Exercise 1

The RNG used is a combination of an MWC (Multiply With Carry) and an XORshift.

```
class XORshift:
    def __init__(self, seed):
        np.seterr(over='ignore')
        self._state1 = np.uint64(seed)
        self._state2 = np.uint64(seed)
        self._a = np.uint32(4294957665)
        self._high32 = np.uint32(0xffffffff)
        # Some shift constants, initialized here to avoid type casting
        # for every invocation of the RNG
        self._shift1 = np.uint64(17)
        self._shift2 = np.uint64(31)
        self._shift3 = np.uint64(8)
        self._4byte = np.uint64(32)
        if seed == 0:
            raise ValueError('Seed should not be 0!')

    def _generate(self):
        x = self._state1
        y = self._state2
        # We need to convert the shift amounts to uint64 because of a numpy bug
        # See https://github.com/numpy/numpy/issues/2524
        x ^= x >> self._shift1
        x ^= x << self._shift2
        x ^= x >> self._shift3
        y = self._a * (y & self._high32) + (y >> self._4byte)
        self._state1 = x
        self._state2 = y
        return x ^ y

    def __call__(self, n=None, uniform=True):
        if n is None:
            val = self._generate()
        else:
            val = [self._generate() for _ in range(n)]
            val = np.array(val)
        if uniform:
            val = val / (2**64 - 1)
        return val
```

1a To test the quality of the RNG, we generate random numbers from a uniform distribution $U(0, 1)$ and plot their distribution. Also shown is the dependence of a generated number on the number generated previously.

```
seed = 138
print(f'Seed: {seed}')
```

```
generator = NUR_random.XORshift(seed)

first_1000 = generator(1000)

plt.scatter(first_1000[:-1], first_1000[1:])
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel(r'$x_i$')
plt.ylabel(r'$x_{i+1}$')
plt.savefig('plots/ex1_scatter_sequential.pdf')
plt.close()

plt.scatter(range(1000), first_1000)
plt.xlim(0, 1000)
plt.ylim(0, 1)
plt.xlabel(r'$i$')
plt.ylabel(r'$x_i$')
plt.savefig('plots/ex1_scatter_index.pdf')
plt.close()

bins = np.linspace(0, 1, num=21, endpoint=True)
plt.hist(generator(int(1e6)), bins=bins)
plt.xlabel(r'$x$')
plt.ylabel('Counts')
plt.savefig('plots/ex1_hist_uniform.pdf')
plt.close()
```

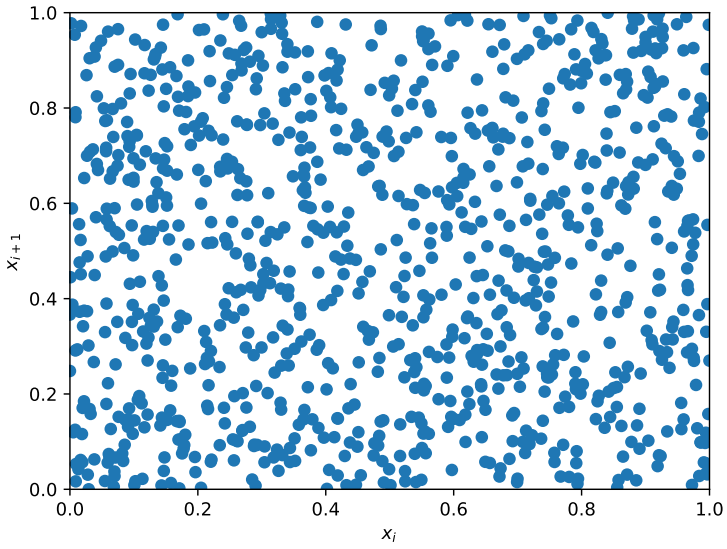

Figure 1: Random number generated against random number generated immediately before. There is no clear correlation visible, which means our RNG is working as expected.

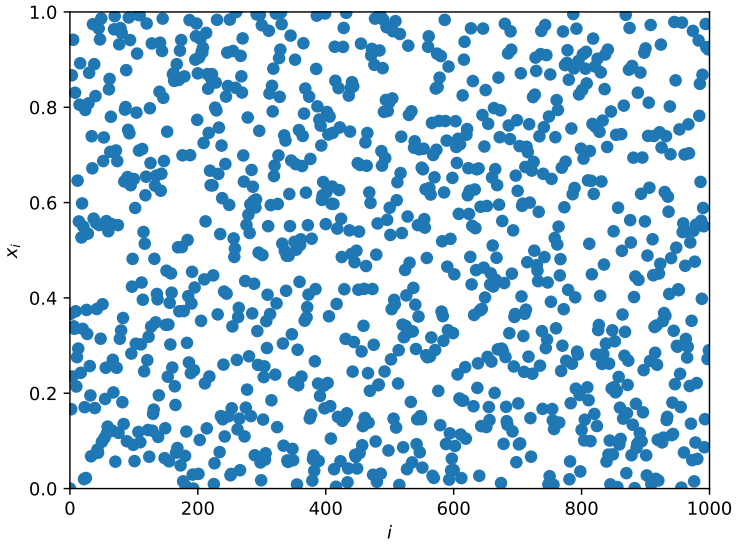

Figure 2: First 1000 random numbers generated. There is no clear trend with index, which is as expected.

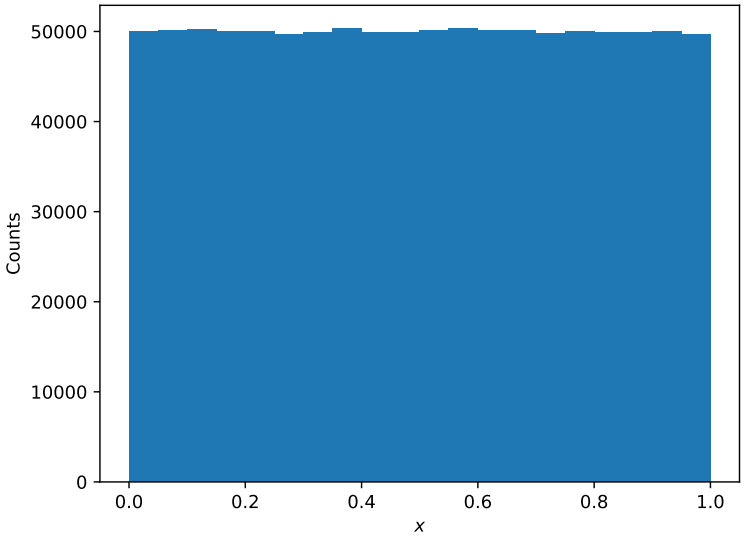

Figure 3: Histogram of 1 million uniform deviates. The numbers seem to be distributed uniformly within the range.

1b To generate standard normal deviates, we can use the Box-Muller method. To generate numbers with a known mean μ and variance σ^2, we transform these numbers by multiplying by σ and adding μ. We generate 1000 numbers with $\mu = 3$ and $\sigma = 2.4$. For 1000 normal deviates, we expect about 3 numbers to lie beyond $\pm 3\sigma$. We bin in intervals of 0.5σ.

```
def Box_Muller(N, generator):
    # Generates N standard normal deviates
    # N & 1 is equivalent to but faster than N % 2
    # Box Muller method generates even number of deviates
    if N & 1 == 0:
        randoms = generator(N)
    else:
        randoms = generator(N + 1)
    normals = np.empty_like(randoms)
    even_randoms = randoms[::2]
    odd_randoms = randoms[1::2]

    rs = (-2 * np.log(even_randoms))**0.5
    normals[::2] = rs * np.cos(2 * np.pi * odd_randoms)
    normals[1::2] = rs * np.sin(2 * np.pi * odd_randoms)
    # Make sure we return an odd number of deviates if needed
    return normals[:N]

# To go from a standard normal to a general normal:
# Multiply by std, add mean
mean = 3
std = 2.4
normals = NUR_random.Box_Muller(1000, generator) * 2.4 + 3

# Plot the histogram
bins = np.linspace(mean - 5*std, mean+5*std, num=21)
plt.hist(normals, bins=bins, density=True, label='Sampled probability density')

# Plot the theoretical distribution
xs = np.linspace(bins[0], bins[-1], num=1000)
ys = (2*np.pi * std**2)**(-0.5) * np.exp(-1 * (xs-mean)**2 / (2*std**2))
plt.plot(xs, ys, label='Theoretical pdf')

# Plot the 1,2,3,4 sigma edges
for sigma_edge in bins[2:-2:2]:
    plt.axvline(sigma_edge, color='gray', alpha=0.5)

plt.legend()
plt.xlabel(r'$x$')
plt.ylabel(r'$P(x)$')
plt.xlim(bins[0], bins[-1])
plt.savefig('plots/ex1_normal_hist.pdf')
plt.close()
```

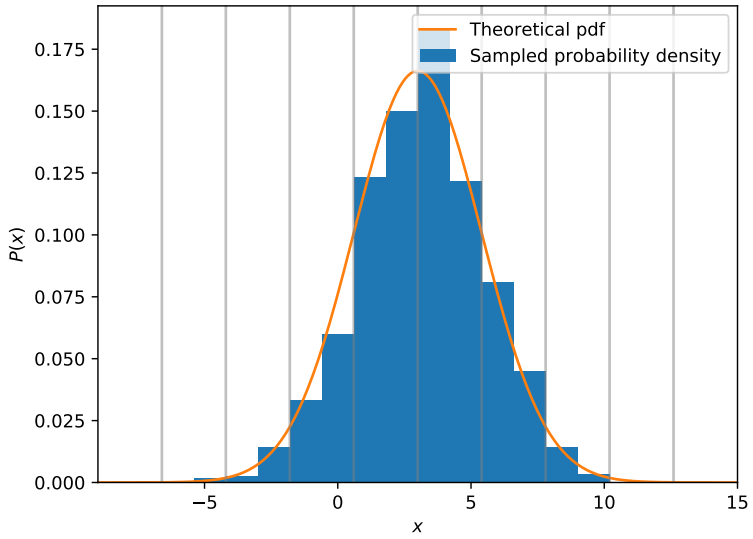

Figure 4: Histogram of 1000 random deviates with $\mu = 3$ and $\sigma = 2.4$, compared with theoretical PDF. The 1 to 5 σ edges are shown as vertical bars. The PDF and the empirical distribution show good agreement.

1c To compare two distributions without binning, we can use the Kolmogorov-Smirnov test. To use this test, we need to calculate the CDF of the normal distribution, for which we use the complementary error function. An approximation for the complementary error function using Chebyshev approximation is given in Numerical Recipes.

```
# Chebyshev coefficients for approximating the complementary error function
# Taken from NR 6.2.2
_erfc_Cheb_coeff = np.array([-1.3026537197817094, 6.4196979235649026e-1,
                             1.9476473204185836e-2, -9.561514786808631e-3,
                             -9.46595344482036e-4, 3.66839497852761e-4,
                             4.2523324806907e-5, -2.0278578112534e-5,
                             -1.624290004647e-6, 1.303655835580e-6,
                             1.5626441722e-8, -8.5238095915e-8,
                             6.529054439e-9, 5.059343495e-9,
                             -9.91364156e-10, -2.27365122e-10,
                             9.6467911e-11, 2.394038e-12,
                             -6.886027e-12, 8.94487e-13,
                             3.13092e-13, -1.12708e-13,
                             3.81e-16, 7.106e-15,
                             -1.523e-15, -9.4e-17,
                             1.21e-16, -2.8e-17])
def _erfc_Cheb(x):
    # Calculates erfc(x) for x>=0 using erfc(x) = t exp(-x**2 + P(t))
    # with t = 2 / (2+z) and P a polynomial defined by
```

```
    # the coefficients _erfc_Cheb
    ncoeffs = len(_erfc_Cheb_coeff)
    t = 2 / (2 + x)
    # Change of variables to -1 <= ty <= 1
    # Also multiply by 2 for using Clenshaws recurrence
    ty = 4*t - 2
    d = 0
    dd = 0
    for i in range(ncoeffs-1, 0, -1):
        temp = d
        d = ty * d - dd + _erfc_Cheb_coeff[i]
        dd = temp
    poly_total = 0.5 * (_erfc_Cheb_coeff[0] + ty * d) - dd
    return t * np.exp(-1*x**2 + poly_total)

def erfc(x):
    # Calculate the complementary error function
    if x < 0:
        return 2 - _erfc_Cheb(-x)
    else:
        return _erfc_Cheb(x)

def normal_cdf(x):
    # Calculates the cdf for the standard normal distribution
    return 0.5 * erfc(-1 * 2**-0.5 * x)
```

We also need to sort the data, for this we use mergesort.

```
def swap(A, a, b):
    temp = A[a]
    A[a] = A[b]
    A[b] = temp

def selection_sort(A):
    N = len(A)
    for i in range(N - 1):
        min_el = i
        for j in range(i+1, N):
            if A[j] < A[min_el]:
                min_el = j
        swap(A, i, min_el)

def _merge(in1, in2, out):
    N1 = len(in1)
    N2 = len(in2)
    N = N1 + N2
    j1 = 0
    j2 = 0
    for i in range(N):
        try:
            if in1[j1] < in2[j2]:
```

```
                out[i] = in1[j1]
                j1 += 1
            else:
                out[i] = in2[j2]
                j2 += 1
        except IndexError:
            # We are at the end of one of the lists
            # Now just append the other list onto out
            if j1 == N1:
                out[i:] = in2[j2:]
            else:
                out[i:] = in1[j1:]
            return

def mergesort(A):
    N = len(A)
    # Lists of size 1 (or 0) are trivially sorted
    # However for small lists selection sort is more efficient
    if N < 10:
        selection_sort(A)
        return
    middle = N >> 1
    mergesort(A[:middle])
    mergesort(A[middle:])
    _merge(np.copy(A[:middle]), np.copy(A[middle:]), A)
```

Now we can finally define and use the KS-test.

```
def KS_cdf(z):
    # Calculates Kolmogorov-Smirnov cdf using approximation
    # from the lecture slides
    if z == 0:
        return 0
    elif z < 1.18:
        exp_term = np.exp(-1 * np.pi**2 / (8 * z**2))
        return (2*np.pi)**0.5 * z**-1 * (exp_term + exp_term**9 + exp_term**25)
    else:
        exp_term = np.exp(-2 * z**2)
        return 1 - 2 * (exp_term - exp_term**4 + exp_term**9)

def KS_test(data, cdf):
    data = np.copy(data)
    mergesort(data)
    N = len(data)
    D = 0
    prev_cdf = 0
    for i, sample in enumerate(data):
        data_cdf = (i + 1) / N
        dist_cdf = cdf(sample)
        distance = max(np.abs(data_cdf - dist_cdf), np.abs(prev_cdf - dist_cdf))
```

```
        if distance > D:
            D = distance
        prev_cdf = data_cdf
    p_val = 1 - KS_cdf(D * (N**0.5 + 0.12 + 0.11 * N**-0.5))
    return D, p_val

Ns = (10**np.linspace(1, 5, num=41)).astype(int)
p_my_KS = np.empty_like(Ns, dtype=np.float64)
p_sp_KS = np.empty_like(Ns, dtype=np.float64)
data = NUR_random.Box_Muller(100000, generator)
for i, N in enumerate(Ns):
    _, p_my_KS[i] = NUR_random.KS_test(data[:N], NUR_random.normal_cdf)
    _, p_sp_KS[i] = kstest(data[:N], 'norm')

plt.scatter(Ns, p_my_KS, label='My results')
plt.scatter(Ns, p_sp_KS, label='Scipy results')
plt.axhline(0.05, color='gray', alpha=0.5)

plt.xscale('log')
plt.xlabel('Number of Gaussian deviates')
plt.ylabel('p-value')
plt.title('Kolmogorov-Smirnov test')
plt.ylim(0, 1)
plt.legend()
plt.savefig('plots/ex1_KS_test.pdf')
plt.close()
```

1d The Kuiper's test is an improvement on the KS-test, because it is more sensitive away from the median compared to the KS-test. The implementation and testing of the Kuiper's test is shown below.

```
def Kuipers_cdf(z):
    # Upper tail approximation for the Kuipers test value
    # Based on Stephens (1970)
    if z < 1:
        return 1
    return (8*z**2 - 2) * np.exp(-2*z**2)

def Kuipers_test(data, cdf):
    # Note that the p values returned are only accurate if they are small
    # Accurate within 2 decimal places for p < 0.74
    # Thus, a rejection of H0 is real, but the p-values are not distributed
    # as could be expected
    data = np.copy(data)
    mergesort(data)
    N = len(data)
    D_plus = 0
    D_minus = 0
    prev_cdf = 0
```

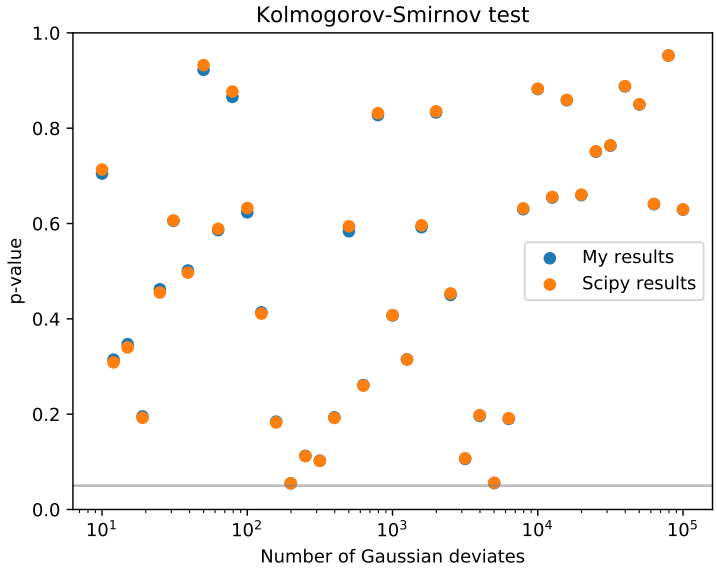

Figure 5: Results of 1-sided KS-test using my code and scipy for comparison, as a function of amount of random numbers tested. Also shown is a p-value threshold of 0.05. The p-values compare well with the scipy results and there are no results that indicate the generated distribution is incorrect.

```
    for i, sample in enumerate(data):
        data_cdf = (i + 1) / N
        dist_cdf = cdf(sample)
        distance_plus = data_cdf - dist_cdf
        distance_minus = dist_cdf - prev_cdf
        if distance_plus > D_plus:
            D_plus = distance_plus
        if distance_minus > D_minus:
            D_minus = distance_minus
        prev_cdf = data_cdf
    D = D_minus + D_plus
    p_val = Kuipers_cdf(D * (N**0.5 + 0.155 + 0.24 * N**-0.5))
    return D, p_val

from astropy.stats import kuiper
p_Kuiper = np.empty_like(Ns, dtype=np.float64)
p_Kuiper_astro = np.empty_like(Ns, dtype=np.float64)
for i, N in enumerate(Ns):
    _, p_Kuiper[i] = NUR_random.Kuipers_test(data[:N], NUR_random.normal_cdf)
    _, p_Kuiper_astro[i] = kuiper(data[:N], norm.cdf)

plt.scatter(Ns, p_Kuiper, label='My results')
plt.scatter(Ns, p_Kuiper_astro, label='Astropy results')
plt.axhline(0.05, color='gray', alpha=0.5)
```

```
plt.xscale('log')
plt.xlabel('Number of Gaussian deviates')
plt.ylabel('p-value')
plt.title("Kuiper's test")
plt.ylim(0, 1)
plt.legend()
plt.savefig('plots/ex1_Kuiper_test.pdf')
plt.close()
```

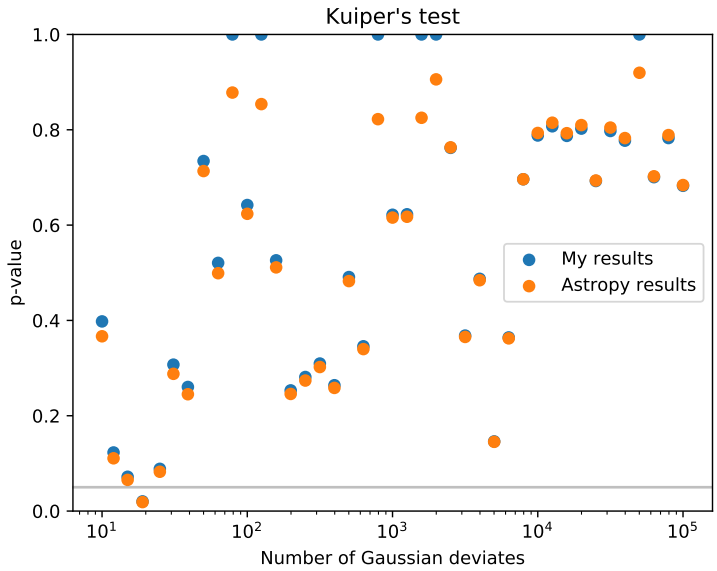

Figure 6: Results of 1-sided Kuiper's test using my code and astropy for comparison, as a function of amount of random numbers tested. Also shown is a p-value threshold of 0.05. The p-values compare well with the astropy results and there are no results that indicate the generated distribution is incorrect. There is 1 point with p-value below 0.05, but this is to be expected, since we are performing 41 tests.

1e We can also use a 2-sided KS test to compare two empirical distributions. Code for this is shown below, based on the implementation in Numerical Recipes.

```
def KS_test_2(data1, data2):
    data1 = np.copy(data1)
    data2 = np.copy(data2)
    mergesort(data1)
    mergesort(data2)
    N1 = len(data1)
    N2 = len(data2)
```

```
    D = 0

    i1 = i2 = 0
    f1 = f2 = 0
    while i1 < N1 and i2 < N2:
        d1 = data1[i1]
        d2 = data2[i2]
        if d1 <= d2:
            i1 += 1
            f1 = i1 / N1
        if d2 <= d1:
            i2 += 1
            f2 = i2 / N2
        distance = np.abs(f2 - f1)
        if distance > D:
            D = distance
    N_eff_sqrt = np.sqrt((N1 * N2) / (N1 + N2))
    p_val = 1 - KS_cdf(D * (N_eff_sqrt + 0.12 + 0.11 / N_eff_sqrt))
    return D, p_val
```

We have used the 2-sided KS test to compare 10 sets of random numbers with our own distribution of standard normals, to find out which sets are not standard normal distributed.

```
randoms_to_test = np.loadtxt('DataFiles/randomnumbers.txt')
p_vals_test = np.empty((Ns.size, 10))
for i, N in enumerate(Ns):
    for j in range(10):
        _, p_vals_test[i, j] = NUR_random.KS_test_2(data[:N], randoms_to_test[:N, j])
for j in range(10):
    plt.scatter(Ns, p_vals_test[:, j], label=f'Set {j+1}')
plt.axhline(0.05/41, color='gray', alpha=0.5)
plt.axhline(0.05/(41*10), color='gray', alpha=0.5)

plt.xscale('log')
plt.xlabel('Amount of numbers tested')
plt.ylabel('p-value')
plt.title('Two-sided KS test')
plt.yscale('log')
plt.ylim(1e-4, 1)
plt.legend(loc='lower left')
plt.savefig('plots/ex1_KS_2sided.pdf')
plt.close()
```

From the results, we see that all datasets other than sets 4 and 6 are not standard normal distributed. Set 4 seems to follow the proper distribution. For set 6, interpretation of the result depends on one's own view of the relevant statistics.

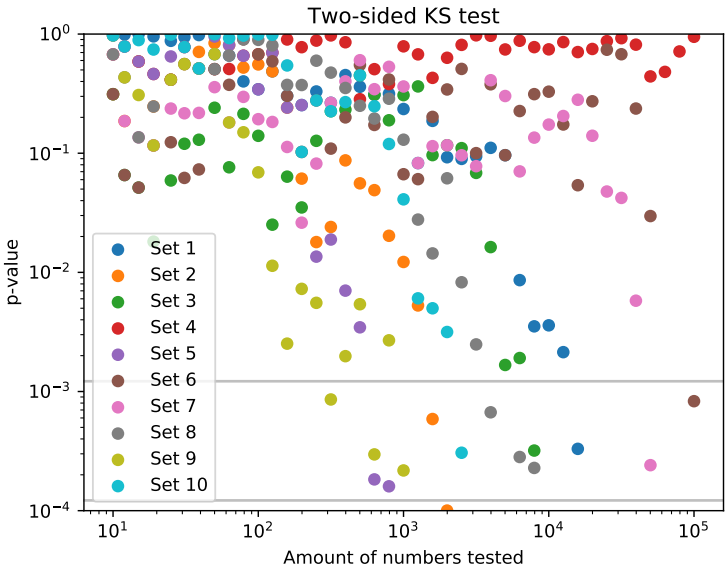

Figure 7: Results of the 2-sided KS test for 10 given datasets. Note that p-values are given on a log-scale. The grey horizontal lines correspond to p-values of 0.05/41 and 0.05/410, corresponding to a 0.05 threshold when performing 41 tests, or 41 tests on 10 datasets. Most datasets are clearly not standard normal distributed.

Exercise 2

To generate a Gaussian random field with known power spectrum, we can generate random normals in a Fourier plane. The variance of the random numbers is then given by $P(k) \propto k^n$. We also need to ensure that the generated Fourier plane has the required symmetry so that the field is real.

```
generator = XORshift(234)
print(f'Seed: 234')

def gen_random_field_2D(N, n):
    field = np.zeros((N, N), dtype=np.complex128)
    for i in range(N):
        for j in range(N//2 + 1):
            k = (i**2 + j**2)**0.5
            try:
                std = k**(n / 2)
                normals = Box_Muller(2, generator) * std
                amp = normals[0] + 1j * normals[1]
                field[i, j] = amp
                field[-i, -j] = amp.conjugate()
            except ZeroDivisionError:
                field[i, j] = 0
    field[0, 0] = 0
    field[0, N//2] = 2 * field[0, N//2].real
    field[N//2, N//2] = 2 * field[N//2, N//2].real
    field[N//2, 0] = 2 * field[N//2, 0].real
    field = np.fft.ifft2(field)
    return field.real

for n in [-1, -2, -3]:
    field = gen_random_field_2D(1024, n)
    plt.matshow(field, origin='lower')
    plt.xlabel('x (Mpc)')
    plt.ylabel('y (Mpc)')
    plt.title(f'n = {n}')
    plt.savefig(f'plots/ex2_n_{abs(n)}.pdf')
```

Three random fields with different power index are shown below.

We can choose the physical size ourselves, however this influences the interpretation of the plot. If we choose that the units of x and y are in Mpc, this means that for a 1024×1024 image, the maximum physical frequency is 1 cycle per 1 Gpc and the minimum frequency is 1 per Mpc.

We can see that a lower index n corresponds to more large scale structure, or equivalently less small-scale variation.

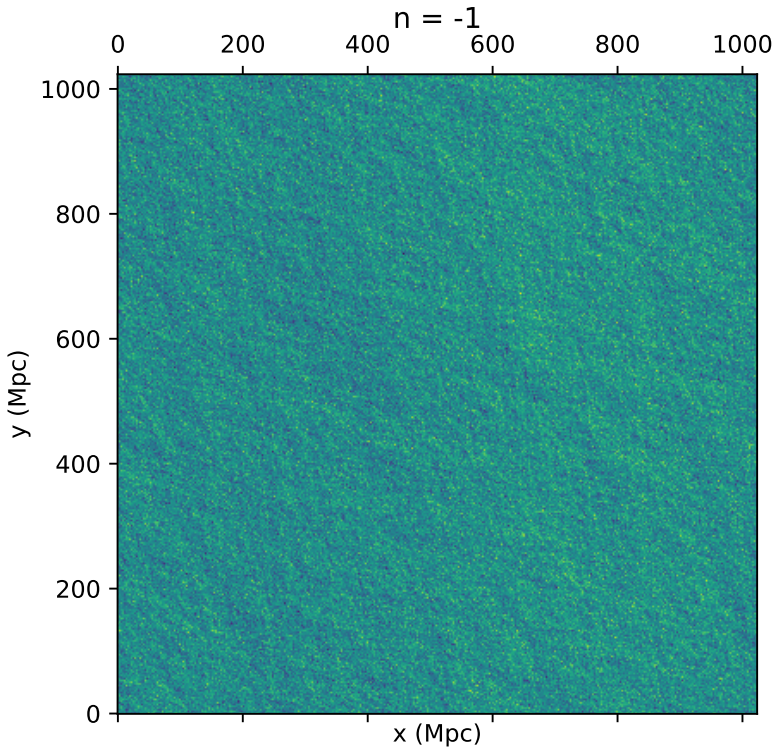

Figure 8: 1024×1024 Gaussian random field with power index $n = -1$.

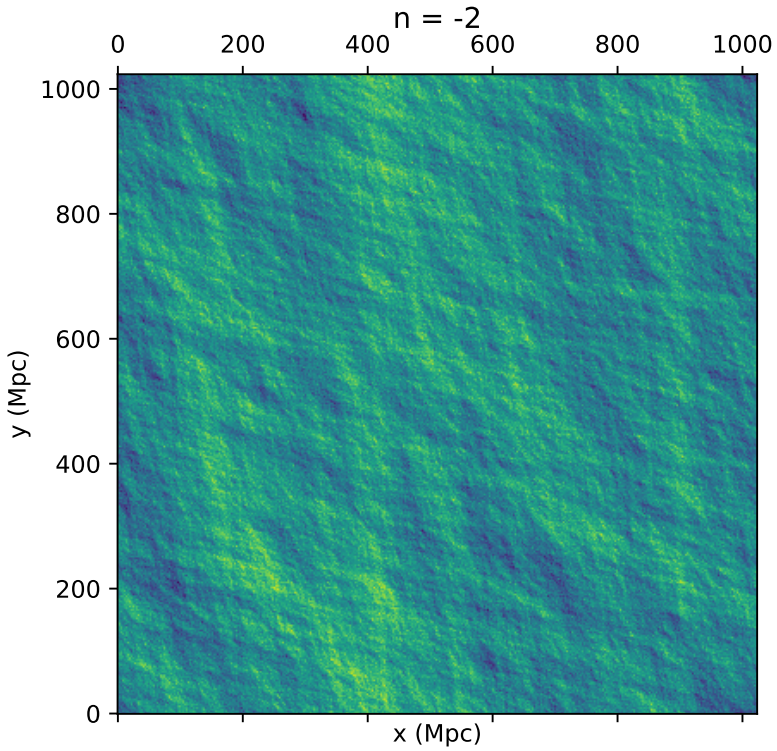

Figure 9: 1024×1024 Gaussian random field with power index $n = -2$.

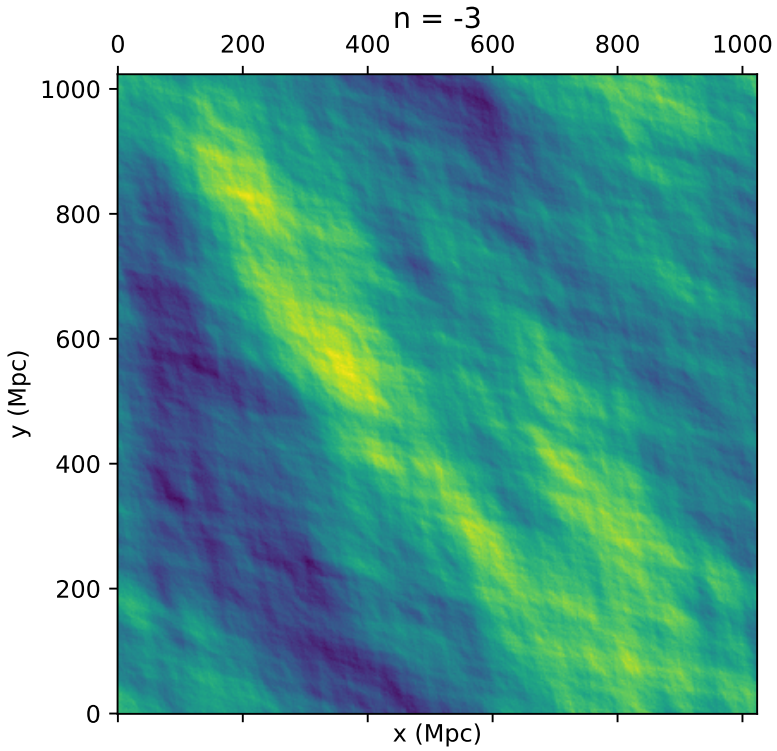

Figure 10: 1024×1024 Gaussian random field with power index $n = -3$.

Exercise 3

The temporal part of the growth equation is:

$$\ddot{D} + 2H\dot{D} = \frac{3}{2}\Omega_0 \frac{H_0^2}{a^3} D \tag{1}$$

For an Einstein-de Sitter universe, where $\Omega_0 = \Omega_m = 1$, we get:

$$a(t) = \left(\frac{3}{2} H_0 t\right)^{2/3} \iff \frac{3}{2}\frac{H_0^2}{a^3} = \frac{2}{3t^2}$$
$$H = \frac{\dot{a}}{a} = \frac{2}{3t}$$

Therefore the linear growth equation for an EdS universe is:

$$\ddot{D} + \frac{4\dot{D}}{3t} = \frac{2D}{3t^2} \tag{2}$$

Using the guess $D \propto t^\alpha$ gives the following general solution:

$$D(t) = At^{2/3} + Bt^{-1} \tag{3}$$

A and B are determined by the initial conditions such that $A + B = D(1)$ and $\frac{2}{3}A - B = D'(1)$.

To solve this equation, we use an embedded RK5(4) solver, with code shown below:

```
import numpy as np

# Tsitouras 5th (4th) order RK method, from
# https://doi.org/10.1016/j.camwa.2011.06.002
# Coefficients taken from the DifferentialEquations.jl package
# Load A coefficients from file
Tsit5A = np.loadtxt('Tsit5A.txt', dtype=np.float64)
# Hardcode c coefficients
Tsit5c = np.array([0.0, 0.161, 0.327, 0.9, 0.980026, 1.0, 1.0], dtype=np.float64)
# coefficients for 5th and 4th order method
Tsit5b_5 = Tsit5A[-1, :]
Tsit5b_4 = np.array([0.0946808, 0.00918357, 0.487771, 1.2343, -2.70771, 1.86663, 0.0151515
                     dtype=np.float64)
# No need to calculate 4th order directly, can precalculate terms
Tsit5b_error = Tsit5b_5 - Tsit5b_4
def Tsit5_step(x, h, y, f_dy, k0):
    ks = np.zeros((7, k0.size))
    xs = x + h * Tsit5c
    ks[0] = k0
    for i in range(1, 7):
        y_val = y + h * Tsit5A[i] @ ks
        ks[i] = f_dy(xs[i], y_val)
    new_y = y + h * Tsit5b_5 @ ks
    err = h * Tsit5b_error @ ks
```

```
    return new_y, err, ks[-1]

def Tsit5_adaptive(x1, x2, y0, f_dy, atol=1e-6, rtol=1e-6, maxsteps=10_000):
    S = 0.95
    step_order = 5
    xs = np.empty(maxsteps+1)
    ys = np.empty((maxsteps+1, y0.size))
    xs[0] = x1
    ys[0] = y0
    k0 = f_dy(xs[0], ys[0])
    h = (x2 - x1) / maxsteps
    i = 0
    for nstep in range(maxsteps):
        new_y, err, k_new = Tsit5_step(xs[i], h, ys[i], f_dy, k0)
        scale = atol + rtol * np.maximum(np.abs(new_y), np.abs(ys[i]))
        norm_err = (np.sum((err/scale)**2) / err.size)**0.5
        if norm_err <= 1:
            xs[i+1] = xs[i] + h
            ys[i+1] = new_y
            i += 1
            k0 = k_new
            if xs[i] >= x2:
                break
            h *= min(5, S * norm_err**(-1/step_order))
            if xs[i] + h > x2:
                h = x2 - xs[i]
        else:
            h *= max(.2, S * norm_err**(-1/step_order))
    if xs[i] < x2:
        print('max steps reached before reaching x2')
    return xs[:i+1], ys[:i+1]
```

And the hardcoded coefficients for A:

```
0.0         0.0         0.0         0.0         0.0         0.0         0.0
0.161         0.0         0.0         0.0         0.0         0.0         0.0
-0.008480655492356989         0.335480655492357         0.0         0.0         0.0         0.0
2.8971530571054935         -6.359448489975075         4.3622954328695815         0.0         0
5.325864828439257         -11.748883564062828         7.4955393428898365         -0.092495066
5.86145544294642         -12.92096931784711         8.159367898576159         -0.071584973281
0.09646076681806523         0.01         0.4798896504144996         1.379008574103742
```

We then solve for different initial conditions, with a relative tolerance of 10^{-5} and an absolute tolerance of 10^{-7}.

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

from ODE import Tsit5_adaptive

def analytic_solution(ts, D, D_prime):
```

```
    # Analytic solution of the EdS linear growth equation
    # Initial conditions are required at t=1
    A = (D + D_prime) * 3 / 5
    B = D - A
    return A*ts**(2/3) + B*ts**-1

def calc_plot_case(D, D_prime, label):
    def f_dD(t, pars):
        # Set up the differential equation
        dD_dt = pars[1]
        d2D_dt2 = 2*pars[0]/(3*t**2) - 4*pars[1]/(3*t)
        return np.array((dD_dt, d2D_dt2))
    t0 = 1
    t1 = 1000
    init = np.array((D, D_prime))
    ts, ys_numeric = Tsit5_adaptive(t0, t1, init, f_dD,
                                    maxsteps=200, atol=1e-7, rtol=1e-5)
    Ds_numeric = ys_numeric[:, 0]
    Ds_analytic = analytic_solution(ts, D, D_prime)

    plt.loglog(ts, Ds_analytic, label='Analytic solution', color='k')
    plt.plot(ts, Ds_numeric, label='Numeric solution', color='b')
    plt.scatter(ts, Ds_numeric, color='b')

    plt.xlabel('t [yr]')
    plt.ylabel('Growth factor D')
    plt.title(f"D={D} D'={D_prime}")
    plt.legend()

    plt.savefig(f'plots/ex3_Growth_ODE_case_{label}.pdf')
    plt.close()

calc_plot_case(3, 2, '1')
calc_plot_case(10, -10, '2')
calc_plot_case(5, 0, '3')
```

With results shown below.

The solver reproduces the analytical soltuion very well. There is a small deviation for the decaying solution, which is because the absolute tolerance is relatively large for this case.

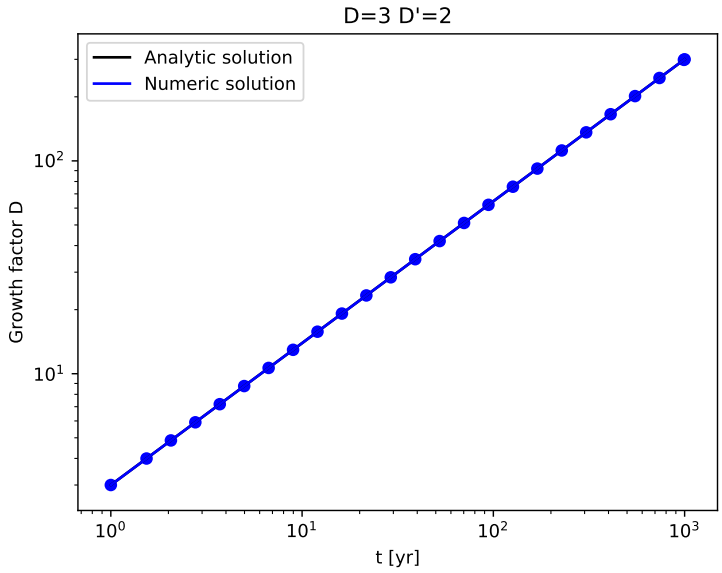

Figure 11: Evolution of linear growth factor for $D(1) = 3$, $D'(1) = 2$.

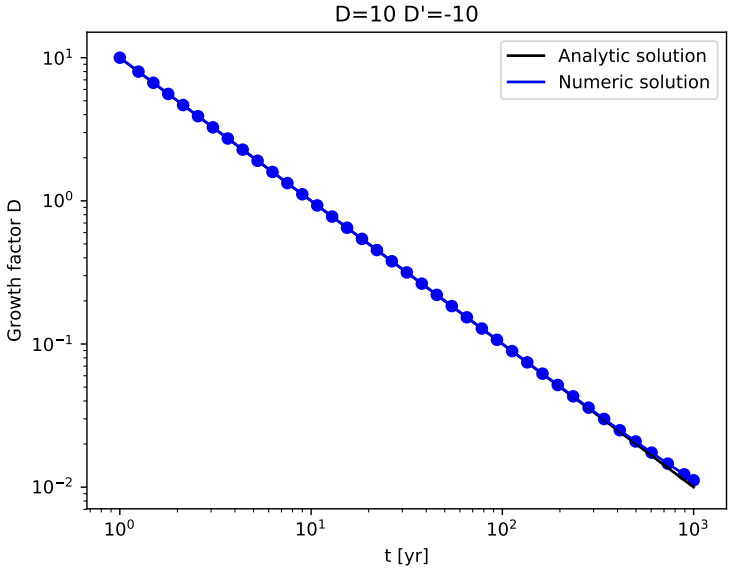

Figure 12: Evolution of linear growth factor for $D(1) = 10$, $D'(1) = -10$.

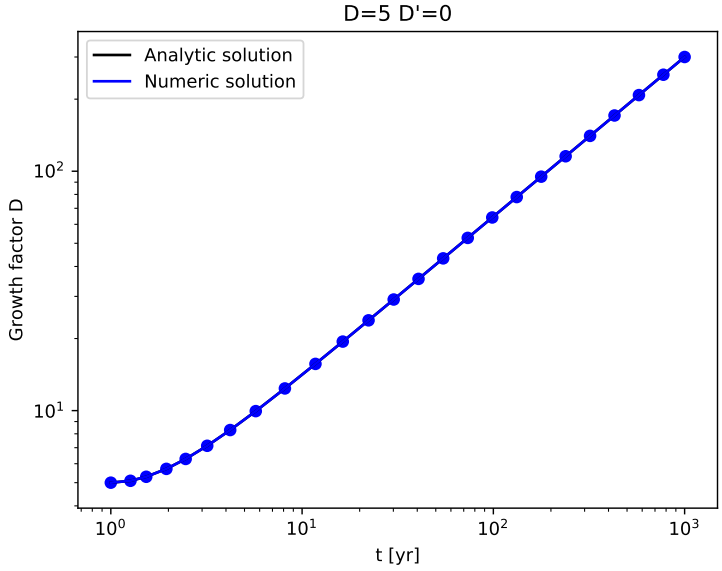

Figure 13: Evolution of linear growth factor for $D(1) = 5$, $D'(1) = 0$.

Exercise 4

We use these constants in this exercise:

```
generator = XORshift(138)

# Normalization of the power spectrum
alpha = 1/8

# Cosmological parameters
Omega_L = 0.7
Omega_M = 0.3
# H_0 in 1/yr
H_0 = 7.16E-11

# Converts from Mpc/yr to km/s
v_convert = 977792221673
```

4a We want to calculate

$$D(z) = \frac{5\Omega_m H_0^2}{2} H(z) \int_z^\infty \frac{1+z'}{H^3(z')} dz' \tag{4}$$

Note that

$$a = \frac{1}{1+z} \implies dz = -\frac{da}{a^2}$$

So that this is equivalent to

$$D(a) = \frac{5\Omega_m}{2} \frac{H(a)}{H_0} \int_0^a \left(a' \frac{H(a')}{H_0} \right)^{-3} da' \tag{5}$$

We know from the Friedmann equation

$$\frac{H(a)}{H_0} = \left(\Omega_m a^{-3} + \Omega_\Lambda \right)^{1/2} \tag{6}$$

So we can calculate $D(z = 50) = D(a = 1/51)$ by numerical integration. We need to perform an open integration, because there is a singularity at $a' = 0$. For this we use the open Romberg integration routine shown below.

```
def open_Romberg_int(f, a, b, order=4):
    def combine(MA, LA, m):
        prefactor = 1/(9**m - 1)
        return MA + prefactor * (MA - LA)
    n_points = 3**order
    xs = np.linspace(a, b, num=n_points*2+1, endpoint=True)[1::2]
    fs = f(xs)
    approximations = np.tile(np.nan, (order + 1, order + 1))
    # Initial approximations of integral
    for n in range(order + 1):
        stride = 3**(order - n)
        start = stride//2
```

```
            dx = (b-a) / 3**n
            approximations[n, 0] = np.sum(fs[start::stride]) * dx
        # Combine the approximations
        for m in range(1, order + 1):
            for n in range(m, order + 1):
                MA = approximations[n, m-1]
                LA = approximations[n-1, m-1]
                approximations[n, m] = combine(MA, LA, m)
        return approximations[-1, -1]

def H_H0(a):
    # Calculates Hubble parameter divided by Hubble constant
    # as a function of the scale factor a
    return (Omega_M * a**-3 + Omega_L)**0.5

def growth_factor_calc(a, return_int=False):
    to_integrate = lambda a: (a * H_H0(a))**-3
    integral_result = open_Romberg_int(to_integrate, 0, a, order=8)
    growth_factor = 2.5 * Omega_M * H_H0(a) * integral_result
    if return_int:
        return growth_factor, integral_result
    else:
        return growth_factor

z = 50
a = 1 / (1 + z)
growth_factor, integral_result = growth_factor_calc(a, return_int=True)
print(f'D(z=50) = {growth_factor}')
```

With the result shown below.

```
D(z=50) = 0.01960778042811382
```

To calculate $\dot{D}$, we use that

$$\dot{D} = \frac{dD}{da}\dot{a} = \frac{dD}{da}aH$$

and derive analytically that

$$\frac{dD}{da} = \frac{5\Omega_m}{2a^3}\left(\frac{H}{H_0}\right)^{-1}\left(\left(\frac{H}{H_0}\right)^{-1} - \frac{3\Omega_m}{2a}\int_0^a \left(a'\frac{H(a')}{H_0}\right)^{-3} da'\right)$$

so that

$$\dot{D} = \frac{5H_0\Omega_m}{2a^2}\left(\left(\frac{H}{H_0}\right)^{-1} - \frac{3\Omega_m}{2a}\int_0^a \left(a'\frac{H(a')}{H_0}\right)^{-3} da'\right) \tag{7}$$

Where we use the numerical result for the integral we obtained earlier.

We can also calculate the derivative $\frac{dD}{da}$ numerically and then multiply by $\dot{a}$. We use Ridders' method to do this.

```
def central_diff_deriv(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)

def Ridders_deriv(f, x, h, target_err=1e-10, max_order=5):
    def CDD(step_size):
        return central_diff_deriv(f, x, step_size)
    # factor by which to reduce step size
    d = 1.4
    err = np.inf
    ans = np.nan
    D = np.tile(np.nan, (max_order, max_order))
    D[0][0] = CDD(h)
    for i in range(1, max_order):
        h /= d
        D[i][0] = CDD(h)
        for j in range(1, i+1):
            d_factor = d**(2 * j)
            D[i][j] = (d_factor * D[i][j-1] - D[i-1][j-1]) / (d_factor  - 1)
            new_err = max(abs(D[i][j] - D[i][j-1]), abs(D[i][j] - D[i-1][j-1]))
            if new_err < err:
                err = new_err
                ans = D[i][j]
                if err < target_err:
                    return ans, err
        if abs(D[i][i] - D[i-1][i-1]) > 2 * err:
            return ans, err
    return ans, err

def analytical_deriv(a, integral_result):
    prefactor = 5 * H_0 * Omega_M / (2 * a**2)
    first_term = 1/H_H0(a)
    second_term = 3 * Omega_M * integral_result / (2 * a)
    return prefactor * (first_term - second_term)

analytical_result = analytical_deriv(a, integral_result)
dD_da, num_der_err = Ridders_deriv(growth_factor_calc, a, 1e-5)
numerical_result = a * H_0 * H_H0(a) * dD_da

print(f'Analytical result for dD/dt: {analytical_result:.10E} yr-1')
print(f'Numerical  result for dD/dt: {numerical_result:.10E} yr-1')
rel_error = abs(numerical_result - analytical_result) / analytical_result
print(f'Relative difference between analytical and numerical derivative: {rel_error:.2E}')
```

With results shown below

```
Analytical result for dD/dt: 2.8006381572E-10 yr-1
Numerical  result for dD/dt: 2.8006381571E-10 yr-1
Relative difference between analytical and numerical derivative: 1.87E-11
```

As we can see, the agreement between the analytical and the numeric derivative is very good.

4c Using the Zeldovich approximation, we can generate initial conditions in 2D. We use the 2D IFFT from numpy to generate initial conditions for 64^2 particles starting from $z = 50$, and enforce the periodic boundary conditions using the mod (%) operator. We also use a utility function to calculate the frequencies in the Fourier plane.

```
def fftfreq(N):
    # returns frequencies of FFT transform
    freqs = np.empty(N)
    pos_edge = (N - 1) // 2 + 1
    freqs[:pos_edge] = np.arange(0, pos_edge)
    freqs[pos_edge:] = np.arange(-(N//2), 0)
    return freqs/N

def gen_Zeldovich_S_2D(grid_size):
    c_k = np.empty((grid_size, grid_size, 2), dtype=np.complex128)
    Nq_index = grid_size // 2
    for i in range(grid_size):
        for j in range(grid_size // 2 + 1):
            normals = Box_Muller(2, generator)
            if i > grid_size // 2:
                k = ((2 * np.pi / grid_size)**2 * ((grid_size-i)**2 + j**2))**0.5
            else:
                k = ((2 * np.pi / grid_size)**2 * (i**2 + j**2))**0.5
            c_k[i, j] = alpha * (normals[0] - 1j*normals[1]) / (2 * k**3)
            c_k[-i, -j] = c_k[i, j].conjugate()
    freqs = 2 * np.pi * fftfreq(grid_size)
    c_k[:, :, 0] *= 1j * freqs[:, np.newaxis]
    c_k[:, :, 1] *= 1j * freqs[np.newaxis, :]

    # Correctly set the components that should be real
    c_k[0, 0] = 0
    c_k[0, Nq_index] = c_k[0, Nq_index].real * 2
    c_k[0, Nq_index] = c_k[0, Nq_index].real * 2
    c_k[Nq_index, Nq_index] = c_k[Nq_index, Nq_index].real * 2

    S = grid_size**2 * np.fft.ifft2(c_k, axes=(0, 1)).real
    return S

# Amount of particles in 1 dimension
N_1 = 64
init_pos = np.empty((N_1, N_1, 2))
init_pos[:, :, 0] = np.arange(N_1)[:, np.newaxis]
init_pos[:, :, 1] = np.arange(N_1)[np.newaxis, :]

S_2D = gen_Zeldovich_S_2D(N_1)

scale_factors = np.linspace(0.025, 1, num=90)
ypos_particles = np.empty((scale_factors.size, 10))
yvel_particles = np.empty_like(ypos_particles)
```

```
for i, a in enumerate(scale_factors):
    z = 1/a - 1
    D, int_result = growth_factor_calc(a, return_int=True)
    dD_dt = analytical_deriv(a, integral_result)
    positions = np.mod(init_pos + S_2D*D, N_1)
    velocities = -1 * a**2 * dD_dt * S_2D

    ypos_particles[i] = positions[0, :10, 1]
    yvel_particles[i] = velocities[0, :10, 1]

    plt.scatter(positions[:, :, 0].flat, positions[:, :, 1].flat,
                s=1, color='k')

    plt.xlim(0, 64)
    plt.ylim(0, 64)

    plt.xlabel('x (Mpc)')
    plt.ylabel('y (Mpc)')
    plt.title(f'z={z:.2f}')
    plt.savefig(f'movies/2D_zeldovich/{i:03d}.png')
    plt.close()
# Rescale positions from the range 0, 64 to -32, 31
rescaled_pos = np.mod(ypos_particles + N_1//2, N_1) - N_1//2
plt.plot(scale_factors, rescaled_pos)
plt.xlim(0, 1)
plt.xlabel('Scale factor $a$')
plt.ylabel('y-coordinate of particles (Mpc)')
plt.savefig('plots/ex4_2D_ypos.pdf')
plt.close()

plt.plot(scale_factors, v_convert * yvel_particles)
plt.xlim(0, 1)
plt.xlabel('Scale factor $a$')
plt.ylabel('y-velocity of particles (km/s)')
plt.savefig('plots/ex4_2D_yvel.pdf')
plt.close()
```

A movie of the evolution can be found in `movies/2D_zeldovich/ICs.mp4`

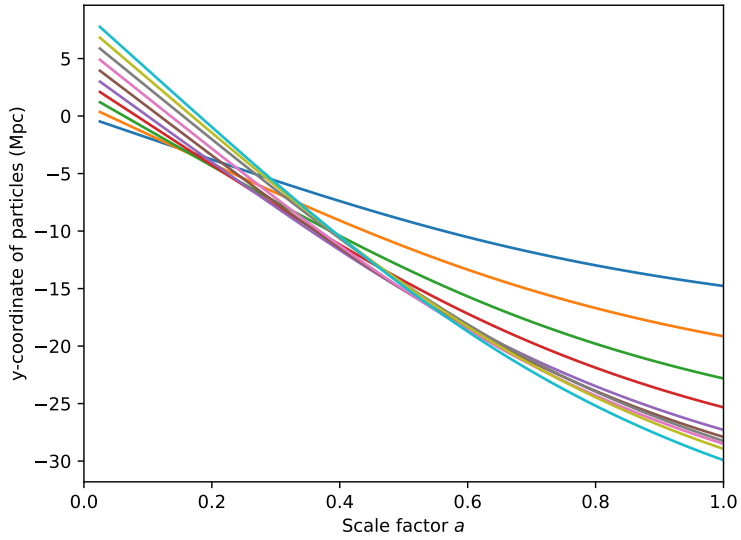

Figure 14: y-positions of the first 10 particles in the 2D ZA as a function of scale factor.

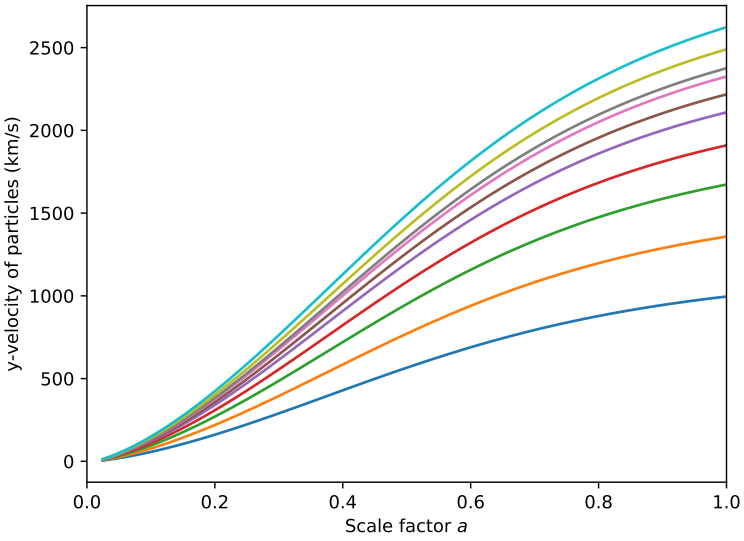

Figure 15: y-velocities of the first 10 particles in the 2D ZA as a function of scale factor.

4d We can also do this in 3D. We generate initial conditions for a later simulation, and plot the evolution of the initial conditions over time. Movies can be found in `movies/3D_zeldovich/xy/ICs.mp4`, `movies/3D_zeldovich/xz/ICs.mp4`, `movies/yz/3D_zeldovich/ICs.mp4` for slices in the xy, xz and yz plane respectively.

```
def gen_Zeldovich_S_3D(grid_size):
    c_k = np.empty((grid_size, grid_size, grid_size, 3), dtype=np.complex128)
    Nq_index = grid_size // 2
    for i in range(grid_size):
        for j in range(grid_size):
            for k in range(grid_size // 2 + 1):
                normals = Box_Muller(2, generator)
                if i > grid_size // 2:
                    i_mag = grid_size - i
                else:
                    i_mag = i
                if j > grid_size // 2:
                    j_mag = grid_size - j
                else:
                    j_mag = j
                k_vec = ((2 * np.pi / grid_size)**2 * (i_mag**2 + j_mag**2 + k**2))**0.5
                c_k[i, j, k] = alpha * (normals[0] - 1j*normals[1]) / (2 * k_vec**3)
                c_k[-i, -j, -k] = c_k[i, j, k].conjugate()
    freqs = 2 * np.pi * fftfreq(grid_size)
    c_k[:, :, :, 0] *= 1j * freqs[:, np.newaxis, np.newaxis]
    c_k[:, :, :, 1] *= 1j * freqs[np.newaxis, :, np.newaxis]
    c_k[:, :, :, 2] *= 1j * freqs[np.newaxis, np.newaxis, :]

    # Correctly set the components that should be real
    for i in (0, Nq_index):
        for j in (0, Nq_index):
            for k in (0, Nq_index):
                c_k[i, j, k] = c_k[i, j, k].real * 2
    c_k[0, 0, 0] = 0

    S = grid_size**3 * np.fft.ifftn(c_k, axes=(0, 1, 2)).real
    return S

S_3D = gen_Zeldovich_S_3D(N_1)

init_pos = np.empty((N_1, N_1, N_1, 3))
init_pos[:, :, :, 0] = np.arange(N_1)[:, np.newaxis, np.newaxis]
init_pos[:, :, :, 1] = np.arange(N_1)[np.newaxis, :, np.newaxis]
init_pos[:, :, :, 2] = np.arange(N_1)[np.newaxis, np.newaxis, :]

# position calculation
z = 50
a = 1 / (1 + z)
D, _ = growth_factor_calc(a, return_int=True)
```

```
positions = np.mod(init_pos + S_3D*D, N_1)

# Because we will later use a leapfrog integrator
# velocities have to be calculated half a timestep earlier
a -= 0.00005
_, int_result = growth_factor_calc(a, return_int=True)
dD_dt = analytical_deriv(a, int_result)
velocities = -1 * a**2 * dD_dt * S_3D

zpos_particles = np.empty_like(ypos_particles)
zvel_particles = np.empty_like(zpos_particles)

with h5py.File('init.hdf5', 'w') as f:
    f.create_dataset('Coordinates', data=positions.reshape((-1, 3)))
    f.create_dataset('Velocities', data=velocities.reshape((-1, 3)))

for i, a in enumerate(scale_factors):
    z = 1/a - 1
    D, int_result = growth_factor_calc(a, return_int=True)
    dD_dt = analytical_deriv(a, int_result)
    positions = np.mod(init_pos + S_3D*D, N_1)
    velocities = -1 * a**2 * dD_dt * S_3D

    output_3D(z, i, positions, velocities, zpos_particles, zvel_particles)

rescaled_pos = np.mod(zpos_particles + N_1//2, N_1) - N_1//2
plt.plot(scale_factors, rescaled_pos)
plt.xlim(0, 1)
plt.xlabel('Scale factor $a$')
plt.ylabel('z-coordinate of particles (Mpc)')
plt.savefig('plots/ex4_3D_zpos.pdf')
plt.close()

plt.plot(scale_factors, v_convert * zvel_particles)
plt.xlim(0, 1)
plt.xlabel('Scale factor $a$')
plt.ylabel('z-velocity of particles (km/s)')
plt.savefig('plots/ex4_3D_zvel.pdf')
plt.close()
```

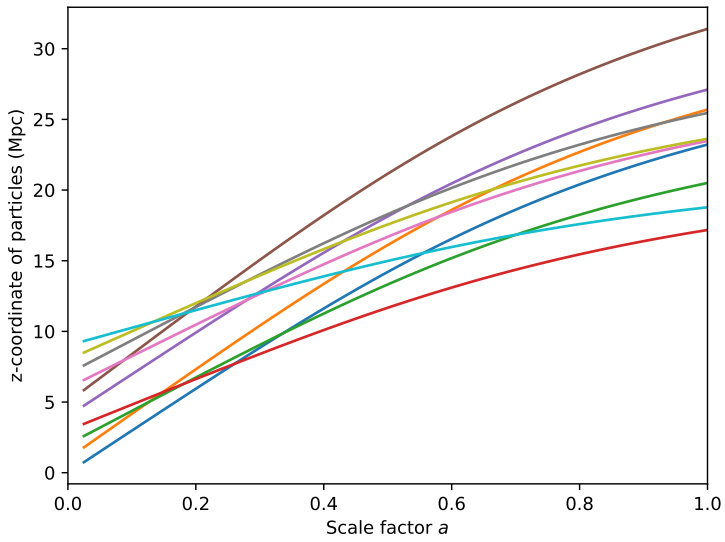

Figure 16: z-positions of the first 10 particles in the 3D ZA as a function of scale factor.

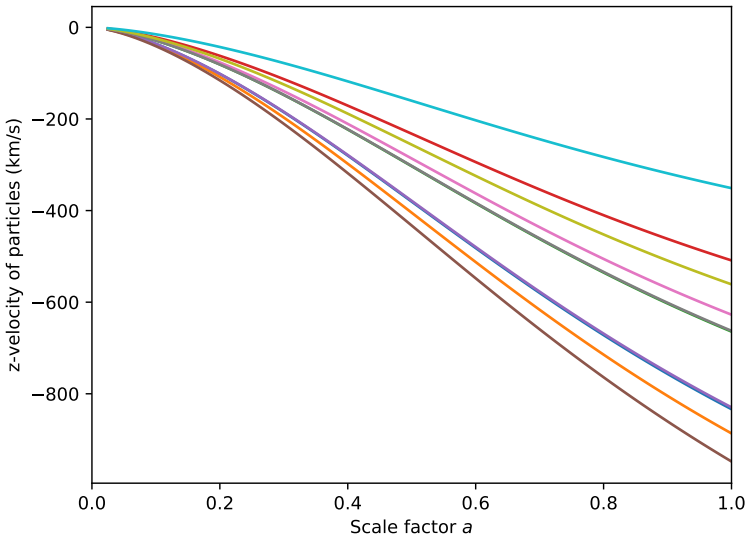

Figure 17: z-velocities of the first 10 particles in the 3D ZA as a function of scale factor.

Exercise 5

5a The mass density defining the Nearest Grid Point (NGP) method in 1D is:

$$S(x) = \frac{1}{\Delta x}\delta\left(\frac{x}{\Delta x}\right) \tag{8}$$

This is the density of a point particle.

Thus we need to assign all the mass of a given particle to the nearest point in our grid, explaining the name. Code for this is shown below.

```
def NGP_mass_assignment_3D(positions, grid_size=16):
    masses = np.zeros((grid_size, grid_size, grid_size))
    for pid in range(positions.shape[0]):
        x, y, z = positions[pid]
        # Calculate the 3 indices
        i = floor(x + 0.5) % grid_size
        j = floor(y + 0.5) % grid_size
        k = floor(z + 0.5) % grid_size
        masses[i, j, k] += 1
    return masses

np.random.seed(121)
positions = np.random.uniform(low=0, high=16, size=(3, 1024))

masses = NGP_mass_assignment_3D(positions.T)

for z in (4, 9, 11, 14):
    plt.matshow(masses[:, :, z], cmap='Greys')
    plt.colorbar()
    plt.xlabel('x (Mpc)')
    plt.ylabel('y (Mpc)')
    plt.savefig(f'plots/ex5_NGP_z{z}.pdf')
    plt.close()

xpositions = np.linspace(0, 16, num=1000, endpoint=False)
val_0 = np.empty_like(xpositions)
val_4 = np.empty_like(xpositions)

for i, xpos in enumerate(xpositions):
    masses = NGP_mass_assignment_3D(np.array([[xpos, 0, 0]]))
    val_0[i] = masses[0, 0, 0]
    val_4[i] = masses[4, 0, 0]

plt.plot(xpositions, val_0, label='Mass in cell 0')
plt.plot(xpositions, val_4, label='Mass in cell 4')
plt.xlabel('x position of particle (Mpc)')
plt.ylabel('Mass')
plt.xlim(0, 16)
plt.legend()
plt.title('Nearest Grid Point assignment')
```

```
plt.savefig('plots/ex5_NGP_xpos.pdf')
plt.close()
```

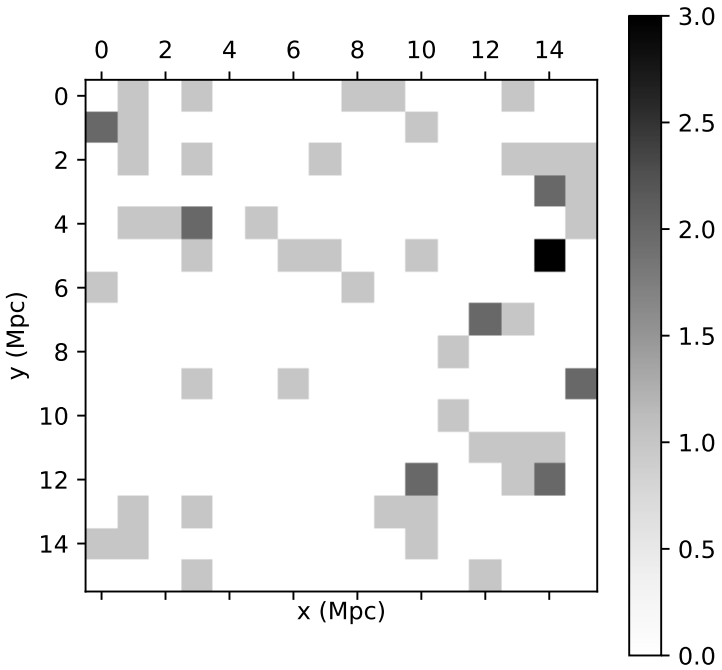

Figure 18: x-y slice of NGP mass assignment with $z = 4$

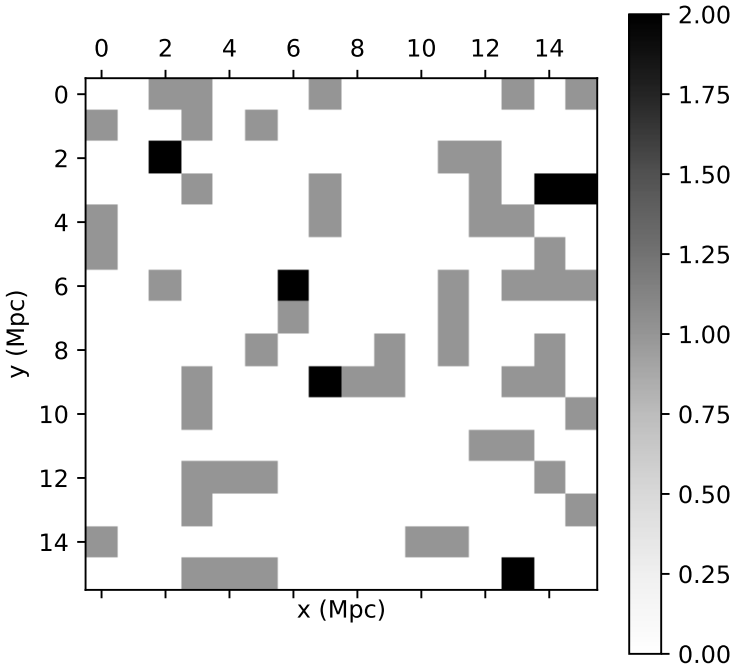

Figure 19: x-y slice of NGP mass assignment with $z = 9$

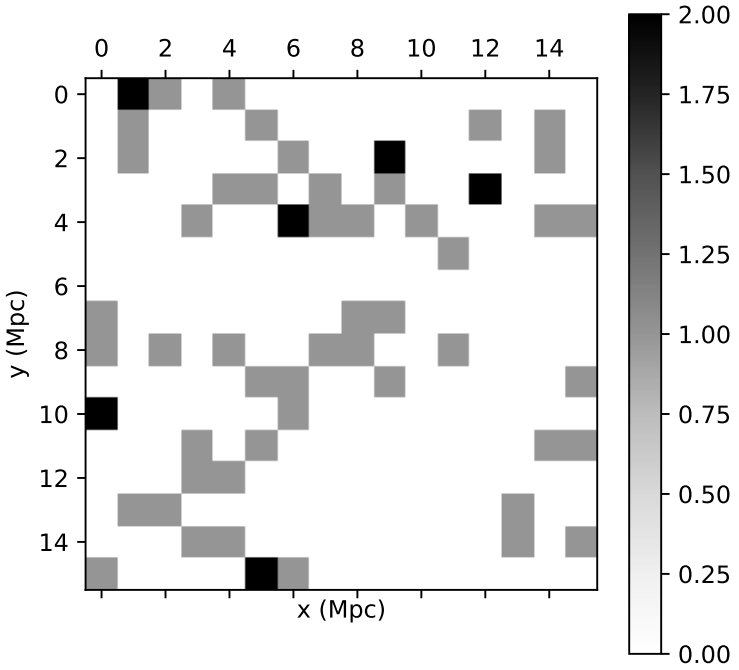

Figure 20: x-y slice of NGP mass assignment with $z = 11$

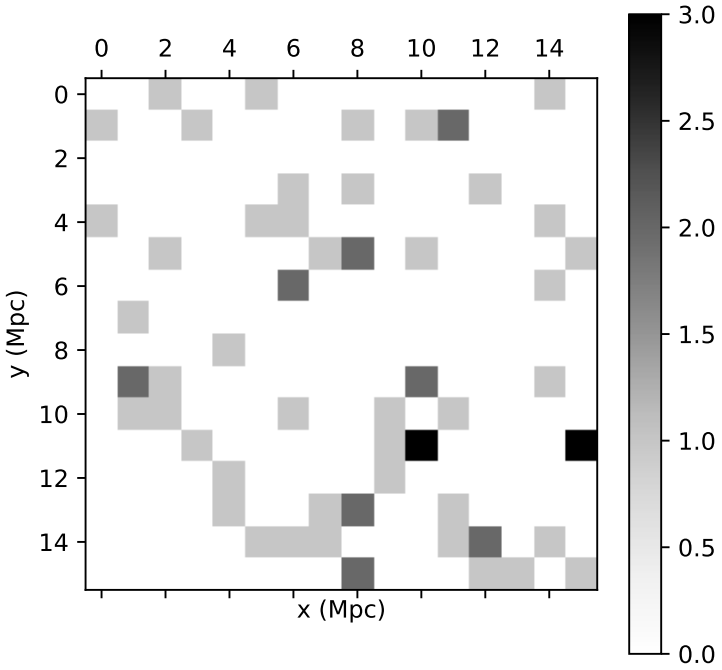

Figure 21: x-y slice of NGP mass assignment with $z = 14$

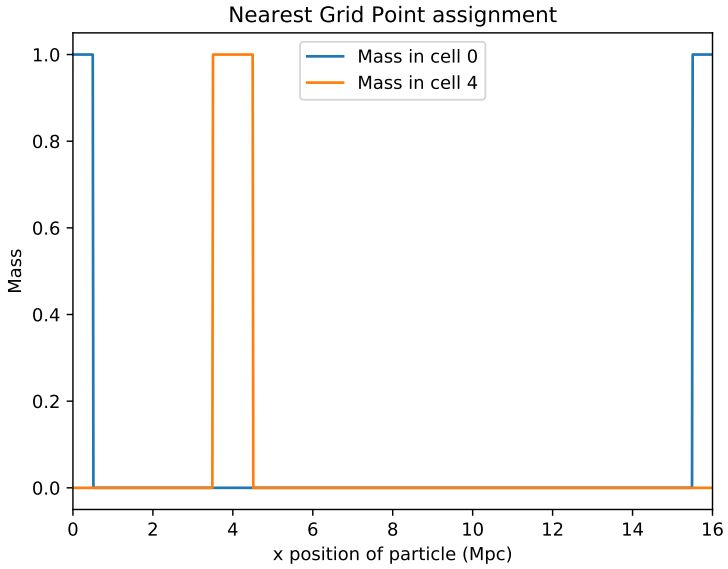

Figure 22: Values of mass in shown cells for one particle in NGP simulation with x-coordinate as shown

5b

5c For the Cloud in Cell method, we can use linear interpolation.

```
def CIC_mass_assignment_3D(positions, grid_size=16):
    masses = np.zeros((grid_size, grid_size, grid_size))
    for pid in range(positions.shape[0]):
        x, y, z = positions[pid]
        i = floor(x)
        j = floor(y)
        k = floor(z)
        ip1 = (i+1) % grid_size
        jp1 = (j+1) % grid_size
        kp1 = (k+1) % grid_size
        dx = x - i
        dy = y - j
        dz = z - k
        tx = 1 - dx
        ty = 1 - dy
        tz = 1 - dz
        masses[i,   j,   k  ] += tx * ty * tz
        masses[i,   j,   kp1] += tx * ty * dz
        masses[i,   jp1, k  ] += tx * dy * tz
        masses[i,   jp1, kp1] += tx * dy * dz
        masses[ip1, j,   k  ] += dx * ty * tz
        masses[ip1, j,   kp1] += dx * ty * dz
        masses[ip1, jp1, k  ] += dx * dy * tz
```

```
                masses[ip1, jp1, kp1] += dx * dy * dz
        return masses

masses = CIC_mass_assignment_3D(positions.T)

for z in (4, 9, 11, 14):
    plt.matshow(masses[:, :, z], cmap='Greys')
    plt.colorbar()
    plt.xlabel('x (Mpc)')
    plt.ylabel('y (Mpc)')
    plt.savefig(f'plots/ex5_CIC_z{z}.pdf')
    plt.close()

for i, xpos in enumerate(xpositions):
    masses = CIC_mass_assignment_3D(np.array([[xpos, 0, 0]]))
    val_0[i] = masses[0, 0, 0]
    val_4[i] = masses[4, 0, 0]

plt.plot(xpositions, val_0, label='Mass in cell 0')
plt.plot(xpositions, val_4, label='Mass in cell 4')
plt.xlabel('x position of particle (Mpc)')
plt.ylabel('Mass')
plt.xlim(0, 16)
plt.legend()
plt.title('Cloud In Cell assignment')
plt.savefig('plots/ex5_CIC_xpos.pdf')
plt.close()
```

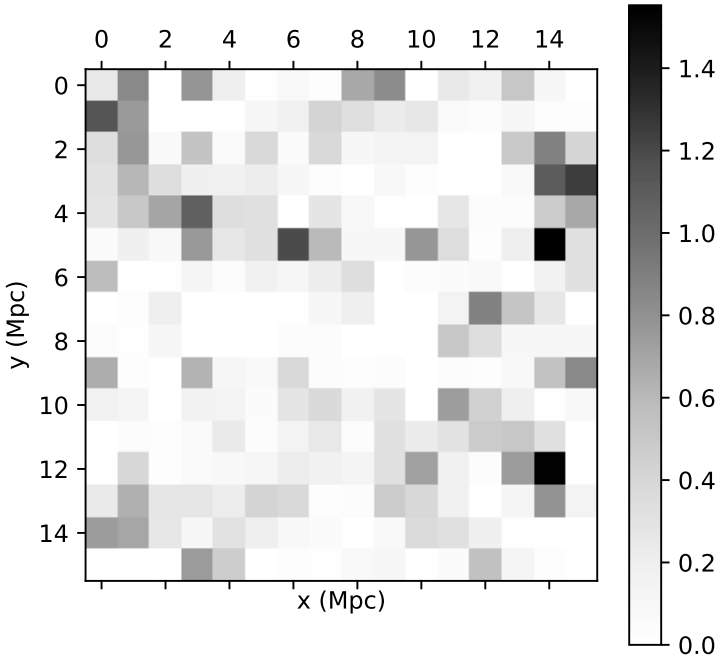

Figure 23: x-y slice of CIC mass assignment with $z = 4$

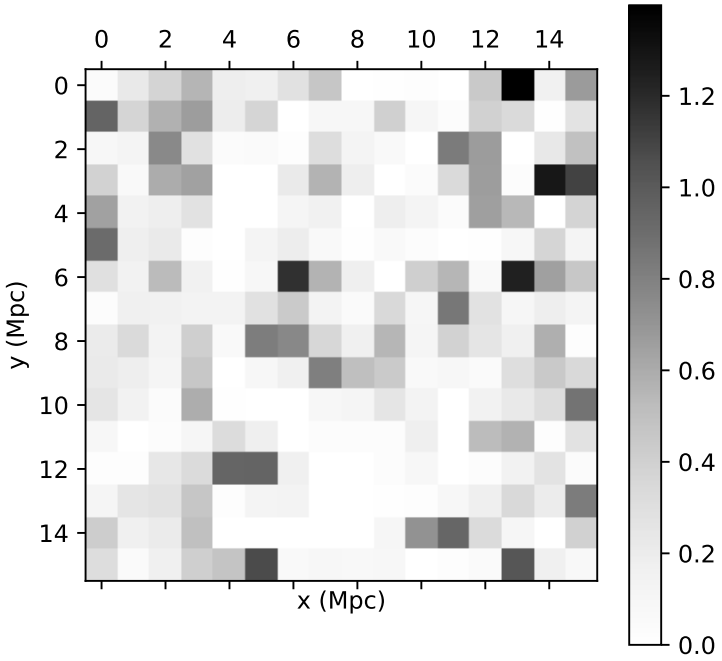

Figure 24: x-y slice of CIC mass assignment with $z = 9$

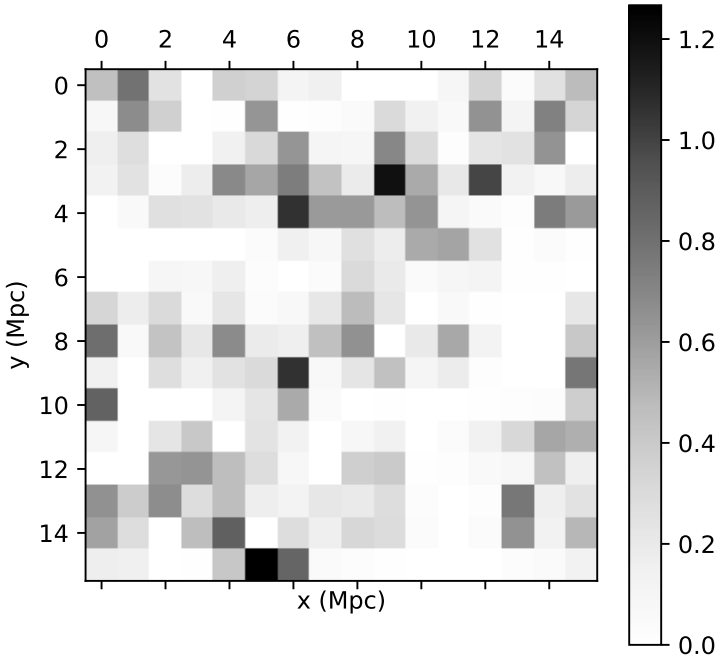

Figure 25: x-y slice of CIC mass assignment with $z = 11$

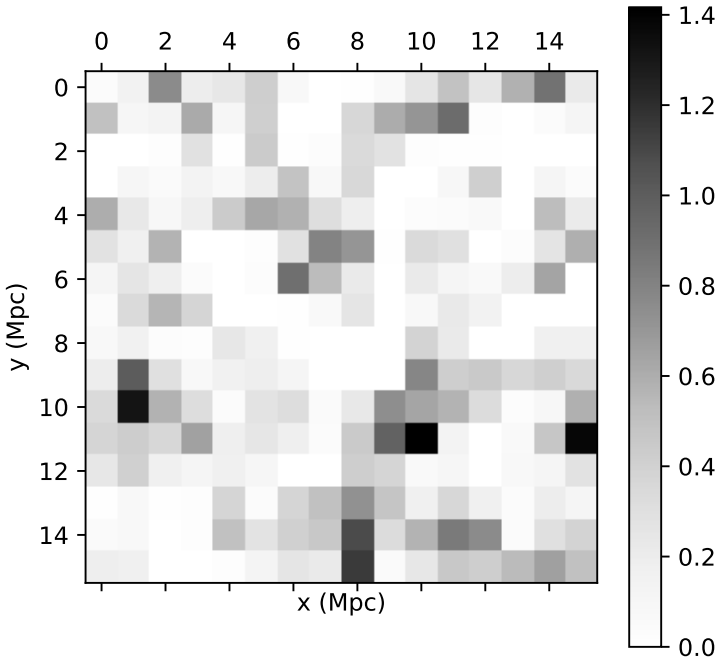

Figure 26: x-y slice of CIC mass assignment with $z = 14$

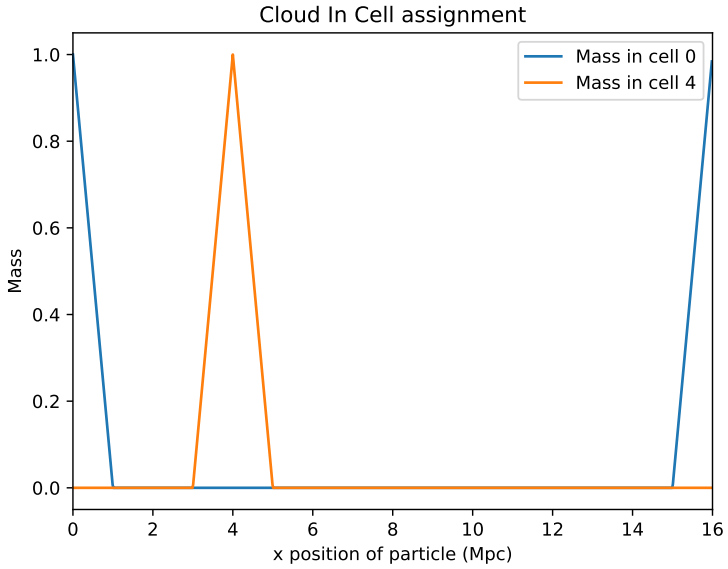

Figure 27: Values of mass in shown cells for one particle in CIC simulation with x-coordinate as shown

5d The implementation of a 1D FFT is shown below.

```
def bit_reverse(data, lg2_N):
    # Bit reverses elements of data, which should be of size 2**lg2_N
    index = np.zeros(1, dtype=np.int64)
    for i in range(lg2_N):
        # A beautiful result from wikipedia:
        # We can double the k-1 bits permutation, concatenate
        # with 1 added to it and get the k bits permutation
        # https://en.wikipedia.org/wiki/Bit-reversal_permutation
        index = np.concatenate((index*2, index*2 +1))
    # now rearrange the data array
    for i, j in enumerate(index):
        if i > j:
            temp = data[i]
            data[i] = data[j]
            data[j] = temp

@njit
def fft(data, sign=-1):
    '''
    In-place DFT calculation, len(data) should be a power of 2
    Note that NR and slides use sign=1,
    FFTW and FFTPACK (and thus numpy and scipy) use sign=-1
    Flipping sign gives the inverse transform up to a 1/n normalization
    Based on Introduction to Algorithms by CLRS 3rd ed. 30.3
```

```
    '''
    N = data.size
    # If N is a power of 2, it has form 1000...,
    # N - 1 has the form 0111...
    # Thus N & (N - 1) == 0 iff N a power of 2
    if N & (N - 1) != 0:
        raise ValueError('Data size must be a power of 2')
    # Note that powers of 2 are exact for floats, so no need to round
    lg2_N = int(np.log2(N))
    bit_reverse(data, lg2_N)
    for s in range(1, lg2_N + 1):
        m = 2**s
        # Calculate principal mth root of unity
        omega_m = np.exp(sign * 2j * np.pi / m)
        for k in range(0, N, m):
            # omega is multiplied by omega_m cycling through
            # the required factors which are all mth roots of unity
            omega = 1
            for j in range(m//2):
                # Apply the Danielson-Lanczos update
                t = omega * data[k + j + m//2]
                u = data[k + j]
                data[k + j] = u + t
                data[k + j + m//2] = u - t
                # Calculate next factor omega
                omega = omega * omega_m
```

We test the FFT both analytically and by comparing to FFPACK using a checkerboard pattern. This corresponds to a wave sampled at the Nyquist Frequency.

```
fs = np.ones(32)
fs[1::2] = -1

fs_myfft = np.array(fs, copy=True, dtype=np.complex128)
fft.fft(fs_myfft)
fs_npfft = np.fft.fft(fs)
if np.allclose(fs_myfft, fs_npfft):
    print('1D FFT is equal to FFTPACK up to machine precision')
else:
    print('Error in 1D FFT')

plt.plot(fs_myfft.real, label='Real part of FT')
plt.plot(fs_myfft.imag, label='Imaginary part of FT')
plt.plot(fs, label='Function values')
plt.xlim(0, 31)
plt.legend()
plt.savefig('plots/ex5_FFT1.pdf')
plt.close()
```

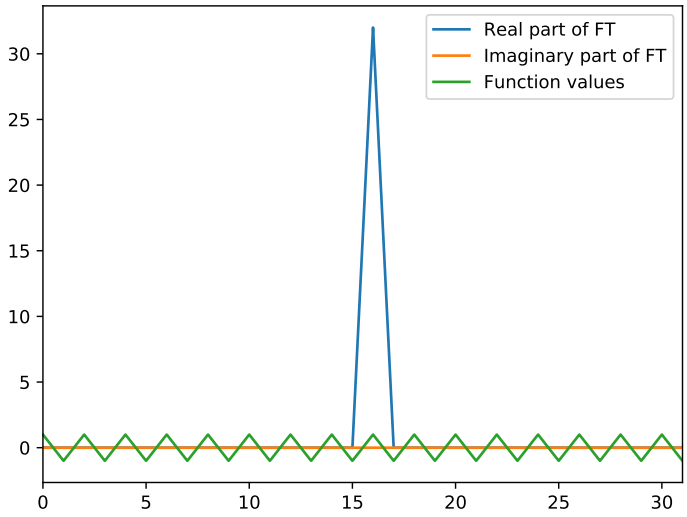

Figure 28: Numerical results for my FFT and numpy FFT for a checkerboard pattern. The results agree within numerical precision.

5e We also implement an FFT in 2 and 3 dimensions.

```
@njit
def fft3(data, sign=-1):
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
            fft(data[i, j, :], sign)
    for i in range(data.shape[1]):
        for j in range(data.shape[2]):
            fft(data[:, i, j], sign)
    for i in range(data.shape[2]):
        for j in range(data.shape[0]):
            fft(data[j, :, i], sign)
@njit
def fft2(data, sign=-1):
    for i in range(data.shape[0]):
        fft(data[i, :], sign)
    for j in range(data.shape[1]):
        fft(data[:, j], sign)
```

We test the 2D FFT on a checkerboard pattern and the 3D FFT on a multivariate gaussian.

```
fs = np.ones((32, 32))
fs[1::2, ::2] = -1
fs[::2, 1::2] = -1
```

```
fs_myfft = np.array(fs, copy=True, dtype=np.complex128)
fft.fft2(fs_myfft)
fs_npfft = np.fft.fft2(fs)

if np.allclose(fs_myfft, fs_npfft):
    print('2D FFT is equal to FFTPACK up to machine precision')
else:
    print('Error in 2D FFT')

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.matshow(fs, cmap='Greys')
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Function values')

ax2.matshow(fs_myfft.real, cmap='Greys')
ax2.set_xlabel('k_x')
ax2.set_ylabel('k_y')
ax2.set_title('Fourier Transform')
plt.tight_layout()
plt.savefig('plots/ex5_FFT2.pdf')
plt.close()

xs = np.arange(-16, 16)
ys = np.arange(-16, 16)
zs = np.arange(-16, 16)
rs_2 = (xs**2)[:, None, None] + (ys**2)[None, :, None] + (zs**2)[None, None, :]

fs = np.exp(-1 * rs_2/8)

fs_myfft = np.array(fs, copy=True, dtype=np.complex128)
fft.fft3(fs_myfft)
fs_npfft = np.fft.fftn(fs)

if np.allclose(fs_myfft, fs_npfft):
    print('3D FFT is equal to FFTPACK up to machine precision')
else:
    print('Error in 3D FFT')
# Shift FFT so DC term is at center
fs_myfft = np.roll(fs_myfft, 16, axis=(0, 1, 2))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3)

ax1.matshow(np.abs(fs_myfft[:, :, 16]))
ax1.set_title('x-y slice')
ax2.matshow(np.abs(fs_myfft[:, 16, :]))
ax2.set_title('x-z slice')
ax3.matshow(np.abs(fs_myfft[16, :, :]))
ax3.set_title('y-z slice')
```

```
plt.savefig('plots/ex5_FFT3.pdf')
plt.close()
```

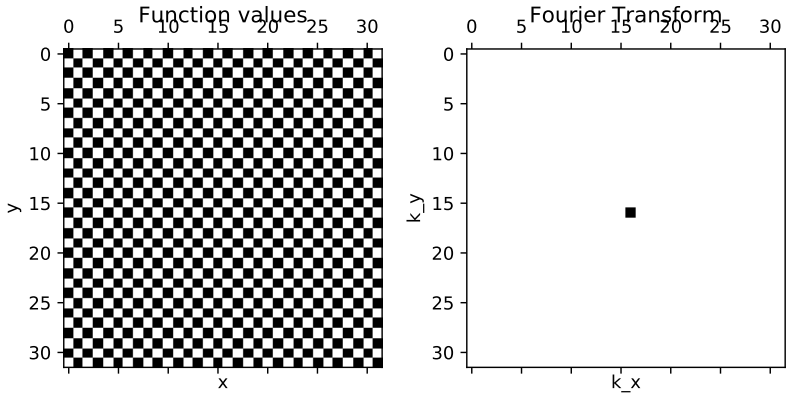

Figure 29: Numerical results for my FFT in 2 dimensions for a checkerboard pattern. The results have been checked to agree within numerical precision.

5f We can calculate the gravitational potential using a Fourier Transform.

```
def mesh_to_potential(masses):
    grid_size = masses.shape[0]
    overdensity = (masses - masses.mean()) / masses.mean()
    potential = np.array(overdensity, copy=True, dtype=np.complex128)
    fft.fft3(potential)
    ks_1D = fft.fftfreq(grid_size)
    ks_square = (ks_1D.reshape(grid_size, 1, 1)**2 +
                 ks_1D.reshape(1, grid_size, 1)**2 +
                 ks_1D.reshape(1, 1, grid_size)**2)
    ks_square[0, 0, 0] = 1
    potential /= ks_square
    fft.fft3(potential, sign=1)
    potential = potential / (grid_size**3)
    return potential.real

masses = CIC_mass_assignment_3D(positions.T)
potential = mesh_to_potential(masses)

for z in (4, 9, 11, 14):
    plt.matshow(potential[:, :, z])
```

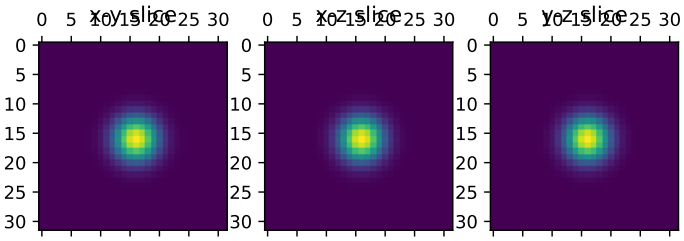

Figure 30: Numerical results for my FFT in 3 dimensions for a 3D gaussian.

```
    plt.xlabel('x (Mpc)')
    plt.ylabel('y (Mpc)')
    plt.savefig(f'plots/ex5_pot_z{z}.pdf')
    plt.close()

plt.matshow(potential[:, 8, :])
plt.xlabel('x (Mpc')
plt.ylabel('z (Mpc')
plt.savefig('plots/ex5_pot_xz.pdf')

plt.matshow(potential[8, :, :])
plt.xlabel('y (Mpc')
plt.ylabel('z (Mpc')
plt.savefig('plots/ex5_pot_yz.pdf')
```

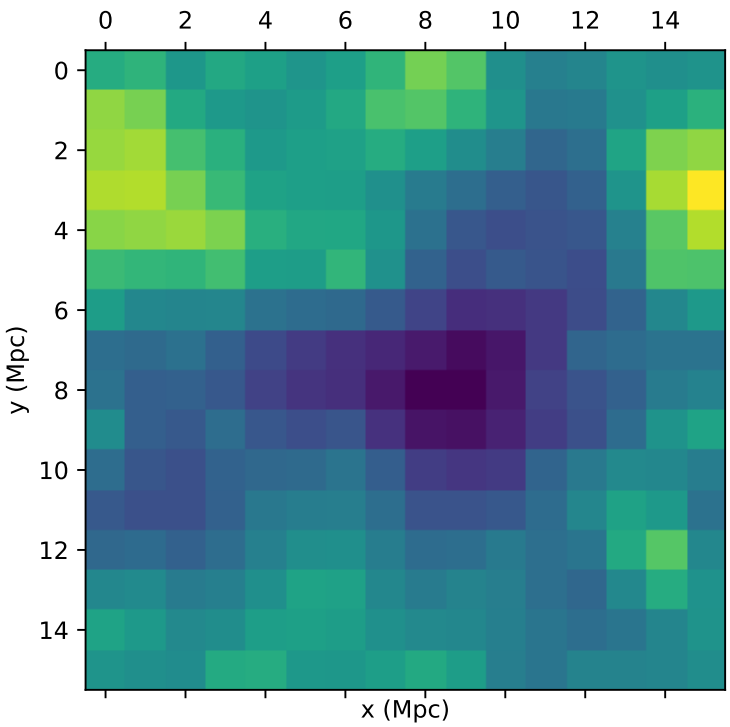

Figure 31: xy Potential at z=4

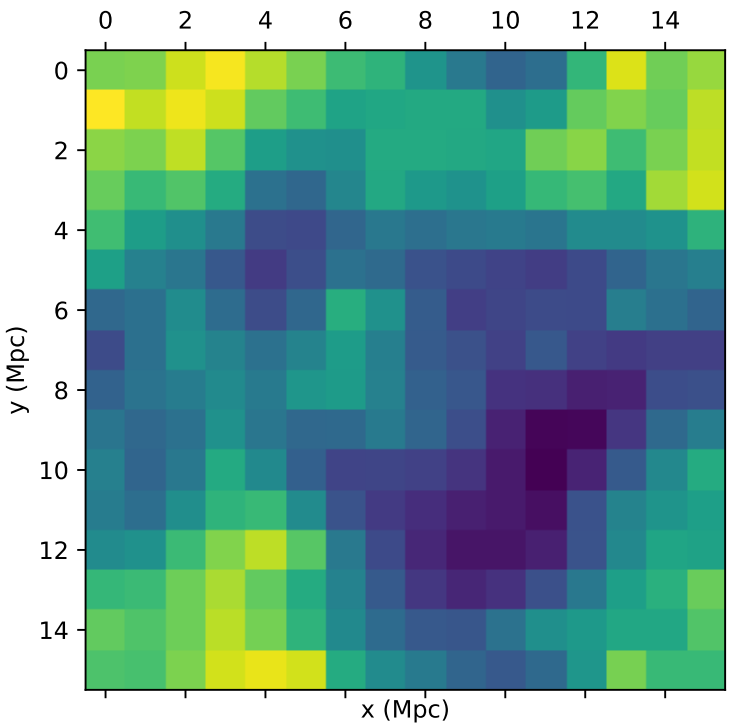

Figure 32: xy Potential at z=9

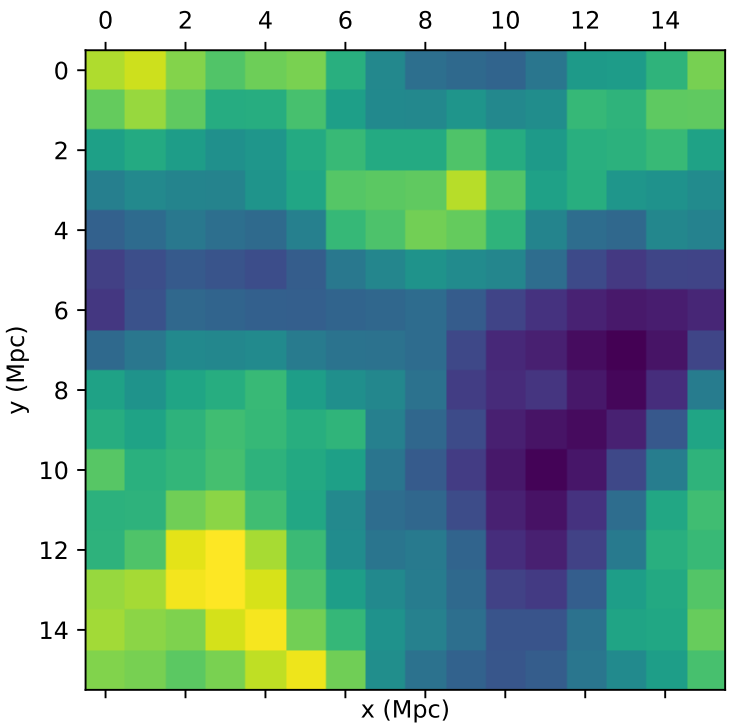

Figure 33: xy Potential at z=11

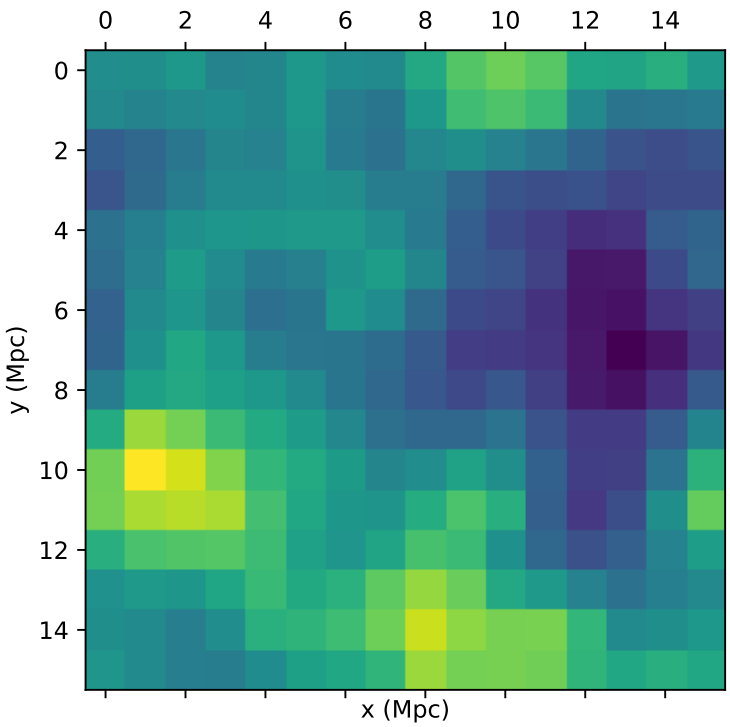

Figure 34: xy Potential at z=14

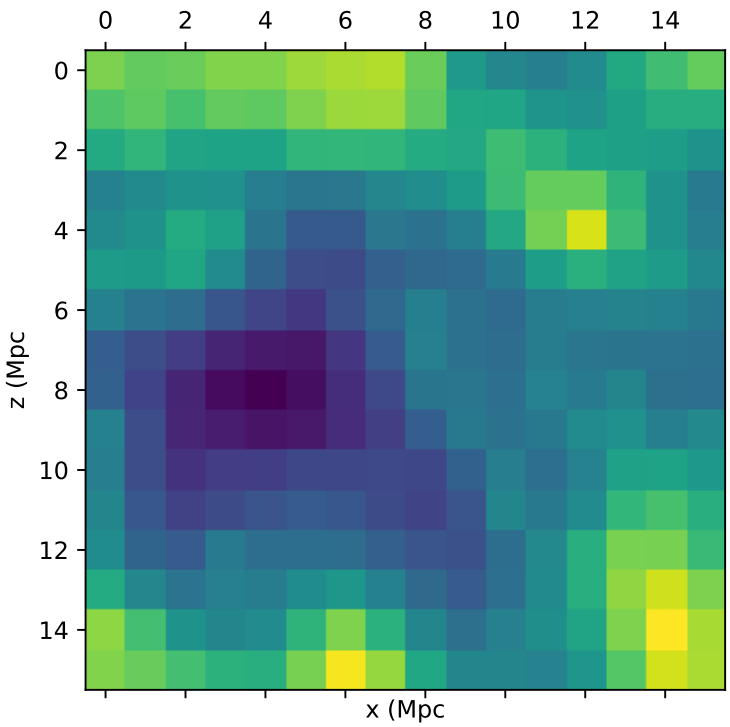

Figure 35: xz Potential at y=8

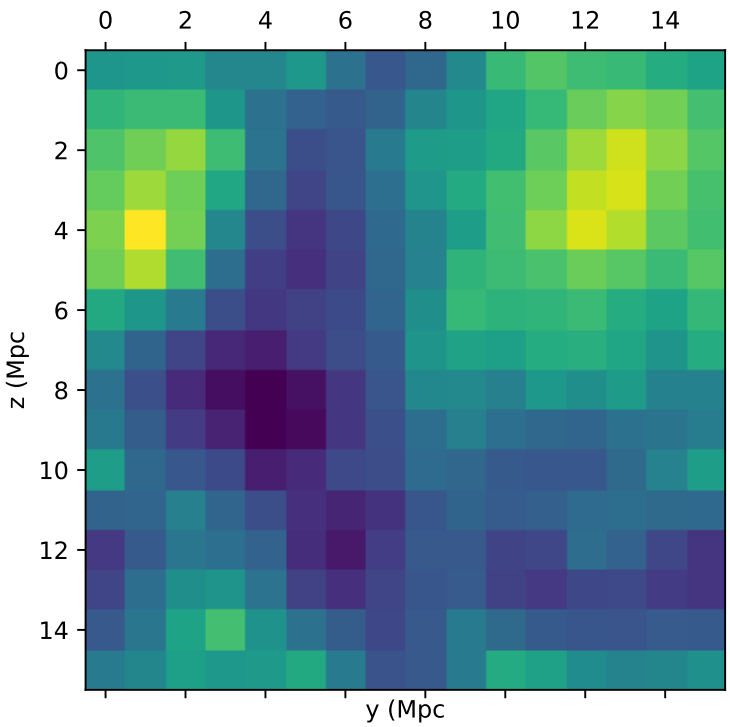

Figure 36: yz Potential at x=8