:
            v = np.exp(-np.pi*np.pi/8/z/z)
            P = np.sqrt(2*np.pi)/z*(v + v**9 + v**25)
            return 1 - P
        elif z >= 1.18:
            v = np.exp(-2*z*z)
            P = 1 - 2*(v-v**4+v**9)
            return 1-P

    n = len(observed)
    c = sum(observed)   # normalisation factor
    dist = np.array([abs(sum(observed[:i])/c-sum(expected[:i])/c) for i in range(n)])

    D = max(abs(dist))
    z = D*(np.sqrt(n) + 0.12 + 0.11/np.sqrt(n))
    return D, Q(z)


datafiles = ['satgals_m11.txt', 'satgals_m12.txt', 'satgals_m13.txt', 'satgals_m14.txt', '
    satgals_m15.txt']
for datafile in datafiles[:2]:
    beg = time.time()
```

```
    # Read data in data class:
    data = Read_data(datafile=datafile, bins_low=1e-4, bins_high=5, nbins=20)

    # Initial tetrahedron:
    x0 = np.array([1, 0.5, 1])
    x1 = np.array([1.1, 0.7, 1.5])
    x2 = np.array([0.8, 1.2, 1.2])
    x3 = np.array([1, 1.3, 1.2])

    n = 3   # number of dimensions

    x = np.vstack((x0, x1, x2, x3))
    # Minimise, starting at initial tetrahedron:
    [[a, b, c], chi_min] = simplex_nd(chi_minimise, x)
    Get_chi_squared(data, a, b, c)
    g = Gtest(observed=data.Ni, expected=data.Ni_poisson)
    # degrees of freedom k: fixed parameter number is the dimension of a point in param space (x0),
    # i.e. 3 here (a, b, c) plus the total number of satellites observed - Nsat. The number of free
    # parameters is the number of bins (each bin = observation of a ~random variable).
    k = data.nbins - len(x0) - 1
    q = Q(k=k, x=g)
    d, q_ks = ks_test(observed=data.Ni, expected=data.Ni_poisson)

    # ---------------- Report results ----------------
    print('\nResults for datafile ', datafile, ':')
    print('<Nsat> =', data.Nsat.round(2))
    print('-- Best fit parameters: --')
    print('a = \t', a.round(3))
    print('b = \t', b.round(3))
    print('c = \t', c.round(3))
    print('Min chi-squared = ', chi_min.round(3), '\n')
    print('G value =\t', g.round(5))
    print('Q value =\t', q)
    print('KS statistic D =', d.round(3))
    print('KS significance Q =', q_ks.round(3), '\n')

    bin_centres = np.array([0.5*(data.bin_edges[i+1] + data.bin_edges[i]) for i in range(len(data.bin_edges)-1)])
    bin_width = np.array([data.bin_edges[i+1]-data.bin_edges[i] for i in range(len(data.bin_edges)-1)])
    plt.figure()
    plt.loglog()
    plt.step(bin_centres, data.Ni, where='mid', c='grey', label='Data')
    plt.step(bin_centres, data.Ni_poisson, where='mid', c='royalblue', label='Min chi-squared')
    plt.xlabel('x')
    plt.ylabel('Counts N')

    [[a2, b2, c2], lmin] = simplex_nd(likelihood_minimise, x)
    Get_chi_squared(data, a2, b2, c2)
    g = Gtest(observed=data.Ni, expected=data.Ni_poisson)
    k = data.nbins - len(x0) - 1
    q = Q(k=k, x=g)
    d, q_ks = ks_test(observed=data.Ni, expected=data.Ni_poisson)
    # ---------------- Report results ----------------
    print('\n')
    print('<Nsat> =', data.Nsat.round(2))
    print('-- Best fit parameters: --')
    print('a = \t', a2.round(3))
    print('b = \t', b2.round(3))
    print('c = \t', c2.round(3))
    print('Min log-likelihood = ', lmin, '\n')
    print('G value =\t', g.round(5))
```

```
    print('Q value =\t', q)
    print('KS statistic D =', d.round(3))
    print('KS significance Q =', q_ks.round(3), '\n')

    plt.step(bin_centres, data.Ni_poisson, where='mid', c='maroon', label='Min log-likelihood')
    plt.legend(frameon=False)
    plt.title(datafile)
    plt.savefig('plots/'+datafile[:11]+'-hist.png', dpi=300)

    end = time.time()
    print(datafile, ' took ', round(end-beg), 's')
```

model–optimisation.py

```
123
Accuracy reached in  3  iterations.

Results for datafile  satgals_m11.txt :
<Nsat> = 0.01
-- Best fit parameters: --
a =      1.1
b =      1.025
c =      1.35
Min chi-squared =  7.515

G value =    0.00333
Q value =    1.0
KS statistic D = 0.107
KS significance Q = 0.973

123456789

<Nsat> = 0.01
-- Best fit parameters: --
a =      1.138
b =      0.719
c =      1.5
Min log-likelihood =  0.09626858070600115

G value =    0.00264
Q value =    1.0
KS statistic D = 0.183
KS significance Q = 0.503

satgals_m11.txt  took  255 s
123
Accuracy reached in  3  iterations.

Results for datafile  satgals_m12.txt :
<Nsat> = 0.25
-- Best fit parameters: --
a =      1.1
b =      1.025
c =      1.35
Min chi-squared =  11.506

G value =    0.11267
Q value =    1.0
KS statistic D = 0.191
KS significance Q = 0.449

123456789

<Nsat> = 0.25
```

```
-- Best fit parameters: --
a =       1.05
b =       0.675
c =       1.5
Min log-likelihood =  1.0201017283500837

G value =     0.08808
Q value =     1.0
KS statistic D = 0.226
KS significance Q = 0.249

satgals_m12.txt  took  205 s
```

output/model–optimisation.txt

```
123456789Results for datafile  satgals_m13.txt :
<Nsat> = 4.37
-- Best fit parameters: --
a =       1.056
b =       0.962
c =       1.369
Min chi-squared =  10.388

G value =     1.72753
Q value =     0.9999999992897227
KS statistic D = 0.198
KS significance Q = 0.403

12345
Accuracy reached in  5   iterations.


<Nsat> = 4.37
-- Best fit parameters: --
a =       1.15
b =       0.725
c =       1.5
Min log-likelihood =  5.251561527060906

G value =     0.75286
Q value =     0.9999999999985792
KS statistic D = 0.097
KS significance Q = 0.991

satgals_m13.txt  took  91 s
123456789Results for datafile  satgals_m14.txt :
<Nsat> = 29.13
-- Best fit parameters: --
a =       1.05
b =       1.0
c =       1.35
Min chi-squared =  14.08

G value =     19.89509
Q value =     0.9998462268152944
KS statistic D = 0.271
KS significance Q = 0.101

123
Accuracy reached in  3   iterations.


<Nsat> = 29.13
-- Best fit parameters: --
a =       1.1
```

```
b =       0.7
c =       1.5
Min log-likelihood =  11.363200229855918

G value =     10.00423
Q value =     0.9999734934673792
KS statistic D = 0.136
KS significance Q = 0.846

satgals_m14.txt  took  79 s
123456789Results for datafile  satgals_m15.txt :
<Nsat> = 329.5
-- Best fit parameters: --
a =       1.091
b =       0.716
c =       1.491
Min chi-squared =  13.71

G value =     107.77364
Q value =     0.9998015873015873
KS statistic D = 0.2
KS significance Q = 0.393

123456789

<Nsat> = 329.5
-- Best fit parameters: --
a =       1.159
b =       0.73
c =       1.5
Min log-likelihood =  52.64625027999234

G value =     86.73297
Q value =     0.9998015873015893
KS statistic D = 0.167
KS significance Q = 0.626

satgals_m15.txt  took  109 s
```

output/1-missing.txt

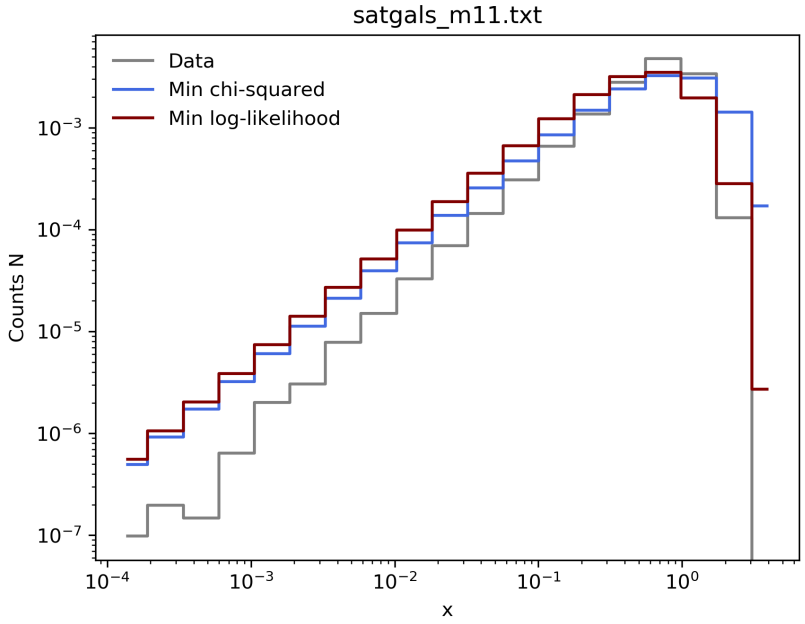


Figure 1: Best fits for the two models and data in the corresponding mass bin.

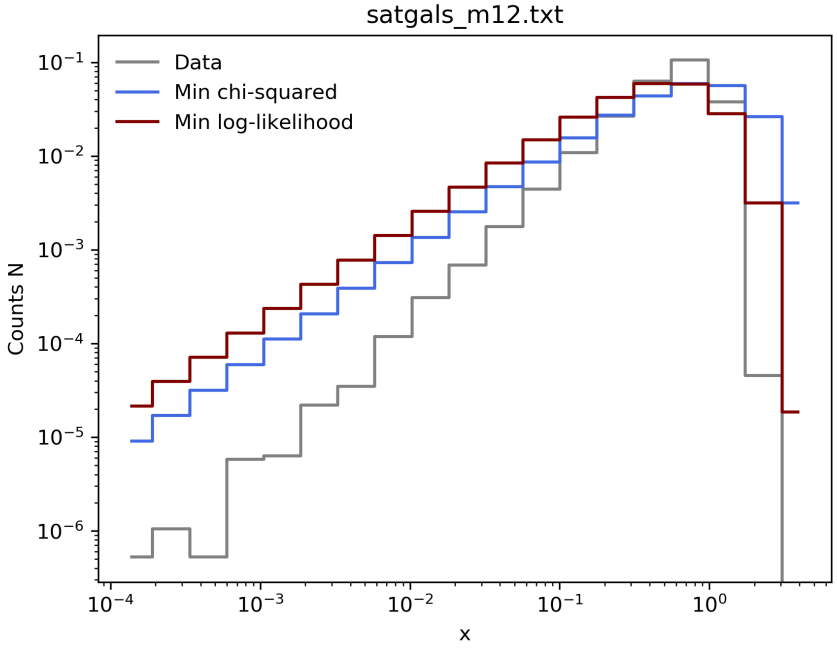


Figure 2: Best fits for the two models and data in the corresponding mass bin.

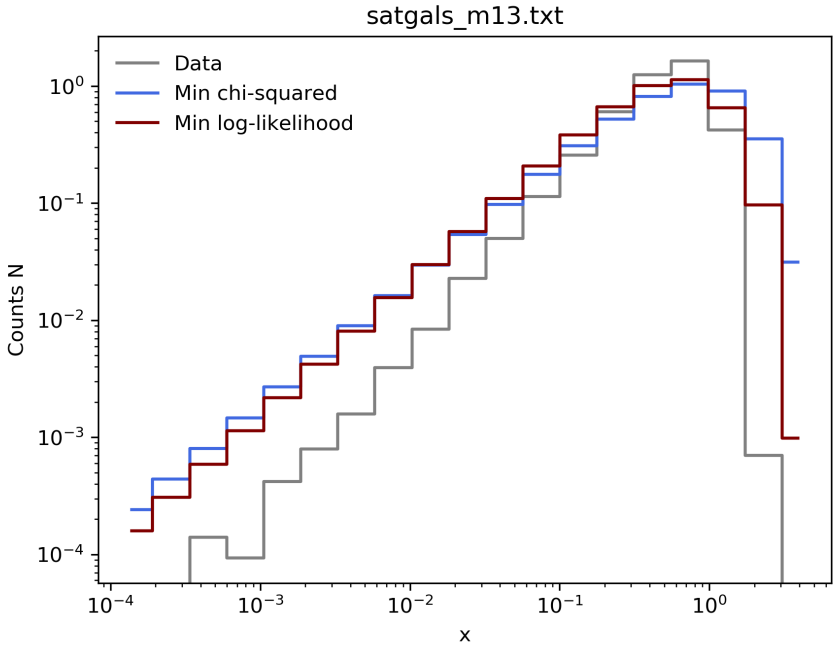


Figure 3: Best fits for the two models and data in the corresponding mass bin.

## 2 Exercise 2

To start the second exercise, aiming to calculate forces using an FFT, we first need to initialise our simulation box. It is a cubic grid with $16^3$ grid points. I define a separate class for a particle and a point, with the latter being a *grid* point, i.e. at a quantised position, whereas a particle can be initialised anywhere. There is also a Cell class, defining a cell starting at coordinates (x, y, z) and having a given width. Mass can also be assigned to cells and grid points; the latter also have a delta attribute to hold the density contrast/overdensity. The Grid class initialises a grid with a given dimension and cell width, then calls a function to initialise the cells, as well as the grid points. The add_particle() method appends an object of Particle class to the grid. In the main function, we first initialise the grid. Then, we add 1024 particles with coordinates given by the `positions` array as described in the exercise (fixing the numpy random seed).

We then proceed to the Cloud-In-Cell (CIC) implementation - method `assign_masses_cic()`. We loop through all the particles, and then through all grid points, and find the 8 nearest grid points to the given particle, using periodic boundary conditions. Finally, we calculate the mean density of the grid $\bar{\rho} = m/V$ i.e. total mass divided by total volume[1], as well as the density contrast $\delta = (\rho - \bar{\rho})/\bar{\rho}$ - an attribute of each grid point. We call the CIC method in the main, and to check for conservation of mass, we can see the sum of masses assigned to all grid points - if the method has assigned the correct weights to the points, and periodic boundary conditions are satisfied, this number should be equal to the total mass in particles, i.e. the number of particles each of mass unity. In the output of the script we can see that the total mass assigned is indeed 1024 with a small numerical error.

Finally, we create plots of the density contrast for the requested grid slices. The plots are shown after the code output.

For part b), we need to solve the spatial dependence of the Poisson equation: $\nabla^2\Phi \propto \delta$. Fourier

[1]The expression in the method assumes the particles are of mass 1 each.

transforming this proportionality, we have

$$k^2\tilde{\Phi} \propto \tilde{\delta} \quad \Rightarrow \quad \tilde{\Phi} \propto \frac{\tilde{\delta}}{k^2}, \tag{7}$$

where $k$ is the wavevector, given by $k^2 = k_x^2 + k_y^2 + k_z^2$. Inverse Fourier transforming the last proportionality, we have that the potential is proportional to the inverse FT of the overdensity divided by the wavevector squared:

$$\Phi \propto \mathcal{F}^{-1}\left(\frac{\tilde{\delta}}{k^2}\right). \tag{8}$$

Therefore, to get the potential, we need to Fourier transform the overdensity, divide by $k^2$, then inverse FT the product. I start with a discrete Fourier transform, see dft(x). The inverse equivalent is idft(x). Then, FFT the implementation that finally worked for me is shown in the fft(x) function. It is a recursive version of the Cooley-Tukey algorithm, dividing the array elements into even and odd until we have only pairs of elements; on the latter, we do a DFT. Then, we do a 'butterfly swap' of the elements at the corresponding indices.[2] This Fourier transforms a 1D array. The inverse equivalent is ifft(x). However, we need a 3D FFT, which is just a sequence of 1D FFTs. This is implemented in fft_3d(x). First we loop over the the first, then, the 2nd, and finally the third dimension of x. The inverse equivalent is ifft_3d(x). In the main, we do a FT of the overdensity $\delta$. I also check whether the result is the same as the numpy result, and they match, as seen in the output. Then, I create the $k^2$ vector, looping through the indices. The FT of the potential is then the FT of the overdensity over $k^2$. Finally, we have to inverse FT this (in 3D). The script finishes with

[2]In order to finally write a working implementation of the FFT, I had to look at a few sources, notably this link and this link, because my first six attempts of different Cooley-Tukey versions, following the slides or book, both recursive and iterative, were unfortunately unsuccessful... Looking at these sources helped me understand what to do better, but I did my best to write *my own* version of the algorithm.

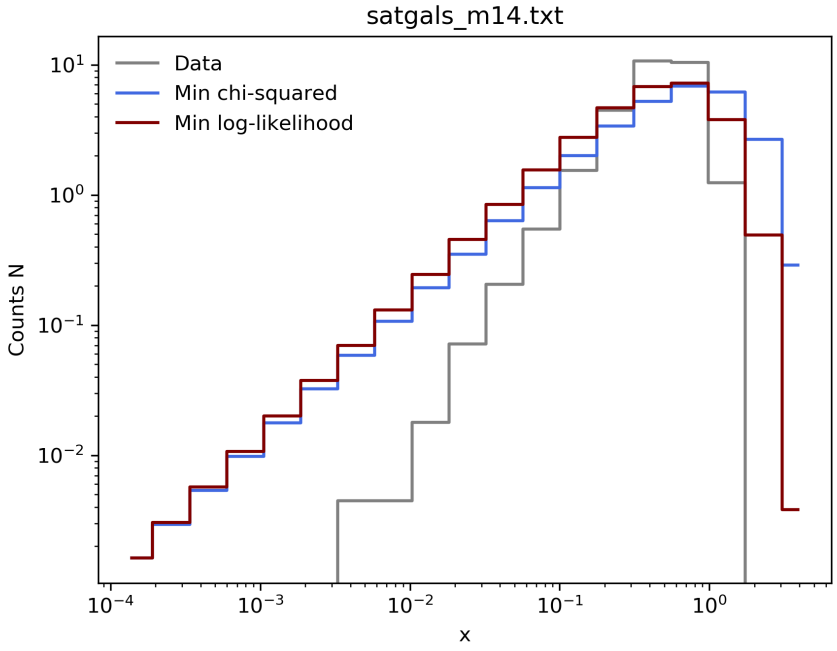


Figure 4: Best fits for the two models and data in the corresponding mass bin.

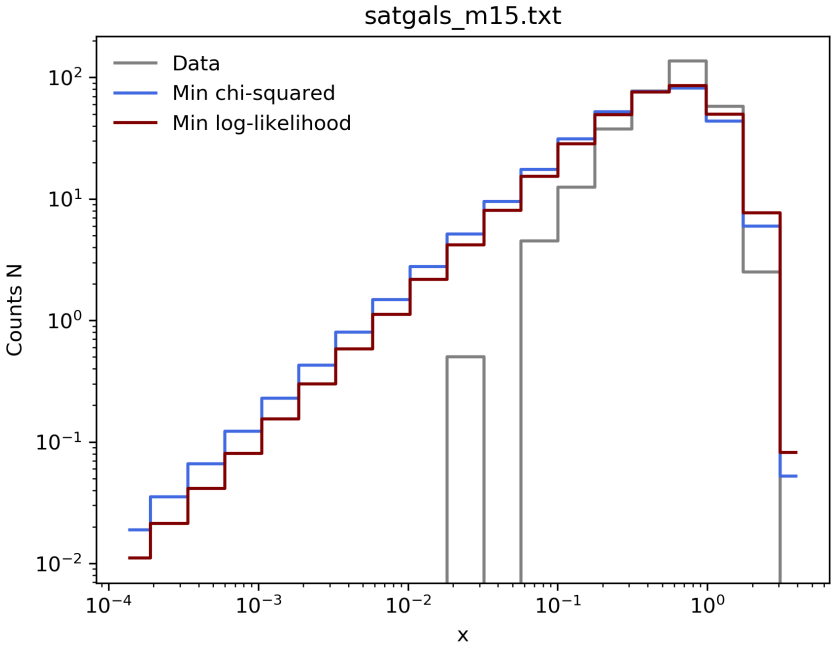


Figure 5: Best fits for the two models and data in the corresponding mass bin.

plotting of the Fourier-transformed potential (log of the absolute value, as it is a complex number, and we expect it to have significant both real and imaginary parts), and the calculated (inverse-fourier-transformed) potential $\Phi$ at the requested grid slices. The plots are shown after the code output. For some reason, the IFT doesn't match the numpy equivalent, but I couldn't trace my error down. Comparing the slices I obtain to the ones of a classmate, though, as well as to the numpy result, I see that the overall structure is still recovered pretty well. The potential value (colorbar) has a different range, though, but after all we didn't care about the normalisation here. Still, I know the result is not entirely correct.

```
#! / us r / bin /env python3
import numpy as np
import matplotlib.pyplot as plt
import math
import time

beg = time.time()


# Particle - has coordinates x, y, z as attributes.
# Default particle mass = 1.
class Particle:
    def __init__(self, x, y, z, mass=1):
        self.x = x
        self.y = y
        self.z = z
        self.mass = mass


# Point - has coordinates, assigned mass, and overdensity as attributes.
# Points form the *grid*, i.e. their positions are quantised. This is handled
# in the grid class. Particles on the other hand can, in principle, be anywhere.
class Point:
```

```
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        self.mass = 0
        self.delta = 0


# A cell is defined by its boundaries, i.e. initial (x,y,z) and width;
# cells have a mass attribute to store total mass in cell.
class Cell:
    def __init__(self, index, x, y, z, width, mass=0):
        self.index = index
        self.x_low = x
        self.x_high = x+width
        self.y_low = y
        self.y_high = y+width
        self.z_low = z
        self.z_high = z+width
        self.mass = mass


# A grid with dim = dimension, defined cell_width. Init method initialises cells in
# the requested grid, as well as the grid points.
class Grid:
    def __init__(self, dimension, cell_width, cells=[], particles=[]):
        self.dimension = dimension
        self.cells = cells
        self.particles = particles
        self.cell_width = cell_width
        self.n_particles = len(self.particles)
        self.mean_density = 0

        self.initialise_cells()
        self.initialise_grid_points()
        self.grid_points = np.array(self.grid_points)

    def initialise_cells(self):
        ind = 0

        for i in range(self.dimension):
            for j in range(self.dimension):
                for k in range(self.dimension):
                    self.cells.append(Cell(ind, i+0.5, j+0.5, k+0.5, width=self.cell_width))
                    ind += 1

    def initialise_grid_points(self):
        self.grid_points = []

        # Not ideal to use 3 nested loops, but for such low dimension of
        # the problem this is not too bad.
        for i in range(int(self.dimension)):
            for j in range(int(self.dimension)):
                for k in range(int(self.dimension)):
                    self.grid_points.append(Point(i+0.5, j+0.5, k+0.5))

    # add particle at (x, y, z) to the grid:
    def add_particle(self, x, y, z, mass=1):
        self.particles.append(Particle(x, y, z, mass))
        self.n_particles = len(self.particles)

    # Cloud-in-cell method: assign masses to the grid points by weighing the mass of each
    # particle based on the distance to its nearest 8 grid points. Total sum of weights = 1.
    def assign_masses_cic(self):
        ngridpoints = len(self.grid_points)
```

```
89
90          # Loop for all particles:
91          for ind in range(self.n_particles):
92              [x, y, z] = [self.particles[ind].x, self.particles[ind].y, self.particles[ind].z
    ]
93
94              for ind_point in range(ngridpoints):
95                  xp = self.grid_points[ind_point].x
96                  yp = self.grid_points[ind_point].y
97                  zp = self.grid_points[ind_point].z
98                  # Periodic boundary conditions:
99                  if abs(x-xp) <= self.cell_width or abs(x-xp-self.dimension) <= self.
    cell_width or abs(x-xp+self.dimension) <= self.cell_width:
100                     if abs(y - yp) <= self.cell_width or abs(y-yp-self.dimension) <= self.
    cell_width or abs(y-yp+self.dimension) <= self.cell_width:
101                         if abs(z - zp) <= self.cell_width or abs(z-zp-self.dimension) <=
    self.cell_width or abs(z-zp+self.dimension) <= self.cell_width:
102                             # print('Closeby point: ', xp, yp, zp)
103                             dx = min(abs(x-xp), abs(x-xp-self.dimension), abs(x-xp+self.
    dimension))
104                             dy = min(abs(y-yp), abs(y-yp-self.dimension), abs(y-yp+self.
    dimension))
105                             dz = min(abs(z-zp), abs(z-zp-self.dimension), abs(z-zp+self.
    dimension))
106                             # print(dx, dy, dz)
107                             self.grid_points[ind_point].mass += (1-dx)*(1-dy)*(1-dz)
108         self.calculate_mean_density()
109         self.calculate_density_contrast()
110
111     # mean density = \bar(\rho) = total mass in particles divided by the grid volume
112     def calculate_mean_density(self):
113         self.mean_density = len(self.particles)/self.dimension**3
114
115     # density contrast = \delta = (rho - \bar(rho))/\bar(rho):
116     # calculate the local density at each grid point, then translate to delta
117     def calculate_density_contrast(self):
118         ngridpoints = len(self.grid_points)
119         for i in range(ngridpoints):
120             rho = self.grid_points[i].mass/self.cell_width**3
121             self.grid_points[i].delta = (rho-self.mean_density)/self.mean_density
122
123
124 ndim = 3
125 nparticles = 1024
126 ngrid = 16
127
128 # Initialise the particle positions:
129 np.random.seed(121)
130 positions = np.random.uniform(low=0, high=16, size=(3, 1024))
131 x = positions[0]
132 y = positions[1]
133 z = positions[2]
134
135 # Intialise grid with cell width of unity:
136 grid = Grid(dimension=ngrid, cell_width=1)
137
138 # Add all particles to the grid, and assign masses to the grid points:
139 [grid.add_particle(x[i], y[i], z[i]) for i in range(len(x))]
140 grid.assign_masses_cic()
141 masses = [grid.grid_points[i].mass for i in range(16**3)]
142
143 print('Conservation of mass check:')
144 print('No. of particles of mass 1 = ', nparticles)
145 print('Total mass assigned to the grid points = ', sum(masses))
146
```

```
147
148 grid_x = np.array([grid.grid_points[i].x for i in range(16**3)])
149 grid_y = np.array([grid.grid_points[i].y for i in range(16**3)])
150 grid_z = np.array([grid.grid_points[i].z for i in range(16**3)])
151 grid_delta = np.array([grid.grid_points[i].delta for i in range(16**3)])
152
153 for z in [4.5, 9.5, 11.5, 14.5]:
154     # find the grid points close to the desired slice: (of course, this is just 1 example
      way to do it)
155     ind = np.isclose(grid_z, z, atol=0.05)
156     plt.figure()
157     plt.scatter(grid_x[ind], grid_y[ind], c=grid_delta[ind], marker='s', s=170)
158     cbar = plt.colorbar()
159     cbar.set_label('Density contrast $\delta$')
160     plt.xlabel('x')
161     plt.ylabel('y')
162     plt.title('Grid slice at z='+str(z))
163     plt.savefig('plots/grid-'+str(math.floor(z))+'.png', dpi=300)
164     plt.close()
165
166
167 # Simple swap function:
168 def swap(a, b):
169     temp = a
170     a = b
171     b = temp
172     return a, b
173
174
175 # Shuffle the array x by reversing the indices of its elements (in binary)
176 def bit_reversal_shuffle(x):
177     n = len(x)
178     # nbits = int(math.log(n, 2))
179     for i in range(n):
180         i_bitwise = '{0:011b}'.format(i)    # some python magic - use format function to get
       binary representation
181         i_bitwise_reversed = i_bitwise[::-1]
182         j = int(i_bitwise_reversed, 2)      # use int function to convert binary -> decimal
183         # print(i, '->', j)
184         x[i], x[j] = swap(x[i], x[j])
185     return x
186
187
188 # Discrete Fourier transform:
189 def dft(x):
190     N = len(x)
191     n = np.arange(N)
192     k = np.arange(N)
193     H = []
194     for i in range(N):
195         factor = np.exp(-2j*np.pi*k[i]*n/N)
196         H.append(sum(x*factor))
197     return H
198
199
200 # Discrete Fourier transform:
201 def idft(x):
202     N = len(x)
203     n = np.arange(N)
204     k = np.arange(N)
205     H = []
206     for i in range(N):
207         factor = np.exp(2j*np.pi*k[i]*n/N)
208         H.append(sum(x*factor))
209     return np.array(H).astype(complex)/2
```

```
210
211
212 # Supposedly working version of a Fast Fourier Transform:
213 def fft(x):
214     x = x.astype(complex)
215     N = len(x)
216     a = np.zeros(N).astype(complex)  # to hold the FT during recursive steps
217     k = np.arange(N)
218
219     if N == 2:  # do DFT on 2-element pairs:
220         # print('do dft')
221         return dft(x)
222     else:
223         even = fft(x[::2])
224         odd = fft(x[1::2])
225         w = np.exp(-2j*np.pi*k/N)
226
227         # do butterfly 'swap'
228         for i in range(len(even)):
229             a[i] = even[i] + w[i]*odd[i]
230             a[i+len(even)] = even[i] + w[i+len(even)]*odd[i]
231
232         return a
233
234
235 # (?) working version of an Inverse Fast Fourier Transform:
236 def ifft(x):
237     x = x.astype(complex)
238     N = len(x)
239     a = np.zeros(N).astype(complex)  # to hold the FT during recursive steps
240     k = np.arange(N)
241
242     if N == 2:  # do DFT on 2-element pairs:
243         # print('do dft')
244         return idft(x)
245     else:
246         even = fft(x[::2])
247         odd = fft(x[1::2])
248         w = np.exp(2j*np.pi*k/N)
249
250         # do butterfly 'swap'
251         for i in range(len(even)):
252             a[i] = even[i] + w[i]*odd[i]
253             a[i+len(even)] = even[i] + w[i+len(even)]*odd[i]
254
255         return a
256
257
258 # 3-dimensional FFT;
259 def fft_3d(x):
260     n = len(x)
261     ft = np.empty(x.shape).astype(complex)
262     row, column, height = x.shape
263     # loop over rows, columns, and then vertically
264     for r in range(row):
265         for c in range(column):
266             ft[r, c, :] = fft(x[r, c, :])
267     for c in range(column):
268         for h in range(height):
269             ft[:, c, h] = fft(ft[:, c, h])
270     for r in range(row):
271         for h in range(height):
272             ft[r, :, h] = fft(ft[r, :, h])
273
274     # ft = [fft_2d(x[i]) for i in range(n)]
```

```
    # ft = np.array(ft).astype(complex)
    return ft


# 3-dimensional IFFT;
def ifft_3d(x):
    n = len(x)
    ft = np.empty(x.shape).astype(complex)
    row, column, height = x.shape
    # loop over vertical dimension, rows, and then columns; order doesn't matter, though
    for r in range(row):
        for c in range(column):
            ft[r, c, :] = ifft(x[r, c, :])
    for c in range(column):
        for h in range(height):
            ft[:, c, h] = ifft(ft[:, c, h])
    for r in range(row):
        for h in range(height):
            ft[r, :, h] = ifft(ft[r, :, h])
    return ft/n     # divide by n - IFT convention


delta_ft = np.array(fft_3d(np.array(grid_delta).reshape((16, 16, 16))))
print('Check: does the FT implementation match the numpy one:')
print(np.allclose(delta_ft, np.fft.fftn(np.array(grid_delta).reshape((16, 16, 16)))))

# k-vector initialisation:
k_squared = np.zeros((16, 16, 16))
for i in range(16):
    for j in range(16):
        for k in range(16):
            k_squared[i, j, k] = i*i + j*j + k*k

k_squared[0, 0, 0] = 1  # k(0, 0, 0) is just normalisation, so we can set it to one

phi_ft = delta_ft/k_squared     # FT of potential
phi = np.array(ifft_3d(phi_ft))  # potential = IFT of FT of potential
print('Does the IFT match?')
print(np.allclose(phi, np.fft.ifftn(phi_ft)))

# Plot the slices:
for z in [4.5, 9.5, 11.5, 14.5]:
    slice = math.floor(z)
    plt.figure()
    plt.imshow(np.log10(abs(phi_ft[:, :, slice])))
    cbar = plt.colorbar()
    cbar.set_label('log$_{10}|\widetilde{\Phi}|$')
    plt.xlabel('$k_x$')
    plt.ylabel('$k_y$')
    plt.title('Grid slice at z='+str(z))
    plt.savefig('plots/grid-phi-ft-'+str(math.floor(z))+'.png', dpi=300)
    plt.close()

    plt.figure()
    plt.imshow(abs(phi[:, :, slice]))
    cbar = plt.colorbar()
    cbar.set_label('$|\Phi|$')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Grid slice at z='+str(z))
    plt.savefig('plots/grid-phi-'+str(math.floor(z))+'.png', dpi=300)
    plt.close()

end = time.time()
print('Ex. 2 took ', round(end-beg), 's')
```

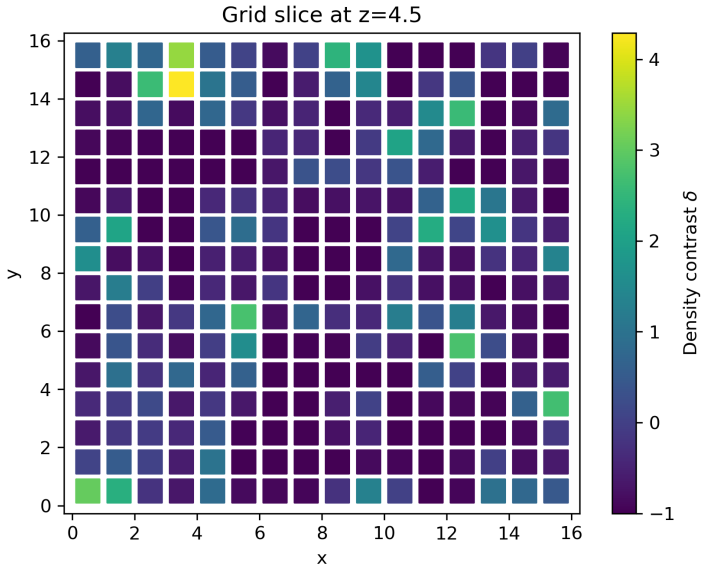


Figure 6: Slice of grid at z=4.5.

cic–fft.py

```
Conservation of mass check:
No. of particles of mass 1 =  1024
Total mass assigned to the grid points =  1024.0000000000023
Check: does the FT implementation match the numpy one:
True
Does the IFT match?
False
Ex. 2 took  28 s
```

output/cic–fft.txt

## Acknowledgements

To solve these exercises, I have again reused and reconstructed parts of the code of my own homework submission for the NUR course in 2019, notably the KS test and the CIC grid. To understand the Fast Fourier Transform, I looked at a few sources online (this link and this link). The rest of the algorithms I've mainly implemented following the lecture slides. Again, I'd like to thank my classmate Gijs Vermariën for the useful discussions.

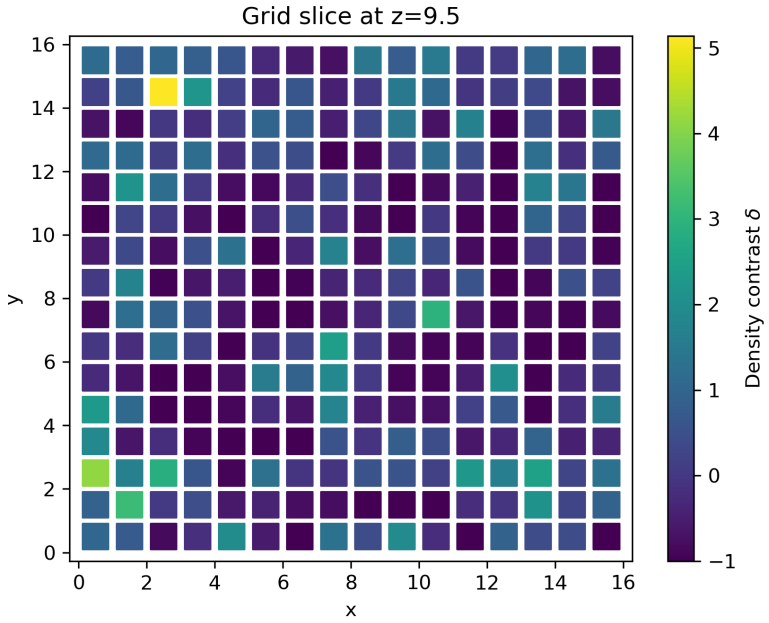


Figure 7: Slice of grid at z=9.5.

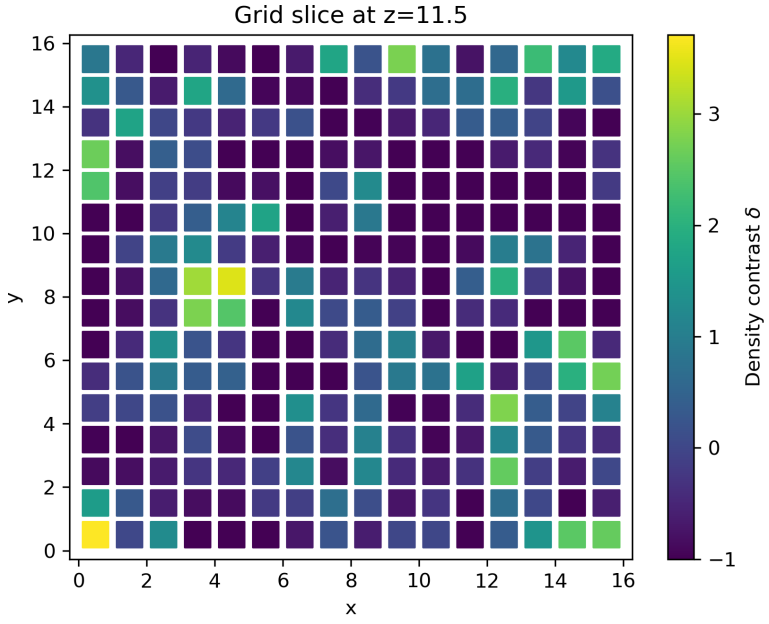


Figure 8: Slice of grid at z=11.5.

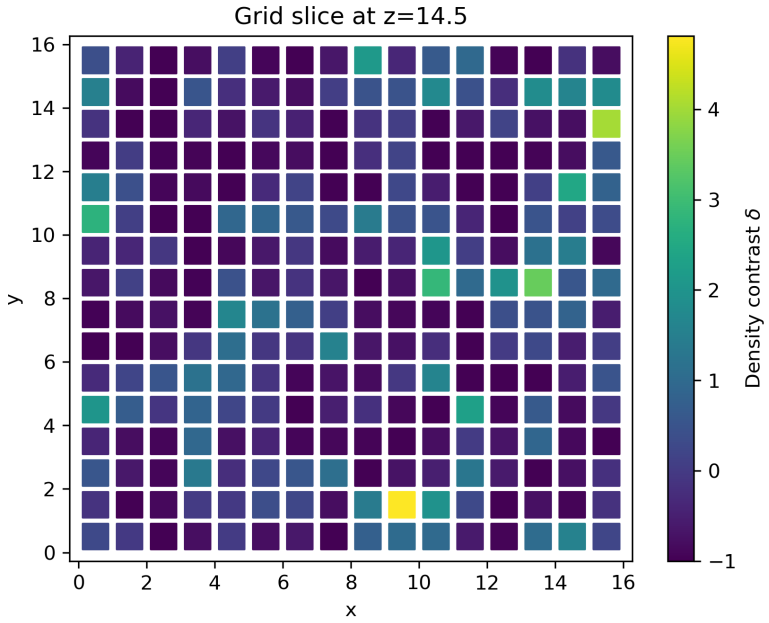


Figure 9: Slice of grid at z=14.5.

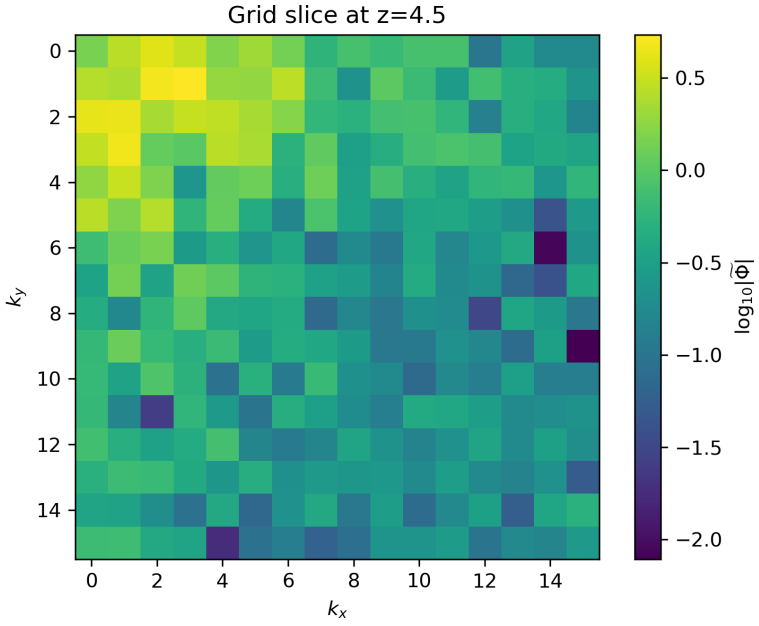


Figure 10: Slice of FR of density contrast divided by $k^2$ at z=4.5.

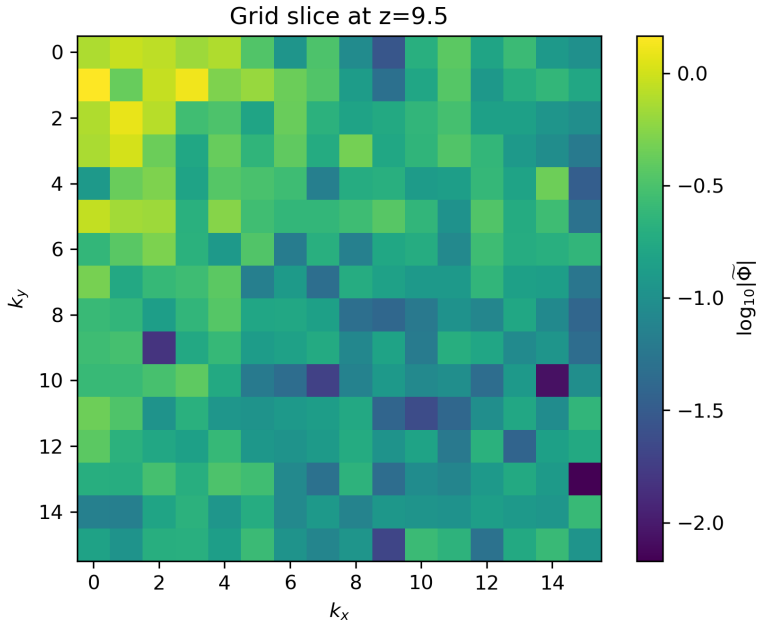


Figure 11: Slice of FR of density contrast divided by $k^2$ at z=9.5.

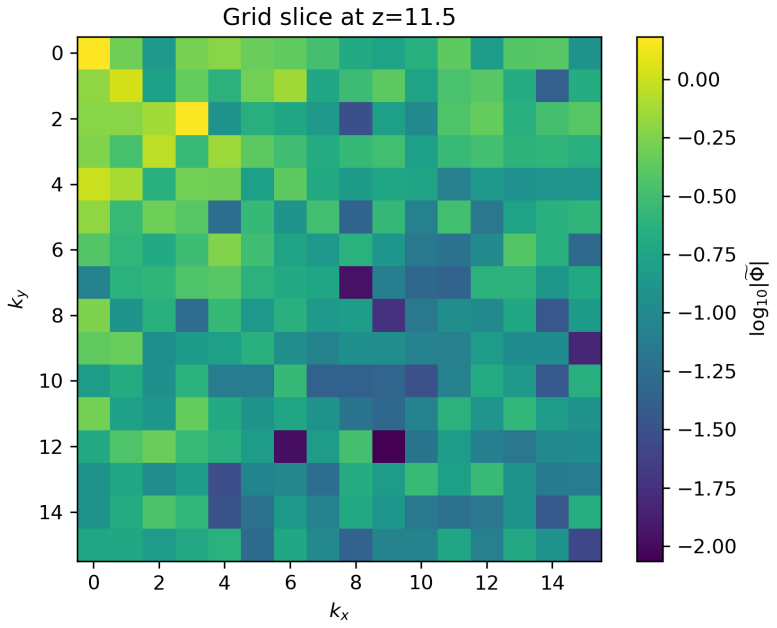


Figure 12: Slice of FR of density contrast divided by $k^2$ at z=11.5.

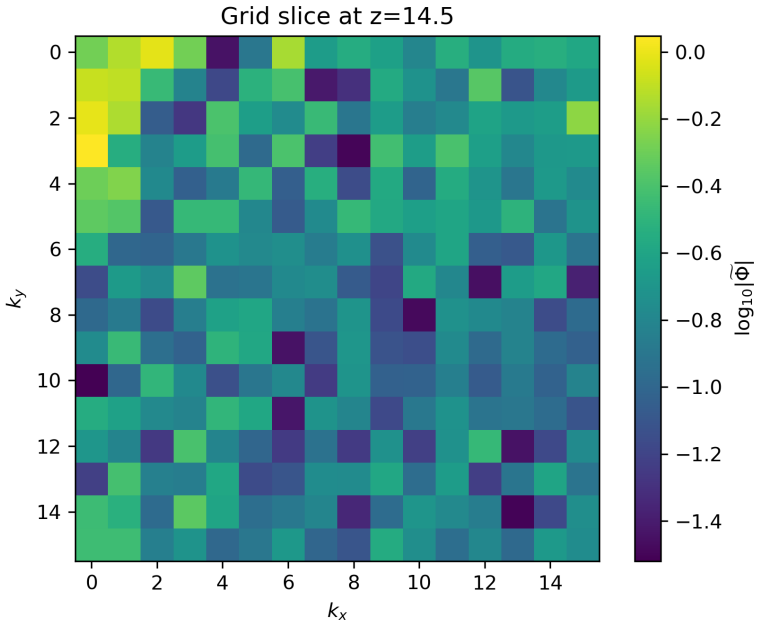


Figure 13: Slice of FR of density contrast divided by $k^2$ at z=14.5.

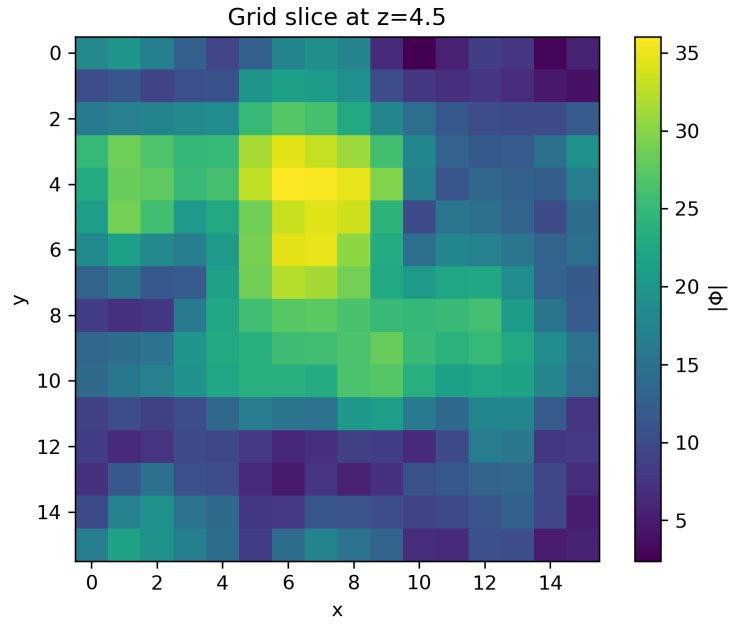


Figure 14: Slice of obtained potential at z=4.5.

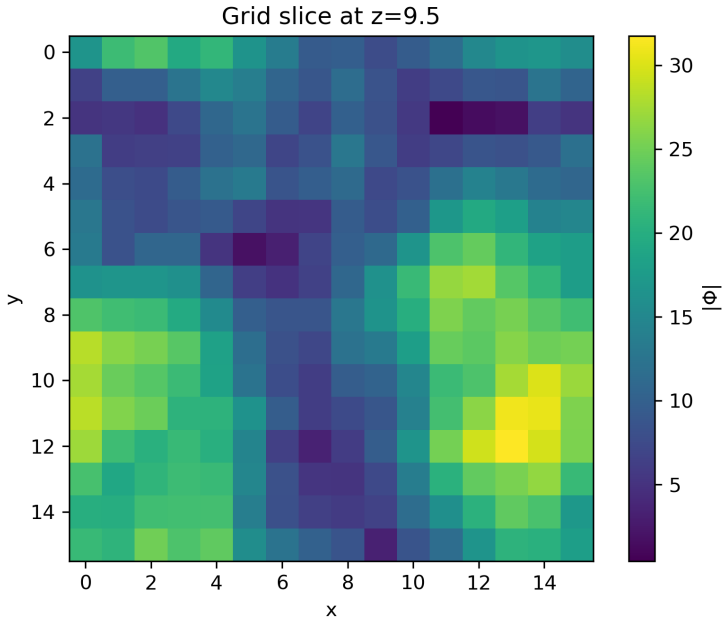


Figure 15: Slice of obtained potential at z=9.5.

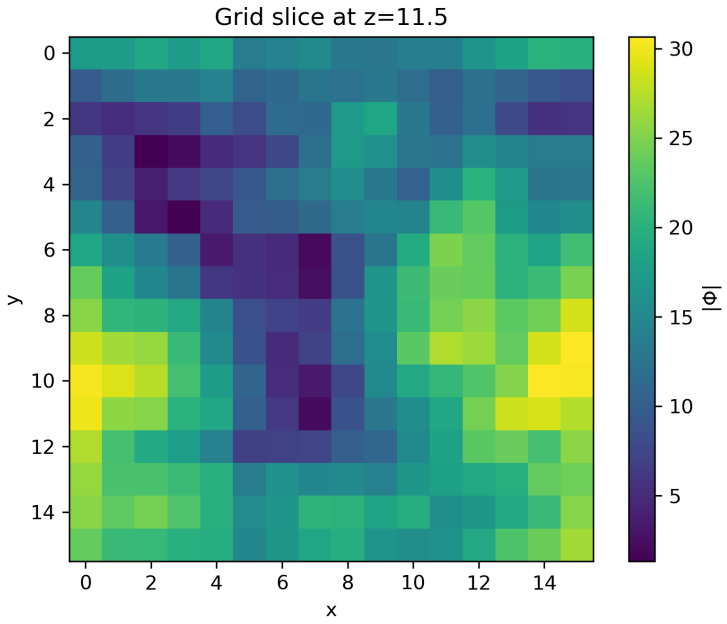


Figure 16: Slice of obtained potential at z=11.5.

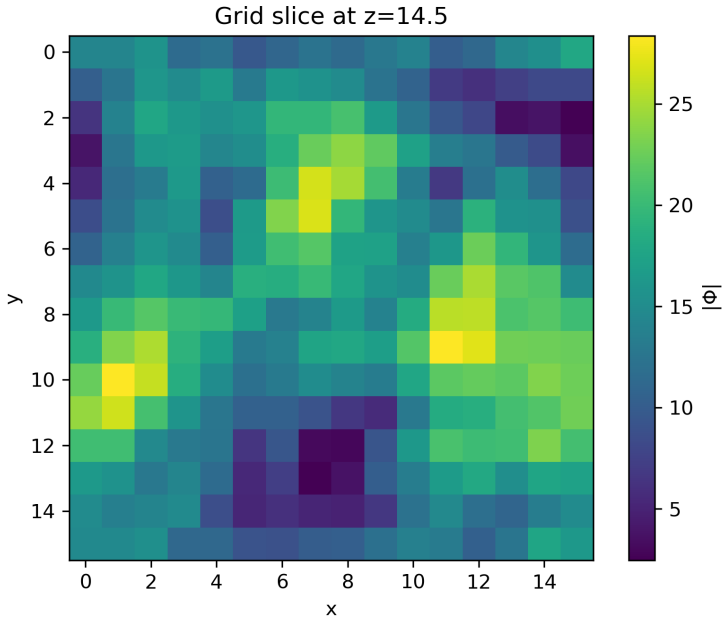


Figure 17: Slice of obtained potential at z=14.5.