Final Exercise

Deadline: May 28, 15:30

Read these instructions carefully before you start coding.

This second hand-in exercises covers material from **all lectures**. Unless noted otherwise, you are expected to code up your own routines using the algorithms discussed in class and **cannot use special library functions** (except exp()). Extremely simple functions like `arange`, `linspace` or `hist` (without using advanced features) are OK. When in doubt, write your own, or ask us. You can use the routines you wrote for the tutorials and for hand-in exercise 1, however, your routines must be written **by yourself**. Codes will be checked for plagiarism (with other students as well as other sources), routines which are too similar get **zero points**.

To hand in your solutions, put your code up on an otherwise empty **GitHub repository** and share it with us (`daalen`, `fonotec` and `syeda-lammim-ahad` – no commits after the deadline!). For every main question (1 through 9) you should write a **separate program**. We **must** be able to run everything with **a single call to a script** `run.sh`, which downloads any data (data needed for an exercise **may not** be included in your repository but needs to be retrieved at runtime), runs your scripts and generates a PDF containing all your source code and outputs **in the following format**:

- Per main question, the code of any shared modules.
- Per sub-question, an explanation of what you did.
- Per sub-question, the code specific to it.
- Per sub-question, the output(s) along with discussion/captions.

Furthermore `run.sh` should be **in the top folder**, so we should be able to run the script by doing `git clone <repository>` followed by `cd <repository>` and `./run.sh`. Have a fellow student test this! Exercises that are not run with this single command or do not have their code and output in the PDF get **zero points** (this includes solutions in Jupyter notebooks).

Ensure that your code runs to completion on the `pczaal` computers using `python3`(!). Codes that do not get **zero points**. Your code should have a total run time of **at most 10 minutes**, solutions generated will not be checked beyond this limit. If, **during testing**, a part of your code takes long to run, remember that you can run it once and read in its output (saved to file) in the rest of your code – however, the rules as stated above still apply to whatever you hand in at the end (everything run with a single command, all outputs produced on the fly, total runtime limit, etc).

For all routines you write, **explain** how they work in the comments of your code and **argue** your choices! Similarly, whenever your code outputs something **clearly indicate** next to the output/in the PDF what is being printed. This includes discussing your plots in their captions.

If a part of your code **does not run**, explain what you did so far and what the problem was and still include that part of the code in the PDF for possible partial credit. If you are unable to get a routine to work but you need it for a follow-up question, use a library routine in the follow-up (and clearly indicate that and why you do so).

1. **Normally distributed pseudo-random numbers**

   (a) Write a random number generator that returns a random floating-point number between 0 and 1. At minimum, use some combination of an MWC and a 64-bit XOR-shift.[1] We'll test its quality: like in the tutorial, have your code plot sequential random numbers against each other in a scatter plot (x_{i+1} vs x_i) for the first 1000 numbers generated. Also plot the value of the random numbers for the first 1000 numbers vs the index of the random number, this means that the x-axis has a value from 0 through 999 (and the y-axis 0 through 1). Finally, have your code generate 1 000 000 random numbers and plot the result of binning these in 20 bins 0.05 wide.

[1] To be clear: whatever your RNG looks like, it should include **both** an MWC and a 64-bit XOR-shift.

Change your RNG parameters if necessary. Make sure you use a fixed seed and set it only once for your entire program. The seed value should be the first output when we run your code for this question.[2]

(b) In several exercises we will need random variables distributed normally. To do this we are going to use the Box-Muller method. But first let's take a look at the Gaussian integral; in general the analytical solution of the Gaussian integral is given by:

$$\int_{-\infty}^{\infty} \mathrm{d}x \exp\left(-\alpha x^2\right) = \sqrt{\frac{\pi}{\alpha}}. \tag{1}$$

This integral can be used to normalize the Gaussian integral. Solve this integral using numerical integration for $\alpha = 2^i$ in which $i = (-10, -9, \ldots, 10)$. Numerically fit a line to the 21 points and show that your fit is sufficiently close to the analytical solution, and make a plot showing the analytical solution and your best fit. Also output your final fit parameters and the analytical fit parameters. We require a relative accuracy of at least 10^{-4} for the final fit parameters.

(c) Now use the Box-Muller method (see lecture) to generate 1000 normally-distributed random numbers. To check if they are following the expected Gaussian distribution, make a histogram (scaled appropriately) with the corresponding true probability distribution (normalized to integrate to 1) as a line. This plot should contain the interval of -5σ until 5σ from the theoretical probability distribution. Indicate the theoretical 1σ, 2σ, 3σ and 4σ intervals with a line. For this plot, use $\mu = 3$ and $\sigma = 2.4$, and choose bins that are appropriate.

(d) Write a code that can do the KS-test on the your function to determine if it is consistent with a normal distribution. For this, use $\mu = 0$ and $\sigma = 1$. Make a plot of the probability that your Gaussian random number generator is consistent with Gaussian distributed random numbers, start with 10 random numbers and use in your plot a spacing of 0.1 dex until you have calculated it for 10^5 random numbers on the x-axis. Compare your algorithm with the KS-test function from `scipy`, `scipy.stats.kstest` by making an other plot with the result from your KS-test and the KS-test from `scipy`.

(e) Write a code that does the Kuiper's test on your random numbers (see tutorial 8) and make the same plot as for the KS-test.

(f) Download a dataset using `wget https://home.strw.leidenuniv.nl/~nobels/coursedata/randomnumbers.txt`.[3] This dataset contains 10 sets of random numbers. Compare these 10 sets with your Gaussian pseudo random numbers ($\mu = 0$, $\sigma = 1$) and make the same plot of the probability as in either of the previous two exercises (your choice).[4] Also make two plots of the first and second moment as a function of the amount of random numbers to check if the random numbers are consistent. Add in both plots the expected first and second moments. Which random number array(s) is/are consistent with Gaussian random numbers with $\sigma = 1$ and $\mu = 0$?

(g) Besides making a Gaussian random number generator that uses the Box-Muller method, we will also look into a method that uses a different technique. Let's make Gaussian random numbers using slice sampling using root finding. Make a histogram with 10^5 Gaussian random numbers generated in this way with the corresponding true probability distribution over-plotted as a line. Also make a plot of the first 1000 random numbers of this method versus the first 1000 random numbers for the Box-Muller method and add the theoretical contours of a bivariate normal distribution (1σ, 2σ, 3σ) to check if your results are correct. In this exercise you are only allowed to use root finding once per random number.[5]

(h) In the case of a Gaussian probability distribution it is not necessary to use a root finding method, make another implementation of the slice sampling method but this time without using root

[2] You will use this random number generator code during a lot of the questions in this exercise, it is allowed to set a new seed in an independent program.

[3] To reiterate, datasets like this should be downloaded when we run your script and are not allowed to be in your Github repository.

[4] E.g. if you use 10 numbers use 10 numbers from the dataset and from your own random number generator.

[5] Any root finding method is allowed if the result is accurate enough.

finding. Again make a histogram with the corresponding true probability distribution as a line for 10^5 Gaussian random numbers. Again make the plots from the previous question for the first 1000 numbers of this method.

2. **Making an initial density field**
Now that we made a good random number generator we can start producing a density field of a cosmological simulation. Generating a Gaussian random field can be easily done in Fourier space.[6] In this case the complex Fourier amplitudes are given by $\tilde{Y} = |\tilde{Y}| \exp(i\phi)$, where ϕ is a random phase and the modules are Rayleigh distributed:

$$f(x) = \frac{2x}{\sigma^2} e^{-\frac{x^2}{\sigma^2}}. \tag{2}$$

The dispersion is trivially know to be given by:

$$\sigma^2 = P(k). \tag{3}$$

This means that the dispersion depends on the wave number.

(a) The first part in this exercise is generating a Fourier plane. This can be done by generating a 2D matrix (a grid) with complex entries, of which the real and imaginary parts may just be normal random numbers with the variance given by the power spectrum. The power spectrum has the following functional form:

$$P(k) \propto k^n. \tag{4}$$

The matrix that is constructed is required to have certain symmetries such that after the inverse Fourier transform the matrix is real. This therefore means that we have the symmetry $\tilde{Y}(-k) = \tilde{Y}^*(k)$. Using this information you can make a Gaussian random field. Make plots of three Gaussian random fields (use `imshow` or `matshow`), using $n = -1$, $n = -2$ and $n = -3$. Give the plots a size of 1024 x 1024 pixels, the axis should be in physical size (e.g. `Mpc`). Choose a minimum physical size and explain how this impacts the maximum physical size, the minimum k and maximum k.[7]

(b) In the case of ΛCDM the power spectrum is given by:

$$P_{\mathrm{CDM}}(k) \propto \frac{k^n}{\left(1 + 3.89q + (16.1q)^2 + (5.46q)^3 + (6.71q)^4\right)^{1/2}} \times \frac{\left(\ln\left(1 + 2.34q\right)\right)^2}{(2.34q)^2}. \tag{5}$$

Here $q = k/\Gamma$ and

$$\Gamma = \Omega_m \exp\left(-\Omega_b - \frac{\Omega_b}{\Omega_m}\right). \tag{6}$$

In the case of ΛCDM the cosmological parameters are given by $\Omega_b = 0.0486$, $\Omega_m = 0.3089$ and $\Omega_\Lambda = 0.6911$. Again make a plot of size 1024 x 1024, for this use as minimum scale of 1 Mpc, the axis should be in physical units (e.g. `Mpc`), think again about how the physical size transforms to k.

(c) Now make a Gaussian random field in 3 dimensions to set up the initial conditions for a cosmological simulation. For this use a power law power spectrum of $n = -2$. This grid only needs to have the size of 64^3. Make plots of $x - y$ slices with z values of 10, 25, 40 and 53 as a test.

3. **Linear structure growth**
A tool often used to understand how the Universe started to grow is linear perturbation theory, the most important result of this analysis is that the evolution of density perturbations in the initial universe evolves according to the following equation:

$$\frac{\partial^2 \delta}{\partial t^2} + 2\frac{\dot{a}}{a}\frac{\partial \delta}{\partial t} = \frac{3}{2}\Omega_0 H_0^2 \frac{\delta}{a^3}. \tag{7}$$

[6] It is not required to make your own FFT in this exercise, you can use an FFT from a `Python` package. In exercise 5 you will make your own FFT

[7] The normalization of the power spectrum is not important for this exercise, you can take that we have $P(k) = k^n$.

This equation describes the linear growth of density perturbations. In the early Universe we can separate the density perturbation as having a spatial part and a temporal part: $\delta = D(t)\Delta(\mathbf{x})$. In the case of a second order equation we have two growth factors. This means that the above partial differential equation becomes:

$$\frac{\mathrm{d}^2\delta}{\mathrm{d}t^2} + 2\frac{\dot{a}}{a}\frac{\mathrm{d}D}{\mathrm{d}t} = \frac{3}{2}\Omega_0 H_0^2 \frac{1}{a^3} D. \tag{8}$$

This equation is called the linearized density growth equation.

(a) Let's look at a matter-dominated Einstein-de Sitter Universe. An Einstein-de Sitter Universe is given by purely matter $\Omega_m = 1$, the scale factor of such an Universe is given by:

$$a(t) = \left(\frac{3}{2}H_0 t\right)^{2/3}. \tag{9}$$

Write down the linearized density growth equation for an Einstein-de Sitter Universe and calculate the numerical solution with the following initial conditions:

	$y(1)$	$y'(1)$
case 1	3	2
case 2	10	-10
case 3	5	0

Solve the ODE for these 3 initial conditions using an appropriate numerical method. Compare the result with the analytical solutions of the ODE. Plot the solution for $t = 1$ until $t = 1000$, use a log-log plot.

(b) Another extreme is an empty Universe which is given by $\Omega_m = 0$, the scale factor of this Universe is given by:

$$a(t) = H_0 t. \tag{10}$$

Calculate the linearized density growth equation for an empty Universe and calculate the numerical solution with the following initial conditions:

	$y(1)$	$y'(1)$
case 1	10	0
case 2	10	-10
case 3	15	-5

Solve the ODE for these 3 initial conditions using an appropriate numerical method. Compare the result with the analytical solutions of the ODE. Plot the solution for $t = 1$ until $t = 1000$, use a log-log plot.

(c) Most of the solutions in (a) and (b) seem to be close to a power law. Determine the best-fit power law that passes through your numerical solution of the differential equations. Make two tables, one table for the fits for (a) with the normalization and the slope for all 3 cases and a table for the fits of (b) with the normalization and slope for all 3 cases. Also make two plots for the fits of (a) with the numerical solution and the fits of (b) with the numerical solution.

4. **Zeldovich approximation**

In this part we will set up the initial conditions using something called the Zeldovich approximation. The Zeldovich approximation is a first order linear Lagrangian structure formation model that can be used to set up initial conditions for a cosmological simulation. In its simplest form the Zeldovich approximation is given by:

$$\mathbf{x}(t) = \mathbf{q} + D(t)\mathbf{S}(\mathbf{q}). \tag{11}$$

At first this may look complicated, but this equation can basically be described as $\mathbf{x}(t) = \mathbf{x}_0 + \mathbf{v}t$ with constant $\mathbf{v}$. This means that $\mathbf{q}$ is the initial position of a matter particle, $\mathbf{x}(t)$ is the position at time t, $D(t)$ is the linear growth function and $\mathbf{S}(\mathbf{q})$ is a time-independent displacement vector. In this exercise we will use the scale factor (a), redshift (z) and time interchangeable as a time variable. For this exercise it is not relevant how a and z relate with t, but it is important to realize that

$a = 1/(1+z)$. Continuing on the Zeldovich approximation, this allows us to calculate the initial conditions at expansion factor a as:

$$\mathbf{x} = \mathbf{q} + D(a)\mathbf{S}(\mathbf{q}), \quad \mathbf{p} = -(a - \Delta a/2)^2 \dot{D}(a - \Delta a/2)\mathbf{S}(\mathbf{q}), \tag{12}$$

in which $\mathbf{x}$ is the position of your particles and $\mathbf{p}$ is the momentum of your particles. In this case the displacement vector $\mathbf{S}$ is given by the FFT:

$$\mathbf{S}(\mathbf{q}) = \alpha \sum_{k_x=-k_{\max}}^{k_{\max}} \sum_{k_y=-k_{\max}}^{k_{\max}} \sum_{k_z=-k_{\max}}^{k_{\max}} i\mathbf{k} c_k \exp\left(i\mathbf{k}\cdot\mathbf{q}\right). \tag{13}$$

Here $k_{\mathrm{x,y,z}} = \frac{2\pi}{N_{\mathrm{g}}} l, m, n$ with N_{g} the number of grid points along each dimension, $l, m, n = 0, \pm 1, \ldots, N_{p,1}/2$ and $k^2 = k_x^2 + k_y^2 + k_z^2 \neq 0$. Furthermore in this equation α is the power spectrum normalization and c_k random numbers. These are given by $c_k = (a_k - ib_k)/2$ in which a_k and b_k are generated as:

$$a_k = \sqrt{P(k)}\frac{\mathrm{Gauss(0,1)}}{k^2} \neq b_k = \sqrt{P(k)}\frac{\mathrm{Gauss(0,1)}}{k^2}. \tag{14}$$

Note that as in the case of the Gaussian random fields the random numbers should satisfy $c_k = c^*_{-k}$ because the FFT is real.
Important: during this exercise we will assume periodic boundary conditions. This means that when a particle moves outside of the simulation box it should enter through the other side of the box.

(a) The first part is calculating the linear growth factor until the desired redshift, $z = 50$. The linear growth factor is expressed in terms of a integral expression given by:[8]

$$D(z) = \frac{5\Omega_{m,0}H_0^2}{2} H(z) \int_z^\infty \frac{1+z'}{H^3(z')} \mathrm{d}z'. \tag{15}$$

Here z is the redshift, $\Omega_{m,0}$ is the matter fraction of the Universe at $z = 0$ ($\Omega_{m,0} = 0.3$), H_0 is the Hubble constant at $z = 0$ and $H(z)$ is the redshift dependent redshift given by:

$$H(z)^2 = H_0^2\left(\Omega_{m,0}(1+z)^3 + \Omega_\Lambda\right). \tag{16}$$

Here Ω_Λ is the dark energy fraction of the Universe given by $\Omega_\Lambda = 0.7$. Calculate the growth factor at $z = 50$ with a relative accuracy of 10^{-5}.

(b) We also want to calculate the derivative at $z = 50$ in order to be able to calculate the momentum of the particles. In order to calculate the time derivative of the linear growth factor it is important to realize that we cannot easily calculate it directly, but need to calculate it indirectly as:

$$\dot{D}(t) = \frac{\mathrm{d}D}{\mathrm{d}a}\dot{a}, \tag{17}$$

in which we know that $\dot{a}(z) = H(z)a(z)$. Numerically calculate the derivative of equation (15) (after rewriting it in terms of a) with a relative accuracy of 10^{-5} and compare your result with the analytical derivative (and as always, show how you found the analytical derivative).

(c) Use the Zeldovich approximation to generate a movie of the evolution of a volume in two dimensions from a scale factor of 0.0025 until a scale factor of 1.0. Use 64 x 64 particles in a square grid. Your movie should contain at least 30 frames per seconds and should take at least 3 seconds.

(d) Generate initial conditions for a three dimensional box to do a N-body simulation, make initial conditions for 64^3 particles starting at redshift $z = 50$. Besides this make 3 separate movies of a slice of thickness 1/64th of your box at its center, make a slice for $x - y$, $x - z$ and $y - z$. Again make a movie of at least 3 seconds with at least 30 frames per second. Remember: slice, not projection!

[8] Reminder: a, z and t can be used interchangeably in this exercise, and it is not relevant to know how a and z are related to t, only the relation between a and z is relevant: $a = 1/(1+z)$.

5. **Mass assignment schemes**
Our simple cosmological N-body simulation is going to use the particle mesh algorithm to calculate a density mesh which we are going to use to calculate the gravitational forces in our simulation. To do this we need to assume a certain size, mass, shape and internal density for each particle. This allows us to determine the appropriate interpolation scheme which we need to use to assign densities to the grid cells. We define the 1D particle shape as $S(x)$ to be the mass density at the distance x from the particle for cell size Δx. In this exercise we will use periodic boundary conditions.

(a) The most simple choice for the particle shape is a point like shape given by:

$$S(x) = \frac{1}{\Delta x}\delta\left(\frac{x}{\Delta x}\right). \tag{18}$$

Explain how we need to assign mass in this scheme and explain why this method is called the Nearest Grid Point (NGP) method. Code up your own implementation of the mass assignment scheme NGP, using a grid of 16^3. Use `np.random.seed(121)` and display $x-y$ slices with z values of 4, 9, 11 and 14. Generate the positions of the test particles as:

```
#!/usr/bin/env python3
import numpy as np
np.random.seed(121)
positions = np.random.uniform(low=0,high=16,size=(3,1024))
```

(b) To check the robustness of your implementation make a plot of the x position of an individual particle and the value in cell 4 in 1 dimension and let x vary from the lowest value to the highest possible value in x. Repeat for cell 0.

(c) Right now we want to improve this method, because NGP has several disadvantages. For this we are going to use a method which assumes that particles are cubes in 3D of uniform density and have the size of a grid cell. This means that the particle shape is given by:

$$S(x) = \begin{cases} 1, & \text{if } |x| < \frac{1}{2}\Delta x \\ 0, & \text{otherwise.} \end{cases} \tag{19}$$

Calculate how mass needs to be assigned in the case of this other method called the Cloud In Cell (CIC) method and implement it in code. To check the robustness of your implementation again make the same plots as before for seed `np.random.seed(121)` (see questions (a) and (b)).

(d) An even more improved shape for the mass assignment is using the Triangular Shaped Cloud (TSC) method, which uses a shape function given by:

$$S(x) = \begin{cases} 1 - \frac{|x|}{\Delta x}, & \text{if } |x| < \Delta x \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

Implement this function and do the same tests for this method as in the previous methods.

(e) Write your own FFT algorithm, check that your code works with a 1D function (no Gaussian) by making a plot of the FFT and compare your result with a python package and the analytical FFT of your function. In the rest of this exercise you need to use your own FFT.

(f) Generalize your own FFT algorithm to 2 and 3 dimensions and make a plot of the FFT of a 2D function (no bivariate Gaussian) and compare it with the analytical FFT of your FFT. Also make a plot of the FFT of a 3D multivariate Gaussian function, plot 3 slices centered at the center for the 3 different slice options $x-y$, $x-z$ and $y-z$.

(g) Consider again the Cloud In Cell method, before this we calculated the mesh, reduce the values of the mesh by subtracting the average and dividing by the average as $\delta = (\rho - \langle\rho\rangle)/\langle\rho\rangle$. We know that in general gravity is described by the Poisson equation which can be expressed as $\nabla^2\Phi \propto \delta$, in this case it is easier to calculate the FFT of the potential and transform it back:

$$\nabla^2\Phi \propto \delta \xrightarrow{\text{FFT}} k^2\tilde{\Phi} \propto \tilde{\delta} \xrightarrow{\text{rewrite}} \tilde{\Phi} \propto \frac{\tilde{\delta}}{k^2}. \tag{21}$$

After this the inverse FFT is taken and the potential is determined. Calculate the potential up to a constant for the same particles. Again make a plot of the potential of $x-y$ slices with z values of 4, 9, 11 and 14. Also make a centered slice for the $x-z$ and $y-z$ plane.

(h) In simulations we often want to know the spatial gradient of the potential, use your potential field to calculate the gradient of the the potential for the first 10 particles. Output the value of the gradient of the potential for these particles in all 3 spatial coordinates. It is important to realize that when assigning inferred quantities for particles they should be assigned with the same weighting as in the case of assigning the mass to the grid. Again use CIC.

6. **Halo mass function**
One of the interesting properties of a simulation is the distribution of halo masses in the simulation. In general the number of dark matter haloes is given by:

$$\frac{\mathrm{d}n(M,t)}{\mathrm{d}M} = f(\nu)\frac{\langle\rho\rangle}{M^2}\left|\frac{\mathrm{d}\ln\nu}{\mathrm{d}\ln M}\right|\mathrm{d}M. \tag{22}$$

Here $\langle\rho\rangle$ is the average matter density of the universe, and $\nu = \delta_c/\sigma$ is the peak height in which $\delta_c = 1.686$. Furthermore the functional from is given by:

$$f(\nu) = A\sqrt{\frac{2a}{\pi}}\nu\left(1+\frac{1}{a\nu^{2q}}\right)\exp\left(-\frac{\nu^2}{2}\right). \tag{23}$$

Here $A = 0.3222$, $a = 0.707$ and $q = 0.3$. Finally, the rms density fluctuation is approximately given by:

$$\sigma(M) = \sigma_8\left(\frac{M}{M_8}\right)^{-\alpha}, \tag{24}$$

in which σ_8 is the fluctuation at 8 Mpc given by $\sigma_8 = 0.82$ and M_8 is the average mass in a 8 Mpc sphere approximately given by $M_8 = 9\cdot10^{14}$ $\mathrm{M}_\odot$, lastly $\alpha \approx 1/4$.

(a) Make a log-log plot of single points for $\frac{\mathrm{d}n}{\mathrm{d}\log M}$ at masses of $M = 10^{10}$ $\mathrm{M}_\odot$ with spacing of 0.25 dex until $M = 10^{15.5}$ $\mathrm{M}_\odot$. Interpolate the values in between based on just these points. Make sure to argue why you chose your interpolation scheme.

(b) Download the following dataset: `wget [to be added later]`. This dataset is the output of a halo finder, it contains the masses of the haloes identified in a simulation. Make a binned plot of the haloes as a function of mass, use 14 bins, calculate the bins and the error on the value in the bin, and make a log-log plot.

(c) Use non-linear fitting to fit a line with parameters A, a and α to the data. Use as your initial estimate the values for A, a and α given above. Determine the values of these parameters and do a measure of goodness of fit. Give the values and the goodness of fit, and overplot your best fit halo mass function to the histogram you made earlier.

7. **Finding particle clusters**
In general finding haloes in a simulation is difficult and very advanced algorithms are used to find dark matter haloes. In this exercise we will use a few basic algorithms to find clusters in datasets.

(a) Let's consider the following dataset which can be downloaded as `wget [data will be added later]`. [This question will be amended at a later time based on the topics discussed in lecture 12.]

8. **Voronoi tessellations**
A very useful application of Voronoi tessellations is to determine the positions of clusters. Download the halo positions of the EAGLE simulations using `wget ...`.

(a) Make a plot of the positions of the haloes and determine by eye where the highest density of particles is in the simulation. Highlight this position in the plot.

(b) Do a Voronoi tessellation on the positions of the haloes in 3D and determine the cell with the smallest volume and the largest volume. Determine where the densest cluster is and where the largest empty space is in the volume and show the result.

(c) Make a histogram of the volume of the cells, does the cell volume follow any particular functional form?

9. **Evolving the simulation**

In this final part we will look at a technique to solve the equations of gravity called the particle mesh (PM) algorithm. Compared to the particle-particle (PP) method this method does not calculate all the individual interactions between particles but uses a mesh to calculate the gravitational forces. The PM method is especially popular because of its fast scaling with the number of simulation particles and number of grid cells, because the time scales as time $\propto O(N_p) + O(N_g \log N_g)$ where N_p is the number of particles and N_g is the number of grid cells (compared with $O(N_p^2)$ for the PP method). Besides this the PM method also naturally incorporates periodic boundary conditions required for cosmological simulations. Because of this PM and descendants such as P^3M, TreePM and FMMPM are used widely in todays state-of-the-art cosmological simulations.[9] In general PM codes solve the Poisson equation given by:

$$\nabla^2 \Phi = 4\pi G \rho_{\rm tot} - \Lambda, \tag{25}$$

and the equations of motion of the particles are given by:

$$\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = \mathbf{u}, \quad \frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = -\nabla\Phi. \tag{26}$$

In this case all equations are given in proper coordinates including all derivatives. It however is convenient to use comoving variables and dimensionless units. Lets define the dimensionless units with a tilde as:

$$\tilde{\mathbf{x}} = a^{-1}\frac{\mathbf{r}}{r_0}, \quad \tilde{\mathbf{p}} = a\frac{\mathbf{v}}{v_0}, \quad \tilde{\phi} = \frac{\phi}{\phi_0}, \quad \tilde{\rho} = a^3\frac{\rho}{\rho_0}, \tag{27}$$

in which $\tilde{\mathbf{x}}$ is a comoving coordinate and $\mathbf{v} = \mathbf{u} - H\mathbf{r} = a\dot{\mathbf{x}}$ is the peculiar velocity and ϕ is the peculiar potential given by:

$$\phi = \Phi + \frac{1}{2}a\ddot{a}\left(\frac{r}{a}\right)^2 = \Phi + \frac{H_0^2}{2}\left(\Omega_{\Lambda,0} - \frac{1}{2}\frac{1}{a^3}\Omega_{m,0}\right)r^2. \tag{28}$$

Here $\Omega_m = \frac{8\pi G\rho_0}{3H_0^2} \equiv \frac{\rho_0}{\rho_{\rm crit}}$ and $\Omega_\Lambda = \frac{\Lambda}{3H_0^2}$. But basically there is only one unit which we are free to choose, and the rest follows; we will choose r_0 to be equal to the size of a PM grid cell:

$$r_0 = \frac{L_{\rm box}}{N_g}. \tag{29}$$

N_g^3 is the total number of grid cells. In this case the rest of the units are defined as:

$$t_0 \equiv \frac{1}{H_0}, \quad v_0 \equiv \frac{r_0}{t_0}, \quad \rho_0 \equiv \frac{3H_0^2}{8\pi G}\Omega_m = \Omega_m\rho_{\rm crit}, \quad \phi_0 = \frac{r_0^2}{t_0^2} = v_0^2. \tag{30}$$

Furthermore it is convenient to take the expansion factor as a time variable, in this case the equations of motion become:

$$\nabla^2\phi = 4\pi G\Omega_m\rho_{\rm crit}a^{-1}\delta, \qquad \delta = \frac{\rho - \bar{\rho}}{\bar{\rho}}, \tag{31}$$

$$\text{and} \quad \frac{\mathrm{d}\mathbf{p}}{\mathrm{d}a} = \frac{-\nabla\phi}{\dot{a}}, \qquad \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}a} = \frac{\mathbf{p}}{\dot{a}a^2}. \tag{32}$$

In which $\dot{a}$ can be written as:

$$\dot{a} = \frac{H_0}{\sqrt{a}}\sqrt{\Omega_m + \Omega_k a + \Omega_\Lambda a^3}. \tag{33}$$

[9] TreePM is basically a gravity tree + PM in which the tree uses the center of mass for particles far away, codes like Gadget and Arepo use treePM. FMMPM is a combination of PM and the fast multipole method (FMM) in which the multipoles of groups are particles are calculated when far away, similarly to a tree code, this is used in codes like Swift.

But we also want to make the above equations dimensionless, this means we obtain:

$$\nabla^2\tilde{\phi} = \frac{3}{2}\frac{\Omega_0}{a}\tilde{\delta}, \quad \frac{\mathrm{d}\tilde{\mathbf{p}}}{\mathrm{d}a} = -f(a)\tilde{\nabla}\tilde{\phi}, \quad \frac{\mathrm{d}\tilde{\mathbf{x}}}{\mathrm{d}a} = f(a)\frac{\tilde{\mathbf{p}}}{a^2}. \tag{34}$$

Where we have defined $\tilde{\delta} = \tilde{\rho} - 1$ and $f(a) = \frac{H_0}{\dot{a}} = \sqrt{a}/\sqrt{\Omega_m + \Omega_k a + \Omega_\Lambda a^3}$. This means that if we use the leapfrog algorithm the equations are given by:

$$\tilde{\mathbf{p}}_{n+1/2} = \tilde{\mathbf{p}}_{n-1/2} + f(a)\tilde{g}\Delta a, \tag{35}$$

$$\tilde{\mathbf{x}}_{n+1} = \tilde{\mathbf{x}} + \frac{1}{(a_{n+1/2})^2} f(a_{n+1/2})\tilde{p}_{n+1/2}\Delta a, \tag{36}$$

in which $\tilde{g} = -\tilde{\nabla}\tilde{\phi}$.

Write your own PM algorithm code using the information above. Use a time stepping size of $\Delta a = 0.0001$ and a start redshift of $z = 50$. Output at least 300 hdf5 snapshots to make a movie of the simulation. The code of this simulation should be in the top folder and if we are in your folder we should be able to run it using `./pm.py init.hdf5` in which `init.hdf5` are initial conditions for your simulation. This exercise is not included in the total run time of your script and running this code is allowed to take up to 30 minutes. You are of course allowed to use all code developed so far in this exercise. For the `init.hdf5` you are allowed to use 64^3 randomly placed particles with zero velocity. But you will obtain bonus points if you use the Zeldovich approximation – only do this if you trust your implementation of the Zeldovich approximation. In your simulation use a cosmological model with $\Omega_\Lambda = 0.7$ and $\Omega_m = 0.3$. As an output we expect a movie of at least 10 seconds with 30 frames per second. Make the movie such that the angle we look at the periodic box rotates from 0 degrees to 90 degrees in 10 seconds.