UiO : Institute of Theoretical Astrophysics
The Faculty of Mathematics and Natural Sciences

Report for AST9240:
The evolution of structures in the universe

Maksym Brilenkov

May 1, 2019

1 INTRODUCTION

During the first two milestones, I have already looked into evolution of background properties (Ref. [1]) and ionization history (Ref. [2]) of the Universe. Present work completes all preliminary steps, which are necessary to achieve the final goal of the project - computation of CMB power spectrum; thus, my attention is shifted towards scalar perturbations.

The main aim here, as explained by Erisken *et al* [3], is "to construct the two-dimensional grid in time and Fourier scale, x and k, for each of the main physical quantities of interest, $\Phi(x,k)$, $\Psi(x,k)$, $\delta(x,k)$, $\delta_b(x,k)$, $v(x,k)$, $v_b(x,k)$, $\Theta_l(x,k)$, $\Theta_l^P(x,k)$, $\mathcal{N}_l(x,k)$ and their derivatives". As always, I do so by following approach of Callin *et al* [4].

The report organized as follows. In section 2, I write the main equations to be solved. Section 3 explains all relevant parts of the code written to integrate given differential equations. Section 4 is dedicated for results. Code is listed in the end of this report.

2 METHODS

Callin *et al* [4] uses Newtonian gauge and writes perturbation metric as

$$g_{\mu\nu} = \begin{pmatrix} -(1+2\Psi) & 0 \\ 0 & a^2\delta_{ij}(1+\Phi) \end{pmatrix}, \tag{2.1}$$

which leaves only scalar perturbations for consideration. Perturbations to the photons are defined as the relative variation of the photon temperature (Ref. [4]) and can be expanded in multipoles

$$\Theta_l = \frac{i^l}{2}\int_{-1}^{1} \mathscr{P}_l(\mu)\Theta(\mu)d\mu, \quad \Theta(\mu) = \sum_{l=0}^{\infty} \frac{2l+1}{i^l}\Theta_l\mathscr{P}_l(\mu), \quad \mu = \frac{\vec{k}\cdot\vec{p}}{kp}, \tag{2.2}$$

where $\mathscr{P}_l(\mu)$ are the Legendre polynomials. In addition to the temperature perturbation, there's also perturbations to the photon polarization, which is denoted by $\Theta^P(\mu)$.

Turning into the Fourier space, it is possible (see e.g. [4] or [5]) to write the full set of Einstein-Boltzmann equations as

$$\Theta_0' = -\frac{ck}{\mathscr{H}}\Theta_1 - \Phi', \tag{2.3}$$

$$\Theta_1' = \frac{ck}{3\mathscr{H}}\Theta_0 - \frac{2ck}{3\mathscr{H}}\Theta_2 + \frac{ck}{3\mathscr{H}}\Psi + \tau'\left(\Theta_1 + \frac{1}{3}v_b\right), \tag{2.4}$$

$$\Theta_l = \frac{l}{2l+1}\frac{ck}{\mathscr{H}}\Theta_{l-1} - \frac{l+1}{2l+1}\frac{ck}{\mathscr{H}}\Theta_{l+1} + \tau'\left(\Theta_l - \frac{1}{10}\Pi\delta_{l,2}\right), \quad 2 \le l < l_{\max}, \tag{2.5}$$

$$\Theta_l' = \frac{ck}{\mathscr{H}}\Theta_{l-1} - \frac{c(l+1)}{\mathscr{H}\eta(x)}\Theta_l + \tau'\Theta_l, \quad l = l_{\max}, \tag{2.6}$$

including polarization

$$\Theta_{P0}' = -\frac{ck}{\mathscr{H}}\Theta_{P1} + \tau'\left(\Theta_{P0} - \frac{1}{2}\Pi\right), \tag{2.7}$$

$$\Theta_{Pl}' = \frac{l}{2l+1}\frac{ck}{\mathscr{H}}\Theta_{l-1}^P - \frac{l+1}{2l+1}\frac{ck}{\mathscr{H}}\Theta_{l+1}^P + \tau'\left(\Theta_l^P - \frac{1}{10}\Pi\delta_{l,2}\right), \quad 1 \le l < l_{\max}, \tag{2.8}$$

$$\Theta_{Pl}' = \frac{ck}{\mathscr{H}}\Theta_{l-1}^P - \frac{c(l+1)}{\mathscr{H}\eta(x)}\Theta_l^P + \tau'\Theta_l^P, \quad l = l_{\max}, \tag{2.9}$$

$$\tag{2.10}$$

and neutrinos

$$\mathcal{N}_0' = -\frac{ck}{\mathscr{H}}\mathcal{N}_1 - \Phi', \tag{2.11}$$

$$\mathcal{N}_1' = \frac{ck}{3\mathscr{H}}\mathcal{N}_0 - \frac{2ck}{3\mathscr{H}}\mathcal{N}_2 + \frac{ck}{3\mathscr{H}}\Psi, \tag{2.12}$$

$$\mathcal{N}_l' = \frac{l}{2l+1}\frac{ck}{\mathscr{H}}\mathcal{N}_{l-1} - \frac{l+1}{2l+1}\frac{ck}{\mathscr{H}}\mathcal{N}_{l+1}, \quad 2 \le l < l_{\max,\nu}, \tag{2.13}$$

$$\mathcal{N}_l' = \frac{ck}{\mathscr{H}}\mathcal{N}_{l-1} - \frac{c(l+1)}{\mathscr{H}\eta(x)}\mathcal{N}_l, \quad l = l_{\max,\nu}, \tag{2.14}$$

$$\tag{2.15}$$

where potentials are defined as

$$\Phi' = \Psi - \frac{c^2k^2}{3\mathscr{H}^2}\Phi + \frac{H_0^2}{2\mathscr{H}^2}\left(\Omega_m a^{-1}\delta + \Omega_b a^{-1}\delta_b + 4\Omega_r a^{-2}\Theta_0 + 4\Omega_\nu a^{-2}\mathcal{N}_0\right) \tag{2.16}$$

$$\Psi = -\Phi - \frac{12H_0^2}{c^2k^2a^2}\left(\Omega_r\Theta_2 + \Omega_\nu\mathcal{N}_2\right) \tag{2.17}$$

$$v' = -v - \frac{ck}{\mathscr{H}}\Psi, \tag{2.18}$$

$$v_b' = -v_b - \frac{ck}{\mathscr{H}}\Psi + \tau' R\left(3\Theta_1 + v_b\right), \tag{2.19}$$

$$\delta' = \frac{ck}{\mathscr{H}}v - 3\Phi', \tag{2.20}$$

$$\delta_b' = \frac{ck}{\mathscr{H}}v_b - \Phi' \tag{2.21}$$

together with the additional quantities

$$R = \frac{4\Omega_r}{3\Omega_b a}, \tag{2.22}$$

$$\Pi = \Theta_2 + \Theta_0^P + \Theta_2^P. \tag{2.23}$$

2.1 INITIAL CONDITIONS

In order to integrate equations above numerically, we need to establish the initial conditions. Their derivation (see [4] for more details) is based on the idea that optical depth, τ, together with its derivative, τ', is very large during early time of the Universe (small values of a). Thus, using a very small quantity, $\epsilon = k/\mathscr{H}\tau'$, as an expansion parameter (Ref. [4]), it is possible to write the full set of initial conditions in the form (Ref. [3])

$$\Psi = -1, \tag{2.24}$$

$$\Phi = -\Psi\left(1 + \frac{2f_\nu}{5}\right), \tag{2.25}$$

$$\delta = \delta_b = -\frac{3}{2}\Psi, \tag{2.26}$$

$$v = v_b = -\frac{ck}{2\mathscr{H}}\Psi, \tag{2.27}$$

$$\Theta_0 = -\frac{1}{2}\Psi \tag{2.28}$$

$$\Theta_1 = \frac{ck}{6\mathscr{H}}\Psi \tag{2.29}$$

$$\Theta_2 = \begin{cases} -\frac{8ck}{15\mathscr{H}\tau'}\Theta_1 & P \\ -\frac{20ck}{45\mathscr{H}\tau'}\Theta_1 & \not{P} \end{cases} \tag{2.30}$$

$$\Theta_l = -\frac{l}{2l+1}\frac{ck}{\mathscr{H}\tau'}\Theta_{l-1} \tag{2.31}$$

$$\tag{2.32}$$

including polarization

$$\Theta_0^P = \frac{5}{4}\Theta_2 \tag{2.33}$$

$$\Theta_1^P = -\frac{ck}{4\mathscr{H}\tau'}\Theta_2 \tag{2.34}$$

$$\Theta_2^P = \frac{1}{2}\Theta_2 \tag{2.35}$$

$$\Theta_l^P = -\frac{l}{2l+1}\frac{ck}{\mathscr{H}\tau'}\Theta_{l-1}^P \tag{2.36}$$

$$\tag{2.37}$$

and neutrinos

$$\mathscr{N}_0 = -\frac{1}{2}\Psi \tag{2.38}$$

$$\mathscr{N}_1 = \frac{ck}{6\mathscr{H}}\Psi \tag{2.39}$$

$$\mathscr{N}_2 = -\frac{c^2k^2a^2\Phi}{12H_0^2\Omega_\nu}\frac{1}{\frac{5}{2f_\nu}+1} \tag{2.40}$$

$$\mathscr{N}_l = \frac{ck}{(2l+1)\mathscr{H}}N_{l-1}, \quad l \geq 3 \tag{2.41}$$

This regime is called *tight coupling* and it is relevant as long as $|\tau'| > 10$, or $|k/(\mathscr{H}\tau')| < 1/10$, but no later than the start of recombination. The important thing about this regime, is that the only relevant quantities here are the monopole, Θ_0, the dipole, Θ_1, and the quadrupole, Θ_2, due to the electrons' ability to observe only nearby temperature fluctuations. This implies that I should consider only $l = 0, 1$, while higher order moments are given by the initial conditions (Ref. [3]).

In addition, as explained in [3] and [4], the equations above are very unstable during the early times, due to multiplication of τ' and $(3\Theta_1 - v_b)$ - the first one is very large, while the other is small. Thus, I need to use a proper approximation for $(\Theta_1 - v_b)$ which overwrites expressions for Θ_1' and v_b' as

$$\Theta_1' = \left(q - v_b'\right) \Big/ 3, \tag{2.42}$$

$$v_b' = \left[-v_b - \frac{ck}{\mathscr{H}}\Psi + R\left(q + \frac{ck}{\mathscr{H}}\left(-\Theta_0 + 2\Theta_2\right) - \frac{ck}{\mathscr{H}}\Psi\right)\right] \Big/ \left(1 + R\right), \tag{2.43}$$

$$q = \frac{-\left[\left(1-2R\right)\tau' + \left(1+R\right)\tau''\right]\left(3\Theta_1 + v_b\right) - \frac{ck}{\mathscr{H}}\Psi + \left(1 - \frac{\mathscr{H}'}{\mathscr{H}}\right)\frac{ck}{\mathscr{H}}\left(-\Theta_0 + 2\Theta_2\right) \frac{ck}{\mathscr{H}}\Theta_0'}{\left(1+R\right)\tau' + \frac{\mathscr{H}'}{\mathscr{H}} - 1} \tag{2.44}$$

3 ALGORITHMS

Section 2 completes the set of equations I am going to use during this milestone. To sum up, the idea is pretty straightforward - I need to integrate Eistein-Boltzmann equations twice: for tight coupling regime (starting with scale factor, a, of order 10^{-8} till the end of tight coupling) and later on (from tight coupling till the present day). As always, I am going to use Runge-Kutta method with varying step size, embedded into the modules "ode_solver.f90" and "bs_mod.f90" (please, refer to [6] for more details).

The main working module is "evolution_mod.f90", which stands for *evolution module*. Here,

- I first construct the grid of k-values:

```
ks(1) = k_min
do i = 2, n_k
   ks(i) = k_min + (k_max-k_min) * ((i - 1.d0) / (n_k - 1.d0)
      ↪ )**2
end do
```

- and calculate the values of initial conditions

```
! Getting the value for H_p from previously calculated
   ↪ routine (look into time_mod.f90)
x_init        = log(a_init)
H_p           = get_H_p(x_init)
! Getting value for dtau from previous milestone
dtau          = get_dtau(x_init)
! I define a new variable to ease writing code
allocate(ckHp(n_k))
ckHp(:)           = c * ks(:) / H_p

Psi(0,:)      = -1.d0
! for N = 3 neutrino species
f_nu          = Omega_nu / (Omega_nu + Omega_r)!0.405d0
! Grav. Potential
Phi(0, :)     = -Psi(0, :) * (1.d0 + 2.d0 * f_nu / 5.d0)
delta(0, :)   = -(3.d0 / 2.d0) * Psi(0, :)
delta_b(0, :) = delta(0, :)

! We are looping from k = 1 to k = 100
do i = 1, n_k
   v(0, i)         = -ckHp(i) * Psi(0, i) / 2.d0
   v_b(0, i)       = v(0, i)
   ! Theta_0
   Theta(0, 0, i) = -0.5d0 * Psi(0, i)
   ! Theta_1
   Theta(0, 1, i) =  ckHp(i) * Psi(0, i) / 6.d0
```

```
   Theta(0, 2, i) = -8.d0 * ckHp(i) / (15.d0 * dtau) * Theta
      ↪ (0, 1, i)
   do l = 3, lmax_int
      Theta(0, l, i) = -l / (2.d0 * l + 1.d0) * ckHp(i) *
         ↪ Theta(0, l-1, i) / dtau
   end do

   ! Polarisation
   ThetaP(0, 0, i) = 5.d0 * Theta(0, 2, i) / 4.d0
   ThetaP(0, 1, i) = -ckHp(i) / (4.d0 * dtau) * Theta(0, 2, i
      ↪ )
   ThetaP(0, 2, i) = 0.25d0 * Theta(0, 2, i)
   do l = 3, lmax_int
      ThetaP(0, l, i) = -l / (2.d0 * l + 1.d0) * ckHp(i) *
         ↪ ThetaP(0, l-1, i) / dtau
   end do

   ! Neutrinos
   Nu(0, 0, i) = -0.5d0 * Psi(0, i)
   Nu(0, 1, i) = ckHp(i) * Psi(0, i) / 6.d0
   Nu(0, 2, i) = -(c * ks(i) * a_init / H_0)**2 * Phi(0, i) /
      ↪  (12.d0 * Omega_nu) * (5.d0 / (2.d0 * f_nu) + 1.d0)
      ↪ **(-1)
   do l = 3, lmax_nu
      Nu(0, l, i) = ckHp(i) * Nu(0, l-1, i) / (2.d0 * l + 1.
         ↪ d0)
   end do

end do
```

- To start integration, I need to know the time when tight coupling ends. This is given by the function "get_tight_coupling_time(k)"

```
function get_tight_coupling_time(k)
implicit none

real(dp),                 intent(in) :: k
real(dp), allocatable, dimension(:) :: condition
real(dp)                            :: x_current, x_step,
   ↪ x_coupling
logical                             :: found_coupling_time
real(dp)                            ::
   ↪ get_tight_coupling_time

! Setting the time when Recombination starts
z_start_rec = 1630.4d0
x_start_rec = -log(1.d0 + z_start_rec)
! Setting-up the array for every condition listed above
allocate(condition(1:3))
```

```
! Setting the initial value for x
x_current = x_init
! Creating a very small step to count from x_init to
   ↪ x_start_rec
x_step = (x_start_rec - x_init) * 0.00001d0
! Variable to stop the loop when the time is found
found_coupling_time = .false.
! Looping through conditions till x_start_rec or if one of
   ↪ the other
! statements give us the earlier time
do while ((found_coupling_time == .false.) .and. (x_current
   ↪ <= x_start_rec))

   ! In Callin (2006), it is stated to use absolute values of
      ↪  the expressions above
   condition(1) = abs(get_dtau(x_current))
   condition(2) = abs(c * k / (get_dtau(x_current) * get_H_p(
      ↪ x_current)))
   condition(3) = abs(x_current)

   ! This one doesn't work unless we change the values for
      ↪ Omegas (i.e. cosmological parameters),
   ! so I am including it anyway
   if (condition(1) < 10.d0) then
      x_coupling = x_current
      found_coupling_time = .true.
   ! This one works for high values of k
   else if (condition(2) > 0.1d0) then
      x_coupling = x_current
      found_coupling_time = .true.
   else
      x_coupling = x_current
   end if

   ! Going to the next point on the grid
   x_current = x_current + x_step
end do

! Returning the time of Tight Coupling
get_tight_coupling_time = x_coupling

deallocate(condition)
end function get_tight_coupling_time
```

which takes as input the current grid value of k, checks for the three conditions (explained in Section 2) and returns the value of x (remember $x = \ln a$).

- In addition, I also wrote two subroutines - "equations_before_tight_coupling"

and "equations_after_tight_coupling" - in the form of "derivs" from "ode_solver" module. They take as an input the current values of variables in question (e.g. Θ) in a given moment of time (x-value to be precise) and return the derivatives in accordance to Einstein-Boltzmann equations for a relevant period.

- With the above code at hand, I proceed for the integration. The subroutine responsible for it is called "integrate_perturbation_eqns". Looping through k-grid values, I integrate Einstein-Boltzmann equations separately for two different regimes. This process implies the construction of the second grid (composed of x-values) for each regime. These grid values are used as one of the input parameters for subroutines explained above and as a step parameter in "odeint" subroutine, used for integration.

- The last step is to find and save the values of variables in question, which corresponds to the k-values I am interested in. Say, I want a value of $k = 10H_0/c$. As I have an array, ks, which consists of only 100 numerical values, it may be possible that the value I've chosen doesn't exist. So, I am trying to *calculate* this index numerically using

$$index = \sqrt{\frac{k_{chosen} - k_{min}}{k_{max} - k_{min}} \cdot n_k^2}, \tag{3.1}$$

and then convert this number to the closest integer value, which will be the index in ks array. After that, I am looping through k-values and, if the calculated index equals to current one, I am passing values to a subroutine "save_data", which saves it to the separate ".dat" files. The part of the code looks like this (subroutine code can be found in the end of this report)

```
! Choose the value of k you want to plot
allocate(k_chosen(1:6))
k_chosen = (/ 0.1d0, 1.d1, 5.d1, 16.d1, 36.d1, 1.d3 /)
k_chosen = k_chosen * H_0 / c
! Find its index in the array of pre-computed values
allocate(index_chosen(1:size(k_chosen)))
do i = 1, size(k_chosen)
  index_chosen(i) = nint(sqrt((k_chosen(i) - k_min) / (k_max-
     ↪ k_min)) * n_k)
  if (index_chosen(i) == 0) then
     index_chosen = 1
  end if
  ! Saving values to separate files (into output directory)
  if (k == index_chosen(i)) then
     call save_data(k)
  end if
end do
```

4 RESULTS

Below I present the main results of this milestone. The Einstein-Boltzmann equations were integrated for several values of k and the results are plotted on figures 4.1-4.6. In addition, I plot both monopole and dipole values, $l = 0, 1$, for Θ, Θ^P and $\mathcal{N}$.

It is clear that baryonic matter strongly couples with radiation before and during the recombination. The oscillations (as seen from figures 4.1 and 4.2 during the time of recombination) for high values of k are due to the coupling between the photon pressure and the gravitational potential in the early Universe.

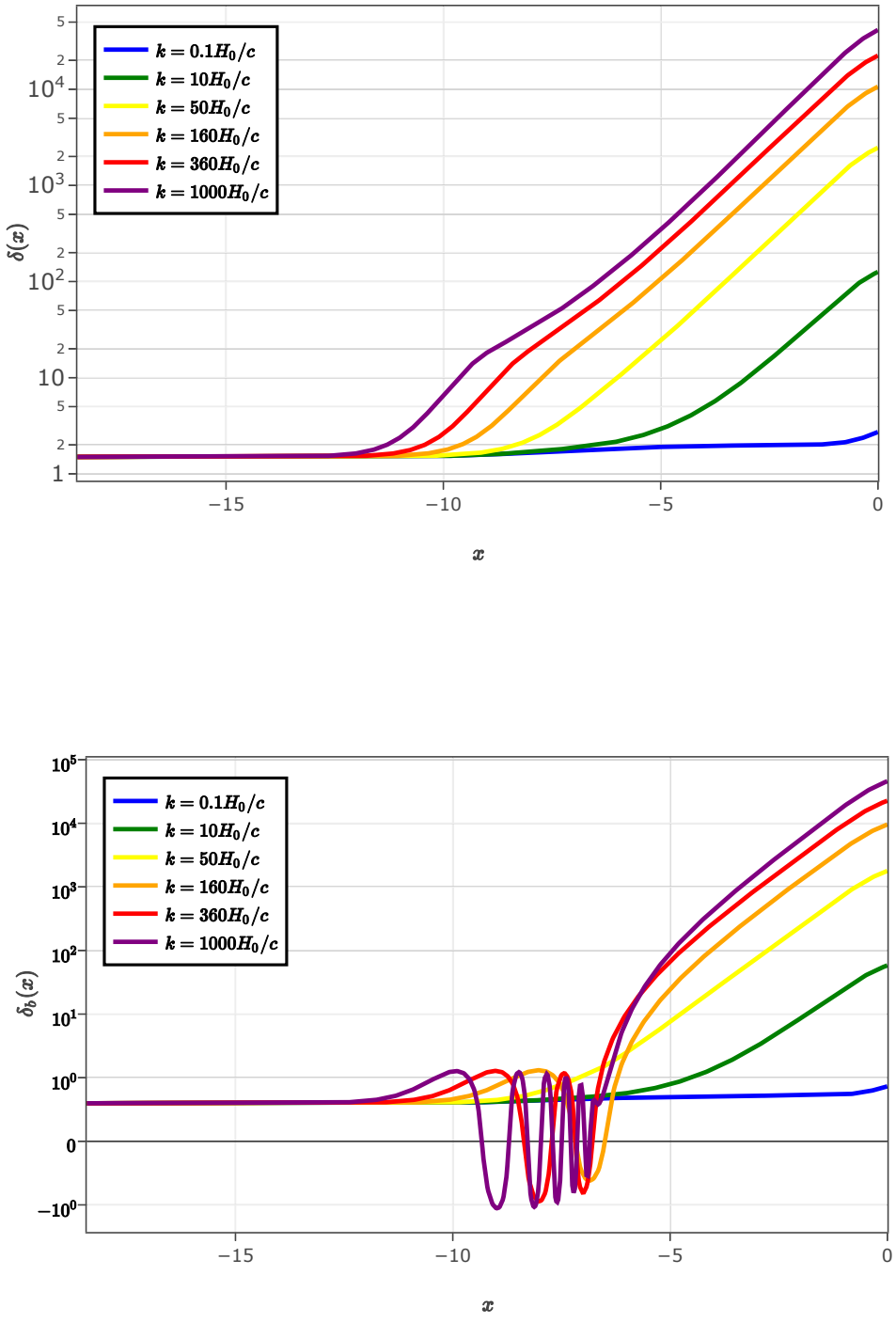

Figure 4.1: A plot of dark (top) and baryonic (bottom) matter overdensities for different values of k.

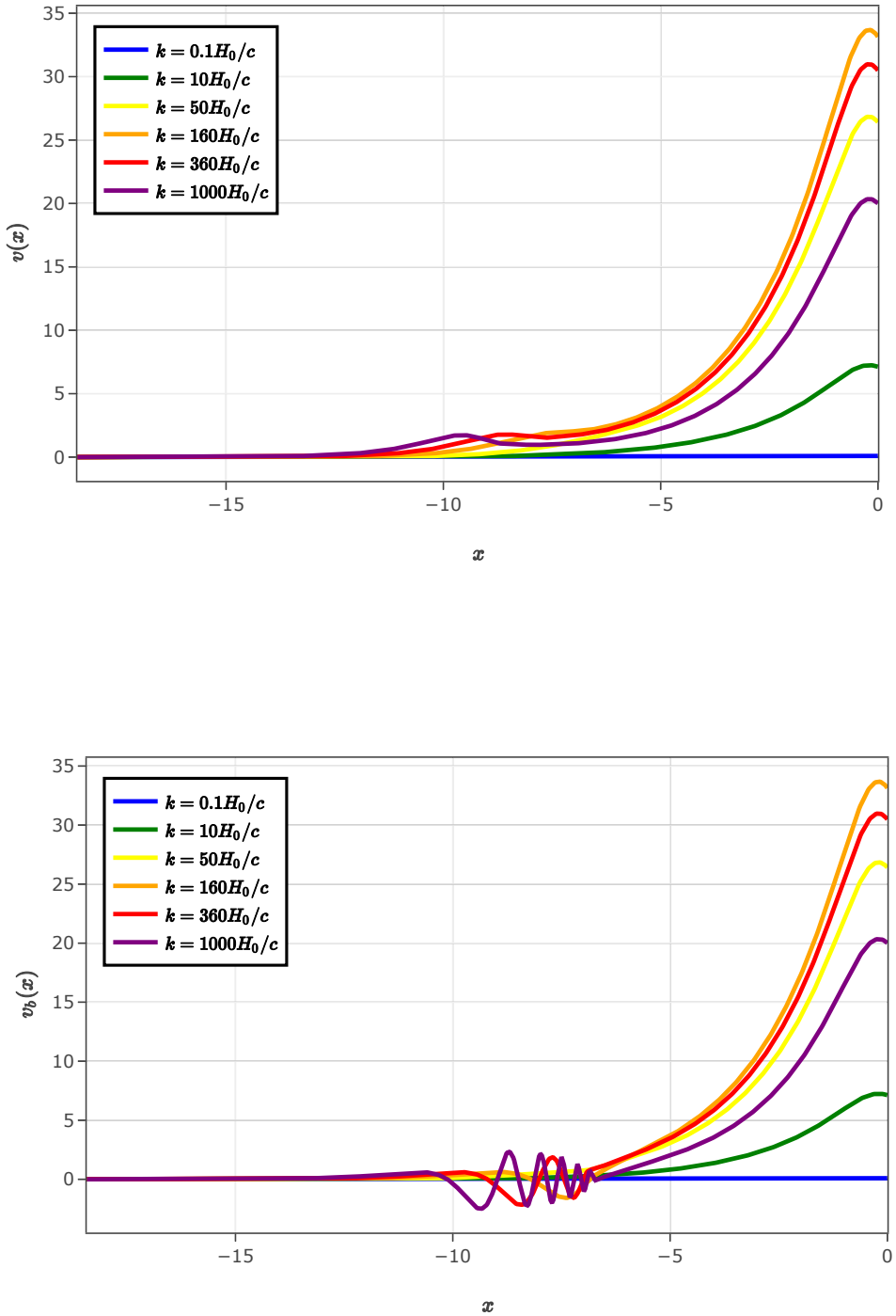

Figure 4.2: A plot of dark (top) and baryonic (bottom) matter velocities for different values of k.

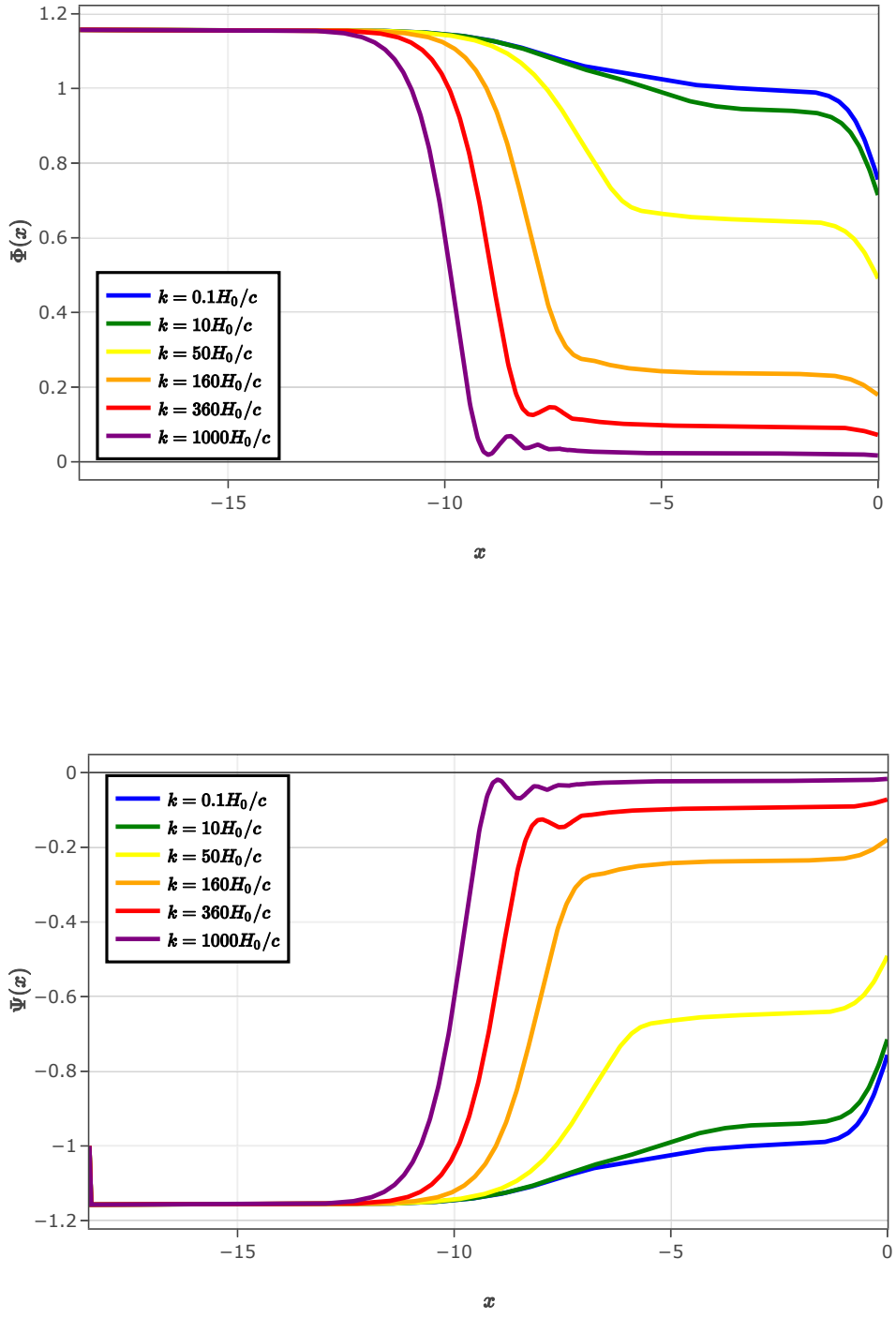

Figure 4.3: The plot of gravitational potentials. One is the inverse of the other (as it should be).

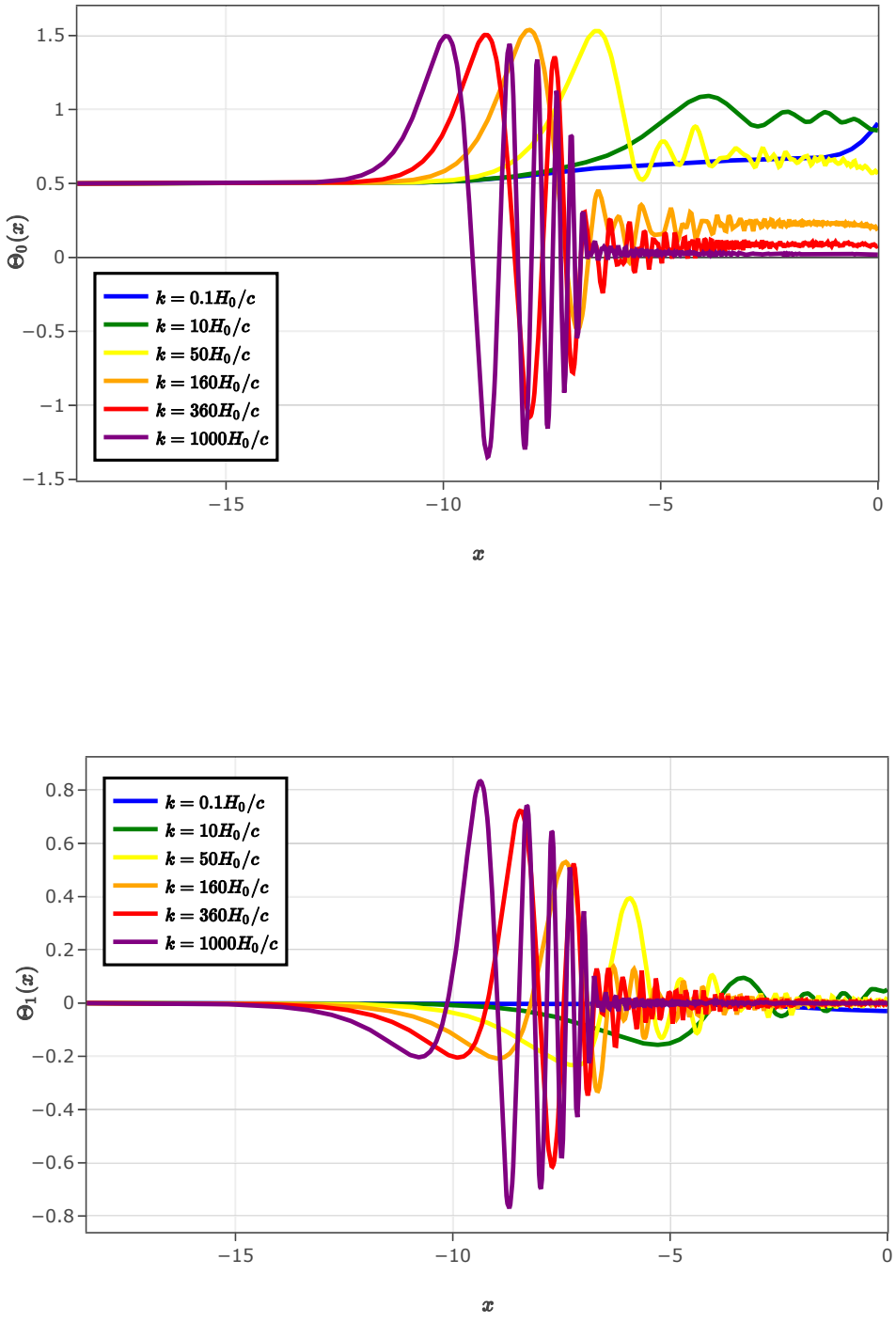

Figure 4.4: The plot of the monopole (top) and dipole (bottom) perturbation to photon distribution, Θ, for different values of k.

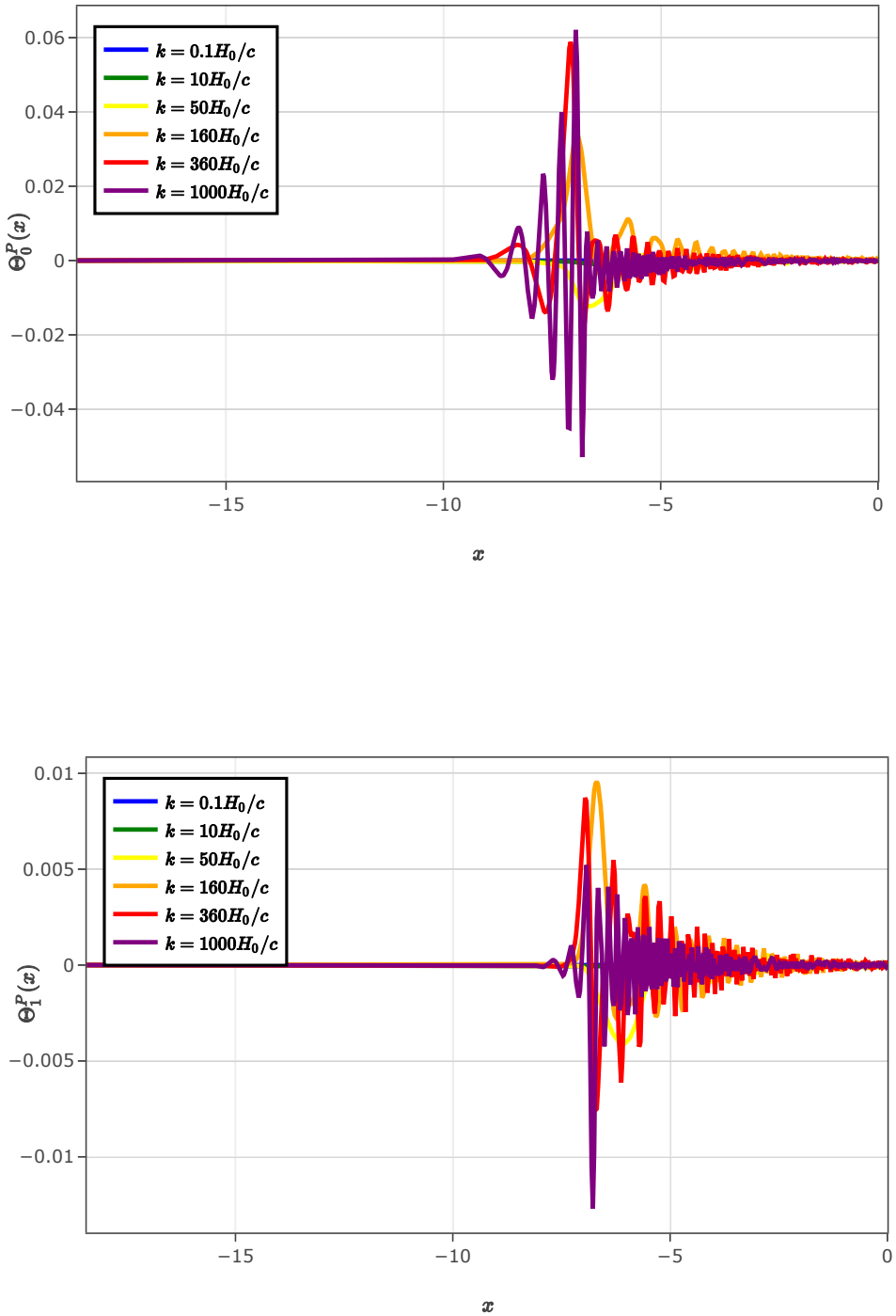

Figure 4.5: The plot of the monopole (top) and dipole (bottom) perturbation to photon polarization, Θ^P, for different values of k.

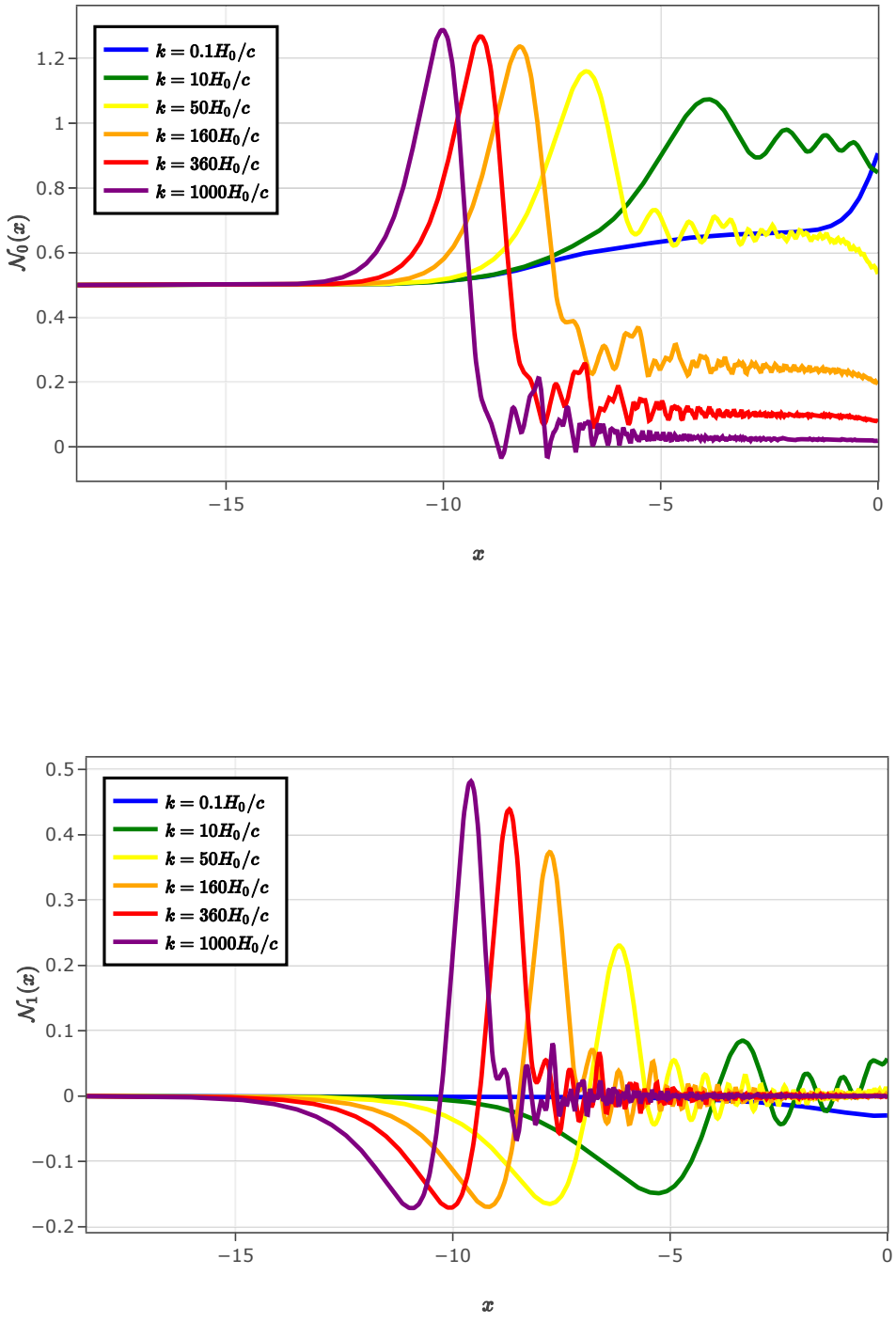

Figure 4.6: The plot of the monopole (top) and dipole (bottom) perturbation to neutrino distribution, $\mathcal{N}$, for different values of k.

REFERENCES

[1] M. Brilenkov, *Report for AST9240: The background evolution of the universe* (2019).

[2] M. Brilenkov, *Report for AST9240: The recombination history of the universe* (2019).

[3] H. K. Eriksen, *Milestone 3: The evolution of structures in the universe* (2019).

[4] P. Callin, *How to calculate the CMB spectrum*, arXiv:astro-ph/0606683.

[5] S. Dodelson, *Modern Cosmology 1st edition*, Academic Press (2003).

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, *Numerical Recipes in Fortran*, Cambridge University Press (1996)

CODE

Only the most relevant parts of the code present. Please note, that running the whole program takes quite some time (more than half an hour, ~ 1.5 minute for one full iteration due to the fact that main equations are highly unstable fin the beginning), but it is possible to run it for only several values of k (which is what I did for debugging purposes). If one wants to do so, one needs to simply change the main loop for k-values in "integrate_perturbation_eqns" subroutine. Furthermore, it is necessary to create the directory "data" in your working directory where code will put output ".dat" files.

Listing 1: evolution_mod.f90

```
module evolution_mod
  use healpix_types
  use params
  use time_mod
  use ode_solver
  use rec_mod
  implicit none

  ! Accuracy parameters
  real(dp),     parameter, private :: a_init   = 1.d-8
  real(dp),     parameter, private :: k_min    = 0.1d0 * H_0 / c
  real(dp),     parameter, private :: k_max    = 1.d3  * H_0 / c
  integer(i4b), parameter          :: n_k      = 100 !100
  integer(i4b), parameter, private :: lmax_int = 6

  ! Perturbation quantities
  !real(dp), allocatable, dimension(:,:,:) :: Theta
  !real(dp), allocatable, dimension(:,:)   :: delta
  !real(dp), allocatable, dimension(:,:)   :: delta_b
  !real(dp), allocatable, dimension(:,:)   :: Phi
  !real(dp), allocatable, dimension(:,:)   :: Psi
  !real(dp), allocatable, dimension(:,:)   :: v
  !real(dp), allocatable, dimension(:,:)   :: v_b
  !real(dp), allocatable, dimension(:,:)   :: dPhi
  !real(dp), allocatable, dimension(:,:)   :: dPsi
  !real(dp), allocatable, dimension(:,:)   :: dv_b
  !real(dp), allocatable, dimension(:,:,:) :: dTheta

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! Perutrbation quantities (Partly defined by me) !
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! Deltas
  real(dp), allocatable, dimension(:,:)   :: delta, delta_b, ddelta, ddelta_b
  ! Velocity
  real(dp), allocatable, dimension(:,:)   :: v, v_b, dv, dv_b
  ! Potential
  real(dp), allocatable, dimension(:,:)   :: Phi, Psi, dPsi, dPhi
  ! Theta
  real(dp), allocatable, dimension(:,:,:) :: Theta, dTheta
  ! Polarisation
  real(dp), allocatable, dimension(:,:,:) :: ThetaP, dThetaP
  ! Neutrinos
  real(dp), allocatable, dimension(:,:,:) :: Nu, dNu
  ! Differential equations
```

```
  real(dp), allocatable, dimension(:)      :: dif_eq
  ! Multipoles for neutrinos
  integer(i4b), parameter, private         :: lmax_nu  = 10
  ! The grid points accounting for values before and after tight coupling
  integer(i4b), parameter                  :: n_t_before = 500, n_t_after = 250
  integer(i4b), parameter                  :: n_tot = n_t_before + n_t_after
  ! x = log(a) and other parameters
  real(dp), parameter                      :: a_today = 1.d0
  real(dp),                 private        :: x_init, H_p, dtau, f_nu, x_current,
      ↪ x_step, x_today
  ! ckHp = c * ks / H_p
  real(dp), allocatable, dimension(:)      :: ckHp, x_evol!ckHp_new
  real(dp)                                 :: z_start_rec, x_start_rec
  ! For chosing plotting values
  !real(dp),                   dimension(:) :: k_chosen = (/ 0.1d0, 1.d1, 5.d1, 16.
      ↪ d1, 36.d1, 1.d3 /)
  real(dp),      allocatable, dimension(:) :: k_chosen
  integer(i4b), allocatable, dimension(:) :: index_chosen
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  ! Fourier mode list
  real(dp), allocatable, dimension(:) :: ks

  ! Book-keeping variables
  real(dp),     private :: k_current
  integer(i4b), private :: npar = 6+lmax_int

contains


  ! NB!!! New routine for 4th milestone only; disregard until then!!!
  subroutine get_hires_source_function(k, x, S)
    implicit none

    real(dp), pointer, dimension(:),   intent(out) :: k, x
    real(dp), pointer, dimension(:,:), intent(out) :: S

    integer(i4b) :: i, j
    real(dp)     :: g, dg, ddg, tau, dt, ddt, H_p, dH_p, ddHH_p, Pi, dPi, ddPi
    real(dp), allocatable, dimension(:,:) :: S_lores

    ! Task: Output a pre-computed 2D array (over k and x) for the
    !       source function, S(k,x). Remember to set up (and allocate) output
    !       k and x arrays too.
    !
    ! Substeps:
    !   1) First compute the source function over the existing k and x
    !      grids
    !   2) Then spline this function with a 2D spline
    !   3) Finally, resample the source function on a high-resolution uniform
    !      5000 x 5000 grid and return this, together with corresponding
    !      high-resolution k and x arrays

  end subroutine get_hires_source_function




  ! Routine for initializing and solving the Boltzmann and Einstein equations
  subroutine initialize_perturbation_eqns
    implicit none
```

```
    integer(i4b) :: l, i

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !        Set-upping the grid         !
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    ! Task: Initialize k-grid, ks; quadratic between k_min and k_max
    ! As written in Callin, we do so for n_k = 100 between k_min and k_max
    ! In the following way
    allocate(ks(n_k))
    !ks(1) = k_min
    ks(1) = k_min
    do i = 2, n_k
       ks(i) = k_min + (k_max-k_min) * ((i - 1.d0) / (n_k - 1.d0))**2
       !ks(i) = k_min + (k_max-k_min) * (i / n_k) **2.d0
       !if (i == 1) then
       !   ks(1) = k_min
       !end if
       !if (i == 100) then
       !   ks(100) = k_max
       !end if
    end do

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    ! Allocating the arrays for  perturbation quantities !
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    ! deltas
    allocate(delta(0:n_tot, n_k))
    allocate(delta_b(0:n_tot, n_k))
    allocate(ddelta(0:n_tot, n_k))
    allocate(ddelta_b(0:n_tot, n_k))
    ! velocity
    allocate(v(0:n_tot, n_k))
    allocate(v_b(0:n_tot, n_k))
    allocate(dv(0:n_tot, n_k))
    allocate(dv_b(0:n_tot, n_k))
    ! Potentials
    allocate(Phi(0:n_tot, n_k))
    allocate(Psi(0:n_tot, n_k))
    allocate(dPhi(0:n_tot, n_k))
    allocate(dPsi(0:n_tot, n_k))
    ! Theta
    allocate(Theta(0:n_tot, 0:lmax_int, n_k))
    allocate(dTheta(0:n_tot, 0:lmax_int, n_k))
    ! Polarisation
    allocate(ThetaP(0:n_tot, 0:lmax_int, n_k))
    allocate(dThetaP(0:n_tot, 0:lmax_int, n_k))
    ! Neutrinos
    allocate(Nu(0:n_tot, 0:lmax_nu, n_k))
    allocate(dNu(0:n_tot, 0:lmax_nu, n_k))

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !       Starting calculation         !
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    ! Getting the value for H_p from previously calculated routine (look into
        ↪ time_mod.f90)
    x_init        = log(a_init)
    H_p           = get_H_p(x_init)
    ! Getting value for dtau from previous milestone
    dtau          = get_dtau(x_init)
```

```
    ! I define a new variable to ease writing code
    allocate(ckHp(n_k))
    ckHp(:)           = c * ks(:) / H_p

    ! Task: Set up initial conditions for the Boltzmann and Einstein equations
    Psi(0,:)       = -1.d0
    ! for N = 3 neutrino species
    f_nu           = Omega_nu / (Omega_nu + Omega_r)!0.405d0
    ! Grav. Potential
    Phi(0, :)      = -Psi(0, :) * (1.d0 + 2.d0 * f_nu / 5.d0)
    delta(0, :)    = -(3.d0 / 2.d0) * Psi(0, :)
    delta_b(0, :) = delta(0, :)

    ! We are looping from k = 1 to k = 100
    do i = 1, n_k
       v(0, i)         = -ckHp(i) * Psi(0, i) / 2.d0
       v_b(0, i)       = v(0, i)
       ! Theta_0
       Theta(0, 0, i) = -0.5d0 * Psi(0, i)
       ! Theta_1
       Theta(0, 1, i) =  ckHp(i) * Psi(0, i) / 6.d0
       Theta(0, 2, i) = -8.d0 * ckHp(i) / (15.d0 * dtau) * Theta(0, 1, i)
       do l = 3, lmax_int
          Theta(0, l, i) = -l / (2.d0 * l + 1.d0) * ckHp(i) * Theta(0, l-1, i) /
              ↪ dtau
       end do

       ! Polarisation
       ThetaP(0, 0, i) = 5.d0 * Theta(0, 2, i) / 4.d0
       ThetaP(0, 1, i) = -ckHp(i) / (4.d0 * dtau) * Theta(0, 2, i)
       ThetaP(0, 2, i) = 0.25d0 * Theta(0, 2, i)
       do l = 3, lmax_int
          ThetaP(0, l, i) = -l / (2.d0 * l + 1.d0) * ckHp(i) * ThetaP(0, l-1, i) /
              ↪  dtau
       end do

       ! Neutrinos
       Nu(0, 0, i) = -0.5d0 * Psi(0, i)
       Nu(0, 1, i) = ckHp(i) * Psi(0, i) / 6.d0
       Nu(0, 2, i) = -(c * ks(i) * a_init / H_0)**2 * Phi(0, i) / (12.d0 *
           ↪ Omega_nu) * (5.d0 / (2.d0 * f_nu) + 1.d0)**(-1)
       do l = 3, lmax_nu
          Nu(0, l, i) = ckHp(i) * Nu(0, l-1, i) / (2.d0 * l + 1.d0)
       end do

    end do

  end subroutine initialize_perturbation_eqns

  subroutine integrate_perturbation_eqns
    implicit none

    integer(i4b) :: i, j, k, l, m
    integer(i4b) :: j1, j2, j3
    real(dp)     :: x1, x2, x_init
    real(dp)     :: eps, hmin, h1, x_tc, H_p, dt, t1, t2, ckHp_new

    real(dp), allocatable, dimension(:) :: y, y_tight_coupling, dydx

    ! Variables for numerical integration (using odeint)
    x_init  = log(a_init)
    x_today = log(a_today)
```

```
eps     = 1.d-8
hmin    = 0.d0
h1      = 1.d-5

!allocate(y(npar))
!allocate(dydx(npar))
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! The set of Equations for tight coupling !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! 1, 2 are delta and delta_b
! 3, 4 are v and v_b
! 5 is Phi
! 6, 7 are Theta_0 and Theta_1
! 8, 9 accounts for polarization (ThetaP)
! 10 and later stands for Neutrinos
!allocate(y_tight_coupling(1:(10+lmax_nu)))
! The same indexing holds for their derivatives
!allocate(dif_eq(1:(10+lmax_nu)))

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! The set of Equations for the rest of time-grid !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! First, we calculate the total amount of equations
! we want to make index for.
! 1-5 - see above;
! 6-(6+lmax_int) - for Theta;
! (7+lmax_int)-(7+lmax_int*2) - for ThetaP;
! (8+lmax_int*2)-(8+lmax_int*2 + lmax_nu - for Nu)
j1 = 7 + lmax_int
j2 = j1 + lmax_int + 1
j3 = j2 + lmax_nu
!allocate(y(1:j3))!y(npar))
!allocate(dydx(1:j3))
! The grid for numerical integration
!allocate(x_evol(0:n_tot))

! Propagate each k-mode independently
do k = 1, n_k

   print *, "Start calculation for k =", k
   allocate(y_tight_coupling(1:(10+lmax_nu)))
   ! The same indexing holds for their derivatives
   allocate(dif_eq(1:(10+lmax_nu)))
   allocate(y(1:j3))!y(npar))
   allocate(dydx(1:j3))
   ! The grid for numerical integration
   allocate(x_evol(0:n_tot))

   k_current = ks(k)  ! Store k_current as a global module variable

   ! Initialize equation set for tight coupling
   y_tight_coupling(1)   = delta(0, k)
   y_tight_coupling(2)   = delta_b(0, k)
   y_tight_coupling(3)   = v(0, k)
   y_tight_coupling(4)   = v_b(0, k)
   y_tight_coupling(5)   = Phi(0, k)
   y_tight_coupling(6)   = Theta(0, 0, k)
   y_tight_coupling(7)   = Theta(0, 1, k)
   ! Including Polarization
   y_tight_coupling(8)   = ThetaP(0, 0, k)
   y_tight_coupling(9)   = ThetaP(0, 1, k)
   ! and Neutrinos
```

```
      y_tight_coupling(10:) = Nu(0, :, k)

      ! Find the time to which tight coupling is assumed,
      ! and integrate equations to that time
      x_tc = get_tight_coupling_time(k_current)

      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      ! Integration BEFORE Tight Coupling !
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      ! Task: Integrate from x_init until the end of tight coupling, using
      !       the tight coupling equations

      ! To integrate, first we need to write the equations.
      ! I am using the subroutine to store all the equations
      ! in one array for easier access. First, I am passing
      ! inside the initial conditions and when getting the
      ! equations I want to solve.
      call equations_before_tight_coupling(x_init, y_tight_coupling, dif_eq)
      ! deltas
      ddelta(0, k)      = dif_eq(1)
      ddelta_b(0, k)    = dif_eq(2)
      ! velocity
      dv(0, k)          = dif_eq(3)
      dv_b(0, k)        = dif_eq(4)
      dPhi(0, k)        = dif_eq(5)
      dTheta(0, 0, k)   = dif_eq(6)
      dTheta(0, 1, k)   = dif_eq(7)
      dTheta(0, 2:, k)  = 0.d0 ! assume it is small
      ! Polarization
      dThetaP(0, 0, k)  = dif_eq(8)
      dThetaP(0, 1, 0)  = dif_eq(9)
      dThetaP(0, 2:, k) = 0.d0 ! assume it is small
      ! Neutrinos
      dNu(0, :, k) = dif_eq(10:(10+lmax_nu))
      ! Creating the grid on which we will integrate
      x_step = (x_tc - x_init) / n_t_before

      ! Making loop to go through all grid values
      x_evol(0) = x_init
      Psi(0, k) = -Phi(0, k) - 12.d0 * (H_0 / (c * k_current * exp(x_evol(0))))
          ↪ **2 * (Omega_r * Theta(0, 2, k) + Omega_nu * Nu(0, 2, k))

      do i = 1, n_t_before
         x_evol(i) = x_evol(0) + i * x_step
         ckHp_new = c * k_current / get_H_p(x_evol(i))
         ! Using odeint to integrate all equations
         call odeint(y_tight_coupling, x_evol(i - 1), x_evol(i), eps, h1, hmin,
             ↪ equations_before_tight_coupling, bsstep, output2)
         ! Passing the newly calculated values into the set of equations
         ! to calculate it once again on the next step.
         call equations_before_tight_coupling(x_evol(i), y_tight_coupling, dif_eq
             ↪ )
         !call equations_before_tight_coupling(x2, y_tight_coupling, dif_eq)
         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         ! Memorising the calculated values for a given step:
         ! Actual values
         delta(i, k)       = y_tight_coupling(1)
         delta_b(i, k)     = y_tight_coupling(2)
         v(i, k)           = y_tight_coupling(3)
         v_b(i, k)         = y_tight_coupling(4)
         ! Potentials
         Phi(i, k)         = y_tight_coupling(5)
```

```
      Psi(i, k)          = -Phi(i, k) - 12.d0 * (H_0 / (c * k_current * exp(
          ↪ x_evol(i))))**2 * (Omega_r * Theta(i, 2, k) + Omega_nu * Nu(i, 2,
          ↪  k))
      Theta(i, 0, k)     = y_tight_coupling(6)
      Theta(i, 1, k)     = y_tight_coupling(7)
      Theta(i, 2, k)     = -8.d0 * ckHp_new * Theta(i, 1, k) / (15.d0 *
          ↪ get_dtau(x_evol(i)))
      do l = 3, lmax_int
         Theta(i, l, k) = -l / (2.d0 * l + 1.d0) * ckHp_new * Theta(i, l-1, k)
             ↪  / get_dtau(x_evol(i))
      end do
      ! Including Polarization
      ThetaP(i, 0, k)    = y_tight_coupling(8)
      ThetaP(i, 1, k)    = y_tight_coupling(9)
      ThetaP(i, 2, k)    = Theta(i, 2, k) / 4.d0
      !Print *, ThetaP(i, 2, k)
      do l = 3, lmax_int
         ThetaP(i, l, k) = -l / (2.d0 * l + 1.d0) * ckHp_new * ThetaP(i, l-1,
             ↪ k) / get_dtau(x_evol(i))
      end do
      ! and Neutrinos
      Nu(i, :, k)        = y_tight_coupling(10:10+lmax_nu)

      ! Differential equations
      ! deltas
      ddelta(i, k)       = dif_eq(1)
      ddelta_b(i, k)     = dif_eq(2)
      ! velocity
      dv(i, k)           = dif_eq(3)
      dv_b(i, k)         = dif_eq(4)
      dPhi(i, k)         = dif_eq(5)
      dTheta(i, 0, k)    = dif_eq(6)
      dTheta(i, 1, k)    = dif_eq(7)
      dTheta(i, 2:, k)   = 0.d0
      ! Polarization
      dThetaP(i, 0, k)   = dif_eq(8)
      dThetaP(i, 1, k)   = dif_eq(9)
      dThetaP(i, 2:, k)  = 0.d0
      ! Neutrinos
      dNu(i, :, k)       = dif_eq(10:)
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   end do

   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   ! Integration AFTER Tight Coupling !
   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   ! Task: Set up variables for integration from the end of tight coupling
   ! until today

   ! Writing down the "initial" expressions for each variable
   y(1)               = delta(n_t_before, k)
   y(2)               = delta_b(n_t_before, k)
   y(3)               = v(n_t_before, k)
   y(4)               = v_b(n_t_before, k)
   y(5)               = Phi(n_t_before, k)
   y(6:6+lmax_int)    = Theta(n_t_before, 0:lmax_int, k)
   ! Including Polarization
   y(j1:j1+lmax_int) = ThetaP(n_t_before, 0:lmax_int, k)
   ! and Neutrinos
   y(j2:j3)           = Nu(n_t_before, 0:lmax_nu, k)
   ! deltas
   ddelta(n_t_before, k)      = dydx(1)
```

```
      ddelta_b(n_t_before, k)     = dydx(2)
      ! velocity
      dv(n_t_before, k)           = dydx(3)
      dv_b(n_t_before, k)         = dydx(4)
      dPhi(n_t_before, k)         = dydx(5)
      do l = 0, lmax_int
         dTheta(n_t_before, l, k)  = dydx(6+l)
         dThetaP(n_t_before, l, k) = dydx(j1+l)
      end do
      do l = 0, lmax_nu
         dNu(n_t_before, l, k)     = dydx(j2+l)
      end do

      ! Making loop to go through the rest of grid values
      j = 1
      x_step = (x_today - x_tc) / n_t_after
      do i = (n_t_before + 1), n_tot
         x_evol(i) = x_tc + j * x_step
         j = j + 1
         call odeint(y, x_evol(i-1), x_evol(i), eps, h1, hmin,
             ↪ equations_after_tight_coupling, bsstep, output2)
         ! Passing the newly calculated values into the set of equations
         ! to calculate it once again on the next step.
         call equations_after_tight_coupling(x_evol(i), y, dydx)
         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         ! Memorising the calculated values for a given step:
         ! Actual values
         delta(i, k)          = y(1)
         delta_b(i, k)        = y(2)
         v(i, k)              = y(3)
         v_b(i, k)            = y(4)
         ! Potentials
         Phi(i, k)            = y(5)
         Psi(i, k)            = -Phi(i, k) - 12.d0 * (H_0 / (c * k_current * exp(
             ↪ x_evol(i))))**2 * (Omega_r * Theta(i, 2, k) + Omega_nu * Nu(i, 2,
             ↪  k))
         ! Thetas
         do l = 0, lmax_int
            Theta(i, l, k)    = y(6+l)
            ThetaP(i, l, k)   = y(j1+l)
         end do
         ! Nus
         do l = 0, lmax_nu
            Nu(i, l, k)       = y(j2+l)
         end do

         ! Call the subroutine for saving data into a file
         ! Task: Store derivatives that are required for C_l estimation
         ! deltas
         ddelta(i, k)         = dydx(1)
         ddelta_b(i, k)       = dydx(2)
         ! velocity
         dv(i, k)             = dydx(3)
         dv_b(i, k)           = dydx(4)
         dPhi(i, k)           = dydx(5)
         do l = 0, lmax_int
            dTheta(i, l, k)   = dydx(6+l)
            dThetaP(i, l, k)  = dydx(j1+l)
         end do
         do l = 0, lmax_nu
            dNu(i, l, k)      = dydx(j2+l)
         end do
```

```
         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

      end do

      ! Choose the value of k you want to plot
      allocate(k_chosen(1:6))
      k_chosen = (/ 0.1d0, 1.d1, 5.d1, 16.d1, 36.d1, 1.d3 /)
      k_chosen = k_chosen * H_0 / c
      ! Find its index in the array of pre-computed values
      allocate(index_chosen(1:size(k_chosen)))
      do i = 1, size(k_chosen)
         index_chosen(i) = nint(sqrt((k_chosen(i) - k_min) / (k_max-k_min)) * n_k
             ↪ )
         if (index_chosen(i) == 0) then
            index_chosen = 1
         end if
         ! Saving values to separate files (into output directory)
         if (k == index_chosen(i)) then
            call save_data(k)
         end if
      end do

      deallocate(index_chosen)
      deallocate(k_chosen)

      ! Deallocating quantities to free the memory
      deallocate(y_tight_coupling)
      ! The same indexing holds for their derivatives
      deallocate(dif_eq)

      deallocate(y)
      deallocate(dydx)
      ! The grid for numerical integration
      deallocate(x_evol)
       ! Task: Integrate equations from tight coupling to today

         ! Task: Store variables at time step i in global variables
       !  delta(i, k)   = 0.d0
         !delta_b(i, k) = 0.d0
         !v(i, k)       = 0.d0
         !v_b(i, k)     = 0.d0
         !Phi(i, k)     = 0.d0
         !do l = 0, lmax_int
         !   Theta(i, l, k) = 0.d0
         !end do
         !Psi(i, k)     = 0.d0

         ! Task: Store derivatives that are required for C_l estimation
         !dPhi(i, k)      = 0.d0
         !dv_b(i, k)      = 0.d0
         !dTheta(i, :, k) = 0.d0
         !dPsi(i, k)      = 0.d0
      !end do

   end do

   ! deltas
   deallocate(delta)
   deallocate(delta_b)
   deallocate(ddelta)
   deallocate(ddelta_b)
   ! velocity
```

```
    deallocate(v)
    deallocate(v_b)
    deallocate(dv)
    deallocate(dv_b)
    ! Potentials
    deallocate(Phi)
    deallocate(Psi)
    deallocate(dPhi)
    deallocate(dPsi)
    ! Theta
    deallocate(Theta)
    deallocate(dTheta)
    ! Polarisation
    deallocate(ThetaP)
    deallocate(dThetaP)
    ! Neutrinos
    deallocate(Nu)
    deallocate(dNu)

  end subroutine integrate_perturbation_eqns


  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! Tight Coupling Time Computation !
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  ! Task: Complete the following routine, such that it returns the time at which
  !       tight coupling ends. In this project, we define this as either when
  !       dtau < 10 or c*k/(H_p*dt) > 0.1 or x > x(start of recombination)
  function get_tight_coupling_time(k)
    implicit none

    real(dp),                intent(in) :: k
    real(dp), allocatable, dimension(:) :: condition
    real(dp)                            :: x_current, x_step, x_coupling
    logical                             :: found_coupling_time
    real(dp)                            :: get_tight_coupling_time

    ! Setting the time when Recombination starts
    z_start_rec = 1630.4d0
    x_start_rec = -log(1.d0 + z_start_rec)
    ! Setting-up the array for every condition listed above
    allocate(condition(1:3))
    ! Setting the initial value for x
    x_current = x_init
    ! Creating a very small step to count from x_init to x_start_rec
    x_step = (x_start_rec - x_init) * 0.00001d0
    ! Variable to stop the loop when the time is found
    found_coupling_time = .false.
    ! Looping through conditions till x_start_rec or if one of the other
    ! statements give us the earlier time
    do while ((found_coupling_time == .false.) .and. (x_current <= x_start_rec))

       ! In Callin (2006), it is stated to use absolute values of the expressions
          ↪ above
       condition(1) = abs(get_dtau(x_current))
       condition(2) = abs(c * k / (get_dtau(x_current) * get_H_p(x_current)))
       condition(3) = abs(x_current)

       ! This one doesn't work unless we change the values for Omegas (i.e.
          ↪ cosmological parameters),
       ! so I am including it anyway
```

```
      if (condition(1) < 10.d0) then
         x_coupling = x_current
         found_coupling_time = .true.
      ! This one works for high values of k
      else if (condition(2) > 0.1d0) then
         x_coupling = x_current
         found_coupling_time = .true.
      else
         x_coupling = x_current
      end if

      ! Going to the next point on the grid
      x_current = x_current + x_step
   end do

   ! Returning the time of Tight Coupling
   get_tight_coupling_time = x_coupling

   deallocate(condition)
end function get_tight_coupling_time

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! For Tight Coupling Integration !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! This one is written in the form of derivs (from numerical recepies)
! to be able to integrate with odeint
subroutine equations_before_tight_coupling(x, y, dydx)
   implicit none

   ! Grid values
   real(dp),                intent(in)  :: x
   ! Function values
   real(dp), dimension(:), intent(in)  :: y
   ! Derivative values
   real(dp), dimension(:), intent(out) :: dydx
   ! All other parameters
   real(dp)                             :: k
   real(dp)                             :: a, ckHp, eta, H_p, dH_p, dtau, ddtau,
       ↪ dPhi_bracket, dv_b_bracket
   real(dp)                             :: R, q, PI, delta, delta_b, v, v_b, Phi,
       ↪ Psi
   real(dp), allocatable, dimension(:) :: Theta_, ThetaP_, Nu_
   integer(i4b)                         :: l, l_trick


   allocate(Theta_(0:2))
   allocate(ThetaP_(0:2))
   allocate(Nu_(0:lmax_nu))

   !!!!!!!!!!!!!!!!!!!!!!!!
   ! Necessary parameters !
   !!!!!!!!!!!!!!!!!!!!!!!!
   k               = k_current
   a               = exp(x)
   R               = 4.d0 * Omega_r / (3.d0 * Omega_b * a)
   ! eta(x)
   eta             = get_eta(x)
   ! tau' and tau''
   dtau            = get_dtau(x)
   ddtau           = get_ddtau(x)
   ! H_p and H_p'
```

```
    H_p             = get_H_p(x)
    dH_p            = get_dH_p(x)
    ! To ease calculation
    ckHp            = c * k / get_H_p(x)

    ! Current values of functions (values on each step)
    delta           = y(1)
    delta_b         = y(2)
    v               = y(3)
    v_b             = y(4)
    Phi             = y(5)
    Theta_(0)       = y(6)
    Theta_(1)       = y(7)
    ! Theta_2 as stated in milestone 3 pdf comes from initial condition
    Theta_(2)       = -8.d0 * ckHp * Theta_(1) / (15.d0 * dtau)
    ! Polarization
    ThetaP_(0)      = y(8)
    ThetaP_(1)      = y(9)
    ! From initial conditions as explained in milestone 3 pdf
    ThetaP_(2)      = Theta_(2) / 4.d0
    ! Neutrinos (equations valid for any l, not only l = 0, 1)
    Nu_(0:lmax_nu)  = y(10:10+lmax_nu)

    PI              = Theta_(2) + ThetaP_(0) + ThetaP_(2)
    Psi             = -Phi - 12.d0 * (H_0 / (c * k * a))**2 * (Omega_r * Theta_(2)
        ↪  + Omega_nu * Nu_(2))

    !!!!!!!!!!!!!!!
    ! Derivatives !
    !!!!!!!!!!!!!!!
    ! Phi'
    dPhi_bracket    = Omega_m * a**(-1.d0) * delta + Omega_b * a**(-1.d0) * delta_b
        ↪  + 4.d0 * Omega_r * a**(-2.d0) * Theta_(0) + 4.d0 * Omega_nu * a**(-2.
        ↪ d0) * Nu_(0)
!    dPhi_bracket    = Omega_m * exp(-1.d0 * x) * delta + Omega_b * exp(-1.d0 * x)
    ↪ * delta_b + 4.d0 * Omega_r * exp(-2.d0 * x) * Theta_(0) + 4.d0 * Omega_nu *
    ↪  exp(-2.d0 * x) * Nu_(0)
    dydx(5)         = Psi - (ckHp**2 / 3.d0) * Phi + (H_0 / H_p)**2 * dPhi_bracket
        ↪ / 2.d0
    ! delta'
    dydx(1)         = ckHp * v - 3.d0 * dydx(5)
    ! delta_b'
    dydx(2)         = ckHp * v_b - 3.d0 * dydx(5)
    ! velocity, v' and v_b'
    dydx(3)         = -v - ckHp * Psi
    dv_b_bracket    = -v_b - ckHp * Psi + R * (q + ckHp * (-Theta_(0) + 2.d0 *
        ↪ Theta_(2)) - ckHp * Psi)
    dydx(4)         = dv_b_bracket / (1.d0 + R)
    ! Theta_0'
    dydx(6)         = -ckHp * Theta_(1) - dydx(5)
    ! q parameter
    q               = (-((1.d0 - 2.d0 * R) * dtau + (1.d0 + R) * ddtau) * (3.d0 *
        ↪ Theta_(1) + v_b) - ckHp * Psi + (1.d0 - dH_p/H_p) * ckHp * (-Theta_(0)
        ↪ + 2.d0 * Theta_(2)) - ckHp * dydx(6)) / ((1.d0 + R) * dtau + dH_p/H_p -
        ↪  1.d0)
    ! Theta_1'
    dydx(7)         = (q - dydx(4)) / 3.d0
    ! ThetaP_0'
    dydx(8)         = -ckHp * ThetaP_(1) + dtau * (ThetaP_(0) - PI / 2.d0)
    ! ThetaP_1'
    dydx(9)         = ckHp * ThetaP_(0) / 3.d0 - (2.d0 / 3.d0) * ckHp * ThetaP_(2)
        ↪ + dtau * ThetaP_(1)
```

```
    ! Nu_0'
    dydx(10)        = -ckHp * Nu_(1) - dydx(5)
    ! Nu_1'
    dydx(11)        = ckHp * Nu_(0) / 3.d0 - (2.d0 / 3.d0) * ckHp * Nu_(2) + ckHp *
        ↪  Psi / 3.d0
    ! All other Nu
    do l = (10 + 2), (10 + lmax_nu)
       l_trick = l - 10
       if (l /= (10 + lmax_nu)) then
          dydx(l)  = ckHp * Nu_(l_trick-1) * l_trick / (2.d0 * l_trick + 1.d0) -
              ↪ ckHp * Nu_(l_trick+1) * (l_trick + 1.d0) / (2.d0 * l_trick + 1.d0
              ↪ )
       ! When we reach l_max we change equation (as stated in milestone 3 pdf)
       else if (l == (10 + lmax_nu)) then
          dydx(l)  = ckHp * Nu_(l_trick-1) - (l_trick + 1.d0) * c * Nu_(l_trick) /
              ↪  (H_p * eta)
       end if
    end do

    deallocate(Theta_)
    deallocate(ThetaP_)
    deallocate(Nu_)

  end subroutine equations_before_tight_coupling

! Routine for second part of integration, i.e. Integration after tight coupling
    ↪ ends
  subroutine equations_after_tight_coupling(x, y, dydx)
    implicit none

    ! Grid values
    real(dp),               intent(in)  :: x
    ! Function values
    real(dp), dimension(:), intent(in)  :: y
    ! Derivative values
    real(dp), dimension(:), intent(out) :: dydx
    ! All other parameters
    real(dp)                            :: k
    real(dp)                            :: a, ckHp, eta, H_p, dH_p, dtau, ddtau,
        ↪ dPhi_bracket
    real(dp)                            :: R, q, PI, delta, delta_b, v, v_b, Phi,
        ↪ Psi
    real(dp), allocatable, dimension(:) :: Theta_, ThetaP_, Nu_
    integer(i4b)                        :: l, l_trick, l1, l2

    allocate(Theta_(0:lmax_int))
    allocate(ThetaP_(0:lmax_int))
    allocate(Nu_(0:lmax_nu))

    !!!!!!!!!!!!!!!!!!!!!!!
    ! Necessary parameters !
    !!!!!!!!!!!!!!!!!!!!!!!
    k                = k_current
    a                = exp(x)
    R                = 4.d0 * Omega_r / (3.d0 * Omega_b * a)
    ! eta(x)
    eta              = get_eta(x)
    ! tau' and tau''
    dtau             = get_dtau(x)
    ddtau            = get_ddtau(x)
    ! H_p and H_p'
    H_p              = get_H_p(x)
```

```
    dH_p            = get_dH_p(x)
    ! To ease calculation
    ckHp            = c * k / get_H_p(x)

    ! Current values of functions (values on each step)
    delta           = y(1)
    delta_b         = y(2)
    v               = y(3)
    v_b             = y(4)
    Phi             = y(5)
    ! All Theta
    Theta_(:)       = y(6:6+lmax_int)
    ! Polarization (All ThetaP)
    ! Introducing new counter for better indexing
    l1 = 7 + lmax_int
    ThetaP_(:)      = y(l1:l1+lmax_int)
    ! Neutrinos (All Nu)
    ! And another counter for indexing Neutrinos
    l2 = l1 + lmax_int + 1
    Nu_(:)  = y(l2:l2+lmax_nu)

    PI              = Theta_(2) + ThetaP_(0) + ThetaP_(2)
    Psi             = -Phi - 12.d0 * (H_0 / (c * k * a))**2 * (Omega_r * Theta_(2)
        ↪  + Omega_nu * Nu_(2))

    !!!!!!!!!!!!!!!
    ! Derivatives !
    !!!!!!!!!!!!!!!
    ! Phi'
    dPhi_bracket    = Omega_m * a**(-1.d0) * delta + Omega_b * a**(-1.d0) * delta_b
        ↪  + 4.d0 * Omega_r * a**(-2.d0) * Theta_(0) + 4.d0* Omega_nu * a**(-2.d0
        ↪ ) * Nu_(0)
    dydx(5)         = Psi - (ckHp**2 / 3.d0) * Phi + (H_0 / H_p)**2 * dPhi_bracket
        ↪ / 2.d0
    ! delta'
    dydx(1)         = ckHp * v - 3.d0 * dydx(5)
    ! delta_b'
    dydx(2)         = ckHp * v_b - 3.d0 * dydx(5)
    ! velocity, v' and v_b'
    dydx(3)         = -v - ckHp * Psi
    dydx(4)         = -v_b - ckHp * Psi + dtau * R * (3.d0 * Theta_(1) + v_b) !
        ↪ correct
    ! Theta_0'
    dydx(6)         = -ckHp * Theta_(1) - dydx(5) ! correct
    ! Theta_1'
    dydx(7)         = (ckHp / 3.d0) * Theta_(0) - (2.d0 / 3.d0) * ckHp * Theta_(2)
        ↪ + (ckHp / 3.d0) * Psi + dtau * (Theta_(1) + v_b / 3.d0) ! correct
    ! All other Theta
    do l = (6 + 2), (6 + lmax_int)
       l_trick = l - 6
       if (l /= (6 + lmax_int)) then
          dydx(l) = l_trick / (2.d0 * l_trick + 1.d0) * ckHp * Theta_(l_trick-1) -
              ↪  (l_trick + 1.d0) / (2.d0 * l_trick + 1.d0) * ckHp * Theta_(
              ↪ l_trick+1) + dtau * Theta_(l_trick) ! correct
          ! condition with delta function
          if (l_trick == 2) then
             dydx(l)  = dydx(l) - (dtau * PI / 10.d0) ! correct
          end if
       else if (l_trick == lmax_int) then
          dydx(l)      = ckHp * Theta_(l_trick-1) - (l_trick + 1.d0) * c * Theta_(
              ↪ l_trick) / (H_p * eta) + dtau * Theta_(l_trick) ! correct
       end if
```

```
    end do

    ! ThetaP_0'
    dydx(l1)  = -ckHp * ThetaP_(1) + dtau * (ThetaP_(0) - PI / 2.d0) ! correct
    ! All other ThetaP
    do l = (l1 + 1) , (l1 + lmax_int)
       l_trick = l - l1
       if (l /= (l1 + lmax_int)) then
          dydx(l) = l_trick / (2.d0 * l_trick + 1.d0) * ckHp * ThetaP_(l_trick-1)
              ↪ - (l_trick + 1.d0) / (2.d0 * l_trick + 1.d0) * ckHp * ThetaP_(
              ↪ l_trick+1) + dtau * ThetaP_(l_trick)
          ! accounting for delta function
          if (l_trick == 2) then
             dydx(l) = dydx(l) - dtau * PI / 10.d0
          end if
       else if (l_trick == lmax_int) then
          dydx(l) = ckHp * ThetaP_(l_trick-1) - c * (l_trick + 1.d0) * ThetaP_(
              ↪ l_trick) / (H_p * eta) + dtau * ThetaP_(l_trick)
       end if
    end do

    ! Nu_0'
    dydx(l2)   = -ckHp * Nu_(1) - dydx(5)
    ! Nu_1'
    dydx(l2+1) = ckHp * Nu_(0) / 3.d0 - (2.d0 / 3.d0) * ckHp * Nu_(2) + ckHp * Psi
        ↪  / 3.d0
    ! All other Nu
    do l = (l2 + 2), (l2 + lmax_nu)
       l_trick = l - l2
       if (l /= (l2 + lmax_nu)) then
          dydx(l)  = ckHp * Nu_(l_trick-1) * l_trick / (2.d0 * l_trick + 1.d0) -
              ↪ ckHp * Nu_(l_trick+1) * (l_trick + 1.d0) / (2.d0 * l_trick + 1.d0
              ↪ )
       ! When we reach l_max we change equation (as stated in milestone 3 pdf)
       else if (l_trick == lmax_nu) then
          dydx(l)  = ckHp * Nu_(l_trick-1) - (l_trick + 1.d0) * c * Nu_(l_trick) /
              ↪  (H_p * eta)
       end if
    end do

    deallocate(Theta_)
    deallocate(ThetaP_)
    deallocate(Nu_)
  end subroutine equations_after_tight_coupling


  ! Routine for saving values into the .dat files
  subroutine save_data(k)
    implicit none

    integer(i4b),   intent(in) :: k ! index of k_current
    integer(i4b)               :: i
    character(len=1024)        :: folder, filename
    ! format descriptor
    character(len=1024)        :: format_string, k1

    folder = "data/"
    ! an integer of width 3 with zeros on the left if the value is not enough
    format_string = "(I3.3)"
    ! converting integer to string using a 'internal file'
    write (k1, format_string) k
```

```
filename = "delta_"//trim(k1)//"_x.dat"
open(1, file = trim(folder) // trim(filename), action = "write", status = "
    ↪ replace")
do i = 0, n_tot
   write(1,*) x_evol(i), "␣", delta(i, k)
end do
close(1)

filename = "deltab_"//trim(k1)//"_x.dat"
open(2, file = trim(folder) // trim(filename), action = "write", status = "
    ↪ replace")
do i = 0, n_tot
   write(2,*) x_evol(i), "␣", delta_b(i, k)
end do
close(2)

filename = "v_"//trim(k1)//"_x.dat"
open(3, file = trim(folder) // trim(filename), action = "write", status = "
    ↪ replace")
do i = 0, n_tot
   write(3,*) x_evol(i), "␣", v(i, k)
end do
close(3)

filename = "vb_"//trim(k1)//"_x.dat"
open(4, file = trim(folder) // trim(filename), action = "write", status = "
    ↪ replace")
do i = 0, n_tot
   write(4,*) x_evol(i), "␣", v_b(i, k)
end do
close(4)

! Phi
filename = "Phi_"//trim(k1)//"_x.dat"
open(5, file = trim(folder) // trim(filename), action = "write", status = "
    ↪ replace")
do i = 0, n_tot
   write(5,*) x_evol(i), "␣", Phi(i, k)
end do
close(5)
! Psi
filename = "Psi_"//trim(k1)//"_x.dat"
open(6, file = trim(folder) // trim(filename), action = "write", status = "
    ↪ replace")
do i = 0, n_tot
   write(6,*) x_evol(i), "␣", Psi(i, k)
end do
close(6)

filename = "Theta0_"//trim(k1)//"_x.dat"
open(7, file = trim(folder) // trim(filename), action = "write", status = "
    ↪ replace")
do i = 0, n_tot
   write(7,*) x_evol(i), "␣", Theta(i, 0, k)
end do
close(7)
filename = "Theta1_"//trim(k1)//"_x.dat"
open(8, file = trim(folder) // trim(filename), action = "write", status = "
    ↪ replace")
do i = 0, n_tot
   write(8,*) x_evol(i), "␣", Theta(i, 1, k)
end do
```

```
    close(8)
    filename = "ThetaP0_"//trim(k1)//"_x.dat"
    open(9, file = trim(folder) // trim(filename), action = "write", status = "
        ↪ replace")
    do i = 0, n_tot
       write(9,*) x_evol(i), "␣", ThetaP(i, 0, k)
    end do
    close(9)
    filename = "ThetaP1_"//trim(k1)//"_x.dat"
    open(10, file = trim(folder) // trim(filename), action = "write", status = "
        ↪ replace")
    do i = 0, n_tot
       write(10,*) x_evol(i), "␣", ThetaP(i, 1, k)
    end do
    close(10)
    filename = "Nu0_"//trim(k1)//"_x.dat"
    open(11, file = trim(folder) // trim(filename), action = "write", status = "
        ↪ replace")
    do i = 0, n_tot
       write(11,*) x_evol(i), "␣", Nu(i, 0, k)
    end do
    close(11)
    filename = "Nu1_"//trim(k1)//"_x.dat"
    open(12, file = trim(folder) // trim(filename), action = "write", status = "
        ↪ replace")
    do i = 0, n_tot
       write(12,*) x_evol(i), "␣", Nu(i, 1, k)
    end do
    close(12)

  end subroutine save_data


  ! This one is just for odeint to work
  subroutine output2(x, y)
    use healpix_types
    implicit none
    real(dp),                intent(in)  :: x
    real(dp), dimension(:), intent(in)  :: y
  end subroutine output2

end module evolution_mod
```

I've modified my plotting script from [1] to be able to plot functions for current assignment.

Listing 2: Milestone3_DataPlots.ipynb

```
import os
import sys
import numpy as np
# library for plotting
import plotly
import plotly.plotly as py
# for plotting in offline mode
import plotly.offline as plt
import plotly.graph_objs as go
# for making subplots
from plotly import tools
# for writing plots to a file
import plotly.io as pio
#from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot

plotly.__version__

# Getting the path to current (working) directory
currentDirPath = os.getcwd()
#print(dirpath)

# Checking if data dir exists
inputDir  = 'data'
outputDir = 'plots'
if (os.path.isdir(inputDir) == False):
    print("Data directory doesn't exist!")
    sys.exit(0)
else:
    # Checking if directory for outputting plots exists
    # if it doesn't it will create one
    if (os.path.isdir(outputDir) == False):
        os.mkdir(outputDir)

    plt.init_notebook_mode(connected = True)

    ######
    #Data#
    ######
    # Reading data from a file
    # Milestone 1
    omegaDataFile = np.loadtxt(inputDir + '/' + 'Omega_all_data.dat')
    etaDataFile   = np.loadtxt(inputDir + '/' + 'eta_x_data.dat')
    HxDataFile    = np.loadtxt(inputDir + '/' + 'H_x_data.dat')
    HzDataFile    = np.loadtxt(inputDir + '/' + 'H_z_data.dat')

    #etaDataFile   = np.loadtxt('eta_x_data.dat')

    # Milestone 1
    # Omega_X(x):
    X1            = omegaDataFile[:,0]
    Omega_b       = omegaDataFile[:,1]
    Omega_m       = omegaDataFile[:,2]
    Omega_r       = omegaDataFile[:,3]
    Omega_nu      = omegaDataFile[:,4]
    Omega_lambda  = omegaDataFile[:,5]
    # eta(x):
```

```
X2              = etaDataFile[:,0]
eta_x           = etaDataFile[:,1] / (3.08567758 * 10**(16)) # changing meters
    ↪ to pc
# H(x):
X3              = HxDataFile[:,0]
H_x             = HxDataFile[:,1]
# H(z):
Z               = HzDataFile[:,0]
H_z             = HzDataFile[:,1]


##########
#Plotting#
##########

#
    ↪ ##############################################################################################################
    ↪
# Milestone 1
#
    ↪ ##############################################################################################################
    ↪
# Create traces:
# Omega(x):
omegaB = go.Scatter(
    x = X1,
    y = Omega_b,
    name = '$\Omega_b$',
    line = dict(
        color = ('red'),#'rgb(100, 20, 50)'),
        width = 3))

omegaM = go.Scatter(
    x = X1,
    y = Omega_m,
    name = '$\Omega_m$',
    line = dict(
        color = ('orange'),#'rgb(205, 12, 24)'),
        width = 3))

omegaR = go.Scatter(
    x = X1,
    y = Omega_r,
    name = '$\Omega_r$',
    line = dict(
        color = ('blue'),#'rgb(300, 200, 100)'),
        width = 3))

omegaNu = go.Scatter(
    x = X1,
    y = Omega_nu,
    name = '$\Omega_\nu$',
    line = dict(
        color = ('green'),#'rgb(0, 15, 46)'),
        width = 3))

omegaL = go.Scatter(
    x = X1,
    y = Omega_lambda,
    name = '$\Omega_\Lambda$',
    line = dict(
        color = ('purple'),#'rgb(10, 40, 250)'),
```

```
            width = 3))

    # eta(x):
    etaX = go.Scatter(
        x = X2,
        y = eta_x,
        name = '$\eta(x)$',
        line = dict(
            color = ('blue'),#'rgb(100, 20, 50)'),
            width = 3))

    # H(x):
    Hx = go.Scatter(
        x = X3,
        y = H_x,
        name = '$H(x)$',
        line = dict(
            color = ('blue'),#'rgb(100, 20, 50)'),
            width = 3))

    # H(z):
    Hz = go.Scatter(
        x = Z,
        y = H_z,
        name = '$H(z)$',
        line = dict(
            color = ('blue'),#'rgb(100, 20, 50)'),
            width = 3))

    omegaDataPlot = [omegaB, omegaM, omegaR, omegaNu, omegaL]
    etaDataPlot   = [etaX]
    HxDataPlot    = [Hx]
    HzDataPlot    = [Hz]

    #figure = tools.make_subplots(rows = 2, cols = 2, subplot_titles = ('
        ↪ Cosmological Parameters', 'Conformal Time',
    #                                                                    'Plot 3',
        ↪ 'Plot 4'))

    #figure.add_traces(omegaDataPlot)
    #figure.add_traces(etaDataPlot)
    #figure.append_trace(trace3, 2, 1)
    #figure.append_trace(trace4, 2, 2)

    # Edit the layout
    omegaLayoutPlot = dict(#title = 'Cosmological Parameters',
                  xaxis = dict(title = '$x$', mirror = True, ticks = 'outside',
                               showline = True),
                  yaxis = dict(title = '$\Omega(x)$', type='log', autorange = True
                      ↪ ,
                               mirror = True, ticks = 'outside',
                               showline = True),
                  )
    etaLayoutPlot = dict(#title = 'Conformal Time',
                  xaxis = dict(title = '$x$',
                               mirror = True, ticks = 'outside',
                               showline = True),
                  yaxis = dict(title = '$\eta(x)␣[pc]$', type = 'log', autorange =
                      ↪  True,
                               mirror = True, ticks = 'outside',
                               showline = True),
                  )
```

```
    HxLayoutPlot  = dict(#title = 'Hubble Parameter',
                  xaxis = dict(title = '$x$', mirror = True, ticks = 'outside',
                               showline = True),
                  yaxis = dict(title = '$H(x)␣[s^{-1}]$', type = 'log', autorange
                      ↪ = True,
                               mirror = True, ticks = 'outside',
                               showline = True),
                  )
    HzLayoutPlot  = dict(#title = 'Hubble Parameter',
                  xaxis = dict(title = '$z$', mirror = True, ticks = 'outside',
                               showline = True),
                  yaxis = dict(title = '$H(z)␣[s^{-1}]$', type = 'log', autorange
                      ↪ = True,
                               mirror = True, ticks = 'outside',
                               showline = True),
                  )

    omegaFigure = dict(data = omegaDataPlot, layout = omegaLayoutPlot)
    etaFigure   = dict(data = etaDataPlot,   layout = etaLayoutPlot)
    HxFigure    = dict(data = HxDataPlot,    layout = HxLayoutPlot)
    HzFigure    = dict(data = HzDataPlot,    layout = HzLayoutPlot)

    # Plotting everything
    plt.iplot(omegaFigure, filename = 'omega')
    plt.iplot(etaFigure,   filename = 'eta')
    plt.iplot(HxFigure,    filename = 'Hx')
    plt.iplot(HzFigure,    filename = 'Hz')

    # Saving plots
    pio.write_image(omegaFigure, outputDir + '/' + 'Omega_x.pdf')
    pio.write_image(etaFigure,   outputDir + '/' + 'eta_x.pdf')
    pio.write_image(HxFigure,    outputDir + '/' + 'H_x.pdf')
    pio.write_image(HzFigure,    outputDir + '/' + 'H_z.pdf')

    #
        ↪ ##########################################################################################
        ↪
    # Milestone 2
    #
        ↪ ##########################################################################################
        ↪
    XeDataFile       = np.loadtxt(inputDir + '/' + "X_e_z.dat")

    tauDataFile      = np.loadtxt(inputDir + '/' + "tau_x.dat")
    dtauDataFile     = np.loadtxt(inputDir + '/' + "dtau_x.dat")
    ddtauDataFile    = np.loadtxt(inputDir + '/' + "ddtau_x.dat")

    gtildeDataFile   = np.loadtxt(inputDir + '/' + "g_x.dat")
    dgtildeDataFile  = np.loadtxt(inputDir + '/' + "dg_x.dat")
    ddgtildeDataFile = np.loadtxt(inputDir + '/' + "ddg_x.dat")

    # X_e(z):
    X4               = XeDataFile[:,0]
    Xe               = XeDataFile[:,1]
    # tau(x) and the derivatives:
    X5               = tauDataFile[:,0]
    tau              = tauDataFile[:,1]
    dtau             = np.abs(dtauDataFile[:,1])
    ddtau            = ddtauDataFile[:,1]
    # g(x) and its derivatives:
    X6               = gtildeDataFile[:,0]
    gtilde           = gtildeDataFile[:,1]
```

```
dgtilde           = dgtildeDataFile[:,1]/10
ddgtilde          = ddgtildeDataFile[:,1]/200

# X_e(z):
XeTrace = go.Scatter(
    x = X4,
    y = Xe,
    name = '$X_e$',
    line = dict(
        color = ('blue'),#'rgb(300, 200, 100)'),
        width = 3))

# tau(x) and its derivatives:
tauTrace = go.Scatter(
    x = X5,
    y = tau,
    name = '$\\tau$',
    line = dict(
        color = ('red'),#'rgb(100, 20, 50)'),
        width = 3))

dtauTrace = go.Scatter(
    x = X5,
    y = dtau,
    name = "$|\\tau'(x)|$",
    line = dict(
        color = ('orange'),#'rgb(205, 12, 24)'),
        width = 3,
        dash  = "dash"))

ddtauTrace = go.Scatter(
    x = X5,
    y = ddtau,
    name = "$\\tau''(x)$",
    line = dict(
        color = ('blue'),#'rgb(300, 200, 100)'),
        width = 3,
        dash  = "dot"))

# g(x) and its derivatives:

gtildeTrace = go.Scatter(
    x = X6,
    y = gtilde,
    name = '$g(x)$',
    line = dict(
        color = ('red'),#'rgb(100, 20, 50)'),
        width = 3))

dgtildeTrace = go.Scatter(
    x = X6,
    y = dgtilde,
    name = "$g'(x)$",
    line = dict(
        color = ('orange'),#'rgb(205, 12, 24)'),
        width = 3,
        dash  = "dash"))

ddgtildeTrace = go.Scatter(
    x = X6,
    y = ddgtilde,
    name = "$g''(x)$",
```

```
        line = dict(
            color = ('blue'),#'rgb(300, 200, 100)'),
            width = 3,
            dash  = "dot"))

    XeDataPlot     = [XeTrace]
    tauDataPlot    = [tauTrace, dtauTrace, ddtauTrace]
    gtildeDataPlot = [gtildeTrace, dgtildeTrace, ddgtildeTrace]

    XeLayoutPlot  = dict(#title = 'Xe',
                   xaxis = dict(title = '$z$', range = [1800, 0],
                                mirror = True, ticks = 'outside',
                                showline = True),
                   yaxis = dict(title = '$X_e$', type = 'log', autorange = True,
                                mirror = True, ticks = 'outside',
                                showline = True),
                   )

    tauLayoutPlot = dict(#title = 'tau(x) and its derivatives',
                   xaxis = dict(title = '$x$', range =[-20, 1],
                                mirror = True, ticks = 'outside',
                                showline = True),
                   yaxis = dict(title = "$\\tau(x),␣|\\tau'(x)|,␣\\tau''(x)$",
                                type = 'log', autorange = True,
                                mirror = True, ticks = 'outside',
                                showline = True)
                   )

    gtildeLayoutPlot = dict(#title = 'tau(x) and its derivatives',
                   xaxis = dict(title = '$x$', range = [-9, -5],
                                mirror = True, ticks = 'outside',
                                showline = True,),
                   yaxis = dict(title = "$\\tilde{g}(x),␣\\tilde{g}'(x),␣\\tilde{g
                       ↪ }''(x)$",
                                type = 'linear', autorange = True,
                                mirror = True, ticks = 'outside',
                                showline = True)
                   )

    XeFigure     = dict(data = XeDataPlot, layout = XeLayoutPlot)
    tauFigure    = dict(data = tauDataPlot, layout = tauLayoutPlot)
    gtildeFigure = dict(data = gtildeDataPlot, layout = gtildeLayoutPlot)

    # Plotting everything
    plt.iplot(XeFigure,     filename = 'Xe')
    plt.iplot(tauFigure,    filename = 'tau')
    plt.iplot(gtildeFigure, filename = 'gtilde')

    # Saving plots
    pio.write_image(XeFigure,     outputDir + '/' + 'X_e_z.pdf')
    pio.write_image(tauFigure,    outputDir + '/' + 'tau_x.pdf')
    pio.write_image(gtildeFigure, outputDir + '/' + 'g_x.pdf')

    #
        ↪ ##########################################################################################
        ↪
    # Milestone 3
    #
        ↪ ##########################################################################################
        ↪

    # delta(x):
```

```
    deltaDataFile  = [
                     np.loadtxt(inputDir + '/' + "delta_001_x.dat"),
                     np.loadtxt(inputDir + '/' + "delta_010_x.dat"),
                     np.loadtxt(inputDir + '/' + "delta_022_x.dat"),
                     np.loadtxt(inputDir + '/' + "delta_040_x.dat"),
                     np.loadtxt(inputDir + '/' + "delta_060_x.dat"),
                     np.loadtxt(inputDir + '/' + "delta_100_x.dat")]

    deltabDataFile = [
                     np.loadtxt(inputDir + '/' + "deltab_001_x.dat"),
                     np.loadtxt(inputDir + '/' + "deltab_010_x.dat"),
                     np.loadtxt(inputDir + '/' + "deltab_022_x.dat"),
                     np.loadtxt(inputDir + '/' + "deltab_040_x.dat"),
                     np.loadtxt(inputDir + '/' + "deltab_060_x.dat"),
                     np.loadtxt(inputDir + '/' + "deltab_100_x.dat")]

    # v(x):
    vDataFile      = [
                     np.loadtxt(inputDir + '/' + "v_001_x.dat"),
                     np.loadtxt(inputDir + '/' + "v_010_x.dat"),
                     np.loadtxt(inputDir + '/' + "v_022_x.dat"),
                     np.loadtxt(inputDir + '/' + "v_040_x.dat"),
                     np.loadtxt(inputDir + '/' + "v_060_x.dat"),
                     np.loadtxt(inputDir + '/' + "v_100_x.dat")]

    # v_b(x):
    vbDataFile = [
                     np.loadtxt(inputDir + '/' + "vb_001_x.dat"),
                     np.loadtxt(inputDir + '/' + "vb_010_x.dat"),
                     np.loadtxt(inputDir + '/' + "vb_022_x.dat"),
                     np.loadtxt(inputDir + '/' + "vb_040_x.dat"),
                     np.loadtxt(inputDir + '/' + "vb_060_x.dat"),
                     np.loadtxt(inputDir + '/' + "vb_100_x.dat")]

    # Phi(x):
    PhiDataFile = [
                     np.loadtxt(inputDir + '/' + "Phi_001_x.dat"),
                     np.loadtxt(inputDir + '/' + "Phi_010_x.dat"),
                     np.loadtxt(inputDir + '/' + "Phi_022_x.dat"),
                     np.loadtxt(inputDir + '/' + "Phi_040_x.dat"),
                     np.loadtxt(inputDir + '/' + "Phi_060_x.dat"),
                     np.loadtxt(inputDir + '/' + "Phi_100_x.dat")]

    # Psi(x):
    PsiDataFile = [
                     np.loadtxt(inputDir + '/' + "Psi_001_x.dat"),
                     np.loadtxt(inputDir + '/' + "Psi_010_x.dat"),
                     np.loadtxt(inputDir + '/' + "Psi_022_x.dat"),
                     np.loadtxt(inputDir + '/' + "Psi_040_x.dat"),
                     np.loadtxt(inputDir + '/' + "Psi_060_x.dat"),
                     np.loadtxt(inputDir + '/' + "Psi_100_x.dat")]

    # Theta_0(x):
    Theta0DataFile = [
                     np.loadtxt(inputDir + '/' + "Theta0_001_x.dat"),
                     np.loadtxt(inputDir + '/' + "Theta0_010_x.dat"),
                     np.loadtxt(inputDir + '/' + "Theta0_022_x.dat"),
                     np.loadtxt(inputDir + '/' + "Theta0_040_x.dat"),
                     np.loadtxt(inputDir + '/' + "Theta0_060_x.dat"),
                     np.loadtxt(inputDir + '/' + "Theta0_100_x.dat")]
    # Theta_1(x):
    Theta1DataFile  = [
```

```
                np.loadtxt(inputDir + '/' + "Theta1_001_x.dat"),
                np.loadtxt(inputDir + '/' + "Theta1_010_x.dat"),
                np.loadtxt(inputDir + '/' + "Theta1_022_x.dat"),
                np.loadtxt(inputDir + '/' + "Theta1_040_x.dat"),
                np.loadtxt(inputDir + '/' + "Theta1_060_x.dat"),
                np.loadtxt(inputDir + '/' + "Theta1_100_x.dat")]

# ThetaP_0(x):
ThetaP0DataFile = [
                np.loadtxt(inputDir + '/' + "ThetaP0_001_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP0_010_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP0_022_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP0_040_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP0_060_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP0_100_x.dat")]

# ThetaP_1(x):
ThetaP1DataFile = [
                np.loadtxt(inputDir + '/' + "ThetaP1_001_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP1_010_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP1_022_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP1_040_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP1_060_x.dat"),
                np.loadtxt(inputDir + '/' + "ThetaP1_100_x.dat")]

# Nu_0(x):
Nu0DataFile = [
                np.loadtxt(inputDir + '/' + "Nu0_001_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu0_010_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu0_022_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu0_040_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu0_060_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu0_100_x.dat")]

# Nu_1(x):
Nu1DataFile = [
                np.loadtxt(inputDir + '/' + "Nu1_001_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu1_010_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu1_022_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu1_040_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu1_060_x.dat"),
                np.loadtxt(inputDir + '/' + "Nu1_100_x.dat")]

Colors = ['blue', 'green', 'yellow', 'orange', 'red', 'purple', 'black']
k = ['$k␣=␣0.1␣H_0␣/␣c$', '$k␣=␣10␣H_0␣/␣c$', '$k␣=␣50␣H_0␣/␣c$',
     '$k␣=␣160␣H_0␣/␣c$', '$k␣=␣360␣H_0␣/␣c$','$k␣=␣1000␣H_0␣/␣c$']

# Grid:
X7            = [sublist[0] for sublist in deltaDataFile[1]]
# Values:
delta         = []

deltab        = []
deltab1        = []
deltab2        = []

v             = []
vb            = []
Phi           = []
Psi           = []
Theta0        = []
Theta1        = []
```

```
    ThetaP0       = []
    ThetaP1       = []
    Nu0           = []
    Nu1           = []
    # Traces:
    deltaTrace    = []

    deltabTrace   = []
    deltabTrace1   = []
    deltabTrace2   = []

    vTrace        = []
    vbTrace       = []
    PhiTrace      = []
    PsiTrace      = []
    Theta0Trace   = []
    ThetaP0Trace  = []
    Theta1Trace   = []
    ThetaP1Trace  = []
    Nu0Trace      = []
    Nu1Trace      = []
    # looping through all files
    for i in range(0, 6):
        # Appending the values of delta into an array(i.e. the list of lists)
        delta.append([sublist[1] for sublist in deltaDataFile[i]])

        # Applying trick to show the negative values on y axes
        #deltab.append([np.arcsinh(sublist[1]) for sublist in deltabDataFile[i]])
        deltab.append([np.arcsinh(sublist[1]) for sublist in deltabDataFile[i]])
        deltab1.append([sublist[1] for sublist in deltabDataFile[i]])
        deltab2.append([np.arcsinh(sublist[1]) for sublist in deltabDataFile[i]])
        #for j in range(0, (len(deltab)-1)):
            #print(deltab[i][j])
        #    if (deltab[i][j] < 0):
        #        deltab.append(-np.log(-deltab[j]))
                #print(deltab[i][j])
        #    elif(deltab[i][j] == 0):
        #        deltab.append(0)
        #    elif (deltab[i][j] > 0):
        #        deltab.append(np.log(deltab[j]))


        v.append([sublist[1] for sublist in vDataFile[i]])
        vb.append([sublist[1] for sublist in vbDataFile[i]])
        Phi.append([sublist[1] for sublist in PhiDataFile[i]])
        Psi.append([sublist[1] for sublist in PsiDataFile[i]])
        Theta0.append([sublist[1] for sublist in Theta0DataFile[i]])
        Theta1.append([sublist[1] for sublist in Theta1DataFile[i]])
        ThetaP0.append([sublist[1] for sublist in ThetaP0DataFile[i]])
        ThetaP1.append([sublist[1] for sublist in ThetaP1DataFile[i]])
        Nu0.append([sublist[1] for sublist in Nu0DataFile[i]])
        Nu1.append([sublist[1] for sublist in Nu1DataFile[i]])

        # Feeding the plots with the values
        deltaTrace.append(
            go.Scatter(
                x = X7,
                y = delta[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
```

```
        deltabTrace.append(
            go.Scatter(
                x = X7,
                y = deltab[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        deltabTrace1.append(
            go.Scatter(
                x = X7,
                y = deltab1[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        deltabTrace2.append(
            go.Scatter(
                x = X7,
                y = deltab2[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))

        vTrace.append(
            go.Scatter(
                x = X7,
                y = v[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        vbTrace.append(
            go.Scatter(
                x = X7,
                y = vb[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        PhiTrace.append(
            go.Scatter(
                x = X7,
                y = Phi[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        PsiTrace.append(
            go.Scatter(
                x = X7,
                y = Psi[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        Theta0Trace.append(
            go.Scatter(
                x = X7,
                y = Theta0[i],
```

```
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        Theta1Trace.append(
            go.Scatter(
                x = X7,
                y = Theta1[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        ThetaP0Trace.append(
            go.Scatter(
                x = X7,
                y = ThetaP0[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        ThetaP1Trace.append(
            go.Scatter(
                x = X7,
                y = ThetaP1[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        Nu0Trace.append(
            go.Scatter(
                x = X7,
                y = Nu0[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))
        Nu1Trace.append(
            go.Scatter(
                x = X7,
                y = Nu1[i],
                name = k[i],
                line = dict(
                    color = (Colors[i]),#'rgb(300, 200, 100)'),
                    width = 3)))

    # delta(x):
    deltaDataPlot   = deltaTrace
    #flat_list = [item for sublist in l for item in sublist]

    # delta_b(x):
    deltabDataPlot  = deltabTrace
    deltabDataPlot1  = deltabTrace1
    deltabDataPlot2  = deltabTrace2
    # v(x):
    vDataPlot       = vTrace
    # v_b(x):
    vbDataPlot      = vbTrace
    # Phi(x):
    PhiDataPlot     = PhiTrace
    # Psi(x):
    PsiDataPlot     = PsiTrace
    # Theta(x):
```

```
Theta0DataPlot  = Theta0Trace
Theta1DataPlot  = Theta1Trace
# Theta^P(x):
ThetaP0DataPlot = ThetaP0Trace
ThetaP1DataPlot = ThetaP1Trace
# N(x):
Nu0DataPlot     = Nu0Trace
Nu1DataPlot     = Nu1Trace

# Creating a position for each legend
legendPositionTop = [0.02, 0.96]
legendPositionBot = [0.02, 0.04]

deltaLayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                             mirror = True, ticks = 'outside',
                             showline = True),
                yaxis = dict(title = '$\delta(x)$', type = 'log', autorange =
                    ↪ True,
                             mirror = True, ticks = 'outside',
                             # To show values as number x 10^*
                             showexponent = 'all',
                             exponentformat = 'power',
                             #tickformat = 'power',
                             showline = True),
                # Tell where to put legends (i.e. labels of k values)
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                #height = 600,
                #width = 600,
                )

deltaLayoutPlot3  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                             mirror = True, ticks = 'outside',
                             showline = True),
                yaxis = dict(title = '$\delta(x)$', type = 'log', autorange =
                    ↪ True,
                             mirror = True, ticks = 'outside',
                             # To show values as number x 10^*
                             showexponent = 'all',
                             exponentformat = 'power',
                             #tickformat = 'power',
                             showline = True),
                # Tell where to put legends (i.e. labels of k values)
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                #height = 600,
```

```
                #width = 600,
                )

    # Recover the values on the y axes for delta_b plot
    #deltabAxesValues = []
    #for i in range(0, 6):
        #for j in range(0, (len(deltab[i])-1)):
    #    deltabAxesValues.append([np.sinh(deltab[i])])
    #    print(deltabAxesValues[i])
    deltabAxesValues = [-4, -2, 0, 2, 4, 6, 8, 10, 12]
    deltabAxesText   = ['$-10^1$', '$-10^0$','$0$', '$10^0$', '$10^1$', '$10^2$',
        ↪ '$10^3$', '$10^4$', '$10^5$']
    #k = -2
    deltabLayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                             mirror = True, ticks = 'outside',
                             showline = True),
                yaxis = dict(title = '$\delta_b(x)$', type = 'linear', autorange
                    ↪  = True,
                             mirror = True, ticks = 'outside',
                             #tickvals = [(k+2) for k in range(-2, 13)],
                             #ticktext = ['$-10^1$', '$-10^0$','$0$', '$-10^1$',
                                 ↪  '$-10^1$', '$-10^1$'],
                             tickvals = deltabAxesValues,
                             ticktext = deltabAxesText,
                             tickfont = dict(
                                  family = 'Courier␣New,␣monospace',
                                  size   = 12,
                                  color  = 'black'
                              ),
                             # To show values as number x 10^*
                             #showexponent = 'all',
                             #exponentformat = 'power',
                             showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
    deltabLayoutPlot1  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                             mirror = True, ticks = 'outside',
                             showline = True),
                yaxis = dict(title = '$\delta_b(x)$', type = 'log', autorange =
                    ↪ True,
                             mirror = True, ticks = 'outside',
                             # To show values as number x 10^*
                             showexponent = 'all',
                             exponentformat = 'power',
                             showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
```

```
                    borderwidth = 2
                ),
                )
    deltabLayoutPlot2  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\delta_b(x)$', type = 'linear', autorange
                    ↪  = True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
    vLayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$v(x)$', type = 'linear', autorange = True
                    ↪ ,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
    vbLayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$v_b(x)$', type = 'linear', autorange =
                    ↪ True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
```

```
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )

    PhiLayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\Phi(x)$', type = 'linear', autorange =
                    ↪ True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                legend = dict(
                    x = legendPositionBot[0],
                    y = legendPositionBot[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
    PsiLayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\Psi(x)$', type = 'linear', autorange =
                    ↪ True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )


    Theta0LayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\Theta_0(x)$', type = 'linear', autorange
                    ↪  = True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionBot[0],
                    y = legendPositionBot[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
```

```
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
    Theta1LayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\Theta_1(x)$', type = 'linear', autorange
                    ↪  = True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
    ThetaP0LayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\Theta^P_0(x)$', type = 'linear',
                    ↪ autorange = True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
    ThetaP1LayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\Theta^P_1(x)$', type = 'linear',
                    ↪ autorange = True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
```

```
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
Nu0LayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\mathcal{N}_0(x)$', type = 'linear',
                    ↪ autorange = True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )
Nu1LayoutPlot  = dict(#title = '$\delta(x)$',
                xaxis = dict(title = '$x$', autorange = True,
                            mirror = True, ticks = 'outside',
                            showline = True),
                yaxis = dict(title = '$\mathcal{N}_1(x)$', type = 'linear',
                    ↪ autorange = True,
                            mirror = True, ticks = 'outside',
                            # To show values as number x 10^*
                            showexponent = 'all',
                            exponentformat = 'power',
                            showline = True),
                legend = dict(
                    x = legendPositionTop[0],
                    y = legendPositionTop[1],
                    traceorder = "normal",
                    font = dict(size = 12, color = 'black'),
                    bgcolor = "White",
                    bordercolor = 'Black',
                    borderwidth = 2
                ),
                )

deltaFigure   = dict(data = deltaDataPlot, layout = deltaLayoutPlot)

deltabFigure  = dict(data = deltabDataPlot, layout = deltabLayoutPlot)
deltabFigure1 = dict(data = deltabDataPlot1, layout = deltabLayoutPlot1)
deltabFigure2 = dict(data = deltabDataPlot2, layout = deltabLayoutPlot2)

vFigure       = dict(data = vDataPlot, layout = vLayoutPlot)
vbFigure      = dict(data = vbDataPlot, layout = vbLayoutPlot)

PhiFigure     = dict(data = PhiDataPlot, layout = PhiLayoutPlot)
PsiFigure     = dict(data = PsiDataPlot, layout = PsiLayoutPlot)

Theta0Figure  = dict(data = Theta0DataPlot, layout = Theta0LayoutPlot)
```

```
Theta1Figure  = dict(data = Theta1DataPlot, layout = Theta1LayoutPlot)
ThetaP0Figure = dict(data = ThetaP0DataPlot, layout = ThetaP0LayoutPlot)
ThetaP1Figure     = dict(data = ThetaP1DataPlot, layout = ThetaP1LayoutPlot)
#PhiFigure     = dict(data = PhiDataPlot, layout = PhiLayoutPlot)
Nu0Figure     = dict(data = Nu0DataPlot, layout = Nu0LayoutPlot)
Nu1Figure     = dict(data = Nu1DataPlot, layout = Nu1LayoutPlot)

# Plotting everything
plt.iplot(deltaFigure,   filename = 'delta_x')

plt.iplot(deltabFigure,  filename = 'deltab_x')
#plt.iplot(deltabFigure1,  filename = 'deltab1_x')
#plt.iplot(deltabFigure2,  filename = 'deltab2_x')

plt.iplot(vFigure,       filename = 'v_x')
plt.iplot(vbFigure,      filename = 'vb_x')

plt.iplot(PhiFigure,     filename = 'Phi_x')
plt.iplot(PsiFigure,     filename = 'Psi_x')

plt.iplot(Theta0Figure,  filename = 'Theta0_x')
plt.iplot(Theta1Figure,  filename = 'Theta1_x')
plt.iplot(ThetaP0Figure, filename = 'ThetaP0_x')
plt.iplot(ThetaP1Figure, filename = 'ThetaP1_x')
plt.iplot(Nu0Figure,     filename = 'Nu0_x')
plt.iplot(Nu1Figure,     filename = 'Nu1_x')

# Saving plots (to the plots directory)
pio.write_image(deltaFigure,   outputDir + '/' + 'delta_x.pdf')

pio.write_image(deltabFigure,  outputDir + '/' + 'deltab_x.pdf')
#pio.write_image(deltabFigure1,  outputDir + '/' + 'deltab1_x.pdf')
#pio.write_image(deltabFigure2,  outputDir + '/' + 'deltab2_x.pdf')

pio.write_image(vFigure,       outputDir + '/' + 'v_x.pdf')
pio.write_image(vbFigure,      outputDir + '/' + 'vb_x.pdf')
pio.write_image(PhiFigure,     outputDir + '/' + 'Phi_x.pdf')
pio.write_image(PsiFigure,     outputDir + '/' + 'Psi_x.pdf')
pio.write_image(Theta0Figure,  outputDir + '/' + 'Theta0_x.pdf')
pio.write_image(Theta1Figure,  outputDir + '/' + 'Theta1_x.pdf')
pio.write_image(ThetaP0Figure, outputDir + '/' + 'ThetaP0_x.pdf')
pio.write_image(ThetaP1Figure, outputDir + '/' + 'ThetaP1_x.pdf')
pio.write_image(Nu0Figure,     outputDir + '/' + 'Nu0_x.pdf')
pio.write_image(Nu1Figure,     outputDir + '/' + 'Nu1_x.pdf')
```