UNIVERSITY OF OSLO

Report for AST9240: The recombination history of the universe

Maksym Brilenkov

April 1, 2019

1 Introduction

The calculation of CMB power spectrum is the final goal of the project. To achieve this aim, it is essential to understand the evolution of background properties of the universe and its ionization history. The first part was addressed previously (Ref. [1]), where I have already looked into the background geometry and other essential parameters. Therefore, this report is dedicated to recombination history of the universe.

On practice, this involves calculation of *optical depth*, τ, and so-called *visibility function*, g, which is essentially the probability that a CMB photon observed today was last scattered at conformal time η (Ref. [2]). By definition (Ref. [2, 3]) optical depth is

$$\tau = \int_{\eta}^{\eta_0} n_e \sigma_T a d\eta' , \tag{1.1}$$

and its first derivative is

$$\dot{\tau} = -n_e \sigma_T a \quad \rightarrow \quad \tau' = -\frac{n_e \sigma_T a}{\mathscr{H}} , \tag{1.2}$$

where n_e is electron number density, a is a scale factor, $\sigma_T = \frac{8\pi\alpha^2}{3m_e^2} = 6.652462 \times 10^{-29}$ m^2 is the Thomson cross section [2] and $\mathscr{H} = aH$ defined previously [1] as

$$\mathscr{H}(a) = H_0 \sqrt{(\Omega_m + \Omega_b) a^{-1} + (\Omega_r + \Omega_\nu) a^{-2} + \Omega_\Lambda a^2} . \tag{1.3}$$

The visibility function is (Ref. [2])

$$g(\eta) = -\dot{\tau} e^{-\tau(\eta)} = -\mathscr{H} \tau' e^{-\tau(x)} = g(x) , \tag{1.4}$$

or

$$\tilde{g}(x) = -\tau' e^{-\tau} = \frac{g(x)}{\mathscr{H}} , \tag{1.5}$$

The photons we observe today are travelled freely from the *surface of last scattering*, which corresponds to a peak of g at redshift $z \sim 1100$. Before this time (i.e. above $T \sim 3000$ K), universe was opaque due to the Thomson scattering of baryon-photon plasma.

The report organized as follows. In section 2, I describe the general theory behind computations, while section 3 is dedicated for numerical algorithms used to compute $X_e(x)$, $\tau(x)$, $\tilde{g}(x)$ and their derivatives. Section 4 is dedicated for results. Code is listed in the end of this report.

2 METHODS

To compute τ, we need to know the electron number density, n_e, which, in principle, hard to compute at any given time; thus, I concentrate on a fractional electron density

$$X_e = \frac{n_e}{n_H}. \tag{2.1}$$

where, assuming all baryons are protons (Ref. [2]), the proton number density is equal to

$$n_H = n_b \approx \frac{\rho_b}{m_H} = \frac{\Omega_b \rho_c}{m_H a^3}. \tag{2.2}$$

Here $\rho_{\rm crit} = \frac{3H_0^2}{8\pi G}$ is the critical density of the universe, m_H is the mass of hydrogen atom.

Before recombination it is possible to calculate X_e using Saha equation (see [2] and reference therein)

$$\frac{X_e^2}{1-X_e} \approxeq \frac{1}{n_b}\left(\frac{m_e k_B T_b}{2\pi\hbar^2}\right)^{3/2} \exp\left(-\frac{\epsilon_0}{k_B T_b}\right), \tag{2.3}$$

which can be easily transformed in the quadratic equation form

$$ax^2 + bx + c = 0, \tag{2.4}$$

with the solution

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2.5}$$

In our case, the coefficients are as follows

$$a = 1, \quad b = \frac{1}{n_b}\left(\frac{m_e k_B T_b}{2\pi\hbar^2}\right)^{3/2} \exp\left(-\frac{\epsilon_0}{k_B T_b}\right), \quad c = -b, \tag{2.6}$$

During and after recombination we use Peebles equation instead (Ref. [2]), which reads

$$\frac{dX_e}{dx} = \frac{C_r(T_b)}{H}\left[\beta(T_b)\,(1-X_e) - n_H \frac{\hbar^2}{c}\alpha^{(2)}(T_b) X_e^2\right], \tag{2.7}$$

where

$$C_r(T_b) = \frac{\Lambda_{2s1s} + \Lambda_\alpha}{\Lambda_{2s1s} + \Lambda_\alpha + \beta^{(2)}(T_b)}, \tag{2.8}$$

$$\Lambda_{2s\to 1s} = 8.227 \mathrm{s}^{-1}, \quad \Lambda_\alpha = H\frac{(3\epsilon_0)^3}{(8\pi)^2 n_{1s} (\hbar c)^3}, \tag{2.9}$$

$$n_{1s} = (1-X_e)\, n_H, \quad \beta^{(2)}(T_b) = \beta(T_b)\exp\left(\frac{3\epsilon_0}{4k_B T_b}\right), \tag{2.10}$$

$$\beta(T_b) = \frac{\alpha^{(2)}(T_b)}{c\hbar}\left(\frac{m_e k_B T_b}{2\pi}\right)^{3/2} \exp\left(-\frac{\epsilon_0}{k_B T_b}\right), \tag{2.11}$$

$$\alpha^{(2)}(T_b) = \frac{64\pi}{\sqrt{27\pi}}\left(\frac{\alpha}{m_e}\right)^2 \sqrt{\frac{\epsilon_0}{k_B T_b}}\phi_2(T_b), \tag{2.12}$$

$$\phi_2(T_b) = 0.448 \ln\left(\frac{\epsilon_0}{k_B T_b}\right), \tag{2.13}$$

Here T_b is baryon temperature, H is Hubble constant, k_B is Boltzmann constant, $\hbar$ is the Dirac constant, m_e is the electron mass, $\alpha = e^2/(4\pi\epsilon_0\hbar c) = 7.29735 \cdot 10^{-3}$ is the fine structure constant and $\epsilon_0 = 13.605698$ eV is the ionization energy of hydrogen. In Callin *et al* [2] quantities were provided in natural units, so I needed to slightly change equations (2.7)-(2.13) to match the dimensions.

To describe the baryon temperature we use an approximation (see [2] for details)

$$T_b = T_r = \frac{T}{a}, \quad T_0 = 2.725\text{K}, \tag{2.14}$$

With X_e in hands I proceed to compute n_e as

$$n_e(x) = \frac{\Omega_b \rho_c}{m_H a^3} X_e(x). \tag{2.15}$$

Knowing n_e at any given moment, it is possible now to calculate $\tau(x)$, $g(x)$ and its derivatives numerically. More on it in the next section.

3 Algorithms

The main working module is "rec_mod.f90", which stands for *recombination module*. The idea for numerical computations are nicely explained by Eriksen *et al* [4], which, in my case, can be visualized as follows:

- The preliminary step is, as always, to set up a grid

```
dx = (xstop - xstart) / (n - 1)
x_rec(1) = xstart
do i = 2, n
   x_rec(i) = x_rec(1) + dx * (i - 1)
end do
a_rec = exp(x_rec)
```

  where I choose the total number of grid points, n, to be = 1000.

- I calculate X_e via Eqs.(2.3) and (2.7) by looping through the respective values. I've chosen to switch between these two equations when X_e reaches the value of 0.99 (for explanation, please refer to Callin *et al* [2]). The respective piece of code is

```
use_saha = .true.
! for calculating Peebles eq. through ode solver
h1 = 1.d-6
h_min = 0.d0
eps = 1.d-9
do i = 1, n
   ! Baryon temperature
   T_b = T_0 / exp(x_rec(i))
   ! Baryon number density
   n_b = (Omega_b * rho_c) / (m_H * exp(3.d0 * x_rec(i)))
   if (use_saha) then
      ! Calculate constant in Saha equation
      saha_const = (m_e * k_b * T_b / (2.d0 * pi))**1.5d0 *
         ↪ exp(-epsilon_0 / (k_b * T_b)) / (n_b * hbar**3.d0)
      ! Use Saha equation
      X_e(i) = (-saha_const + sqrt(saha_const**2.d0 + 4.d0 *
         ↪ saha_const)) / 2.d0
      if (X_e(i) < saha_limit) then
         use_saha = .false.
         Xe_i(1) = X_e(i)
      end if
   else
      ! Use the Peebles equation
      call odeint(Xe_i, x_rec(i-1), x_rec(i), eps, h1, h_min,
         ↪ Peebles, bsstep, output1)
      X_e(i) = Xe_i(1)
   end if
end do
```

  where subroutine "Peebles" has the form

```
    subroutine Peebles(x, X_e, dXedx)
      use healpix_types
      implicit none
      real(dp),                intent(in)  :: x
      real(dp), dimension(:), intent(in)  :: X_e
      real(dp), dimension(:), intent(out) :: dXedx
      ! Other variables
      integer(i4b)                         :: i
      ! Preliminary variables
      real(dp)                             :: H, T_b, dummy_const,
         ↪ phi2, n_b
      ! Calculatable variables
      real(dp)                             :: alpha2, lambda_2s1s
      real(dp)                             :: beta, beta2
      real(dp), dimension(:), allocatable :: lambda_alpha, n_1s,
         ↪ C_r
      ! lambda_2s->1s [s^-1]
      lambda_2s1s = 8.227d0
      ! Taking the value of Hubble constant from time_mod (
         ↪ milestone 1) [s^-1]
      H = get_H(x)
      ! Baryon temperature [K]
      T_b = T_0 / exp(x)
      ! Baryon number density [m^-3]
      n_b = Omega_b * rho_c / (m_H * exp(3 * x))
      ! e/kT constant [dimensionless]
      dummy_const = epsilon_0 / (k_b * T_b)
      ! phi2 [dimensionless]
      phi2 = 0.448d0 * log(dummy_const)
      ! alpha2 [kg^-2]
      alpha2 = (64 * pi / (sqrt(27 * pi))) * (alpha / m_e)**2 *
         ↪ sqrt(dummy_const) * phi2
      ! beta [s^-1]
      beta = (alpha2 / (hbar * c)) * (m_e * k_b * T_b / (2 * pi))
         ↪ **1.5d0 * exp(-dummy_const)
      ! beta2 [s^-1]
      beta2 = (alpha2 / (hbar * c)) * (m_e * k_b * T_b / (2 * pi))
         ↪ **1.5 * exp(-dummy_const / 4.d0)
      ! n1s [m^-3]
      n_1s = (1 - X_e) * n_b
      ! Calculating Lambda_alpha [s^-1]
      lambda_alpha = H * (3 * epsilon_0)**3 / (n_1s * (8 * pi)**2
         ↪ * (hbar * c)**3)
      ! C_r [dimensionless]
      C_r = (lambda_2s1s + lambda_alpha) / (lambda_2s1s +
         ↪ lambda_alpha + beta2)
      ! dX_e/dx by Peebles [dimensionless]
      dXedx = (C_r / H) * (beta * (1 - X_e) - n_b * alpha2 * X_e
         ↪ **2 * hbar**2 / c)
    end subroutine Peebles
```

- Having values X_e in hand, I proceed directly to calculation of n_e *on a given grid* using (2.15). As the value of n_e spans many orders of magnitude, it was safer to use $\ln n_e$ instead

```
n_e = log(Omega_b * rho_c / (m_H * exp(3.d0 * x_rec)) * X_e)
```

- The next step is to "reconstruct" the function $n_e(x)$ to be able to evaluate it at arbitrary value of x. For this, I spline it with the routine from "spline_1D_mod.f90" module to get the value of its second derivative

```
yp1 = 1.d30
ypn = 1.d30
call spline(x_rec, n_e, yp1, ypn, n_e2)
```

which allows the "reconstruction" as

```
function get_n_e(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: get_n_e

  ! n_e = log(electron density)
  ! n_e2 = its double derivative
  get_n_e = exp(splint(x_rec, n_e, n_e2, x))
  !print *, "The splint of n_e is", get_n_e
  ! It spans many orders of magnitude
end function get_n_e
```

where exponentiate the final expression to get rid of logarithm and restore the original n_e.

- To calculate $\tau(x)$ together with its derivatives, I follow similar procedure as described above. First, I integrate the analytic expression (1.2), which in code reads

```
subroutine dtau_int(x, y, dydx)
  use healpix_types
  implicit none
  real(dp),                intent(in)  :: x
  real(dp), dimension(:), intent(in)  :: y
  real(dp), dimension(:), intent(out) :: dydx

  dydx = -c * get_n_e(x) * sigma_T * exp(x) / get_H_p(x)
end subroutine dtau_int
```

using "odeint" routine from "spline_1D_mod.f90"

```
tau(n) = 0.d0
tau_i(1) = tau(n)
! We should set h1 again because it could be changed during
   ↪ previous integration to optimize runtime
h1 = 1.d-6
```

```
h_min = 0.d0
eps = 1.d-9
do i = n - 1, 1, -1
   call odeint(tau_i, x_rec(i+1), x_rec(i), eps, h1, h_min,
      ↪ dtau_int, bsstep, output1)
   tau(i) = tau_i(1)
end do
```

to get the values of $\tau(x)$ *on a given grid*. Then, I calculate its second derivative through spline from "spline_1D_mod.f90"

```
call spline(x_rec, tau, dtau_function(x_rec(1)), dtau_function
   ↪ (x_rec(n)), tau2)
```

where

```
function dtau_function(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: dtau_function

  dtau_function = -c * get_n_e(x) * sigma_T * exp(x) / get_H_p
     ↪ (x)
end function dtau_function
```

The last part is to actually "reconstruct" $\tau(x)$ by using

```
function get_tau(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: get_tau

  ! In this way we "recover" tau(x)
  !get_tau = splint(x_rec, tau, tau2, x)
  get_tau = splint(x_rec, tau, tau2, x)
  !get_tau = 1.d0
end function get_tau
```

In a similar way, the derivatives are computed. The only difference, is that I am using "splint$_{\mathrm{d}}$eriv" from "spline_1D_mod.f90" module to calculate $\tau'(x)$ as

```
function get_dtau(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: get_dtau

  ! "recovering" thw first derivative of tau
  get_dtau = splint_deriv(x_rec, tau, tau2, x)
end function get_dtau
```

- Finally, having everything else in place, I proceed to the last part - the calculation of visibility function. I am using (1.5) to calculate $\tilde{g}(x)$ *on a given grid*

```
do i = 1, n
   g(i) = -get_dtau(x_rec(i)) * exp(-get_tau(x_rec(i)))
end do
```

  and then repeat the same I did to calculate optical depth.

- The last part is to save results in separate ".dat" files and plot them. For this purpose, I am using Jupyter notebook and open source graphic library plot.ly (code can be found in the of this report).

4 Results

Below I present main results of this milestone. Plot 4.1 depicts the electron fraction as a function of redshift. To switch from x to redshift z, I've used their respective definitions (see [2])

$$x = \ln a, \quad z = (a_0 - a)/a. \tag{4.1}$$

where a_0 is the value of the scale factor at present time.

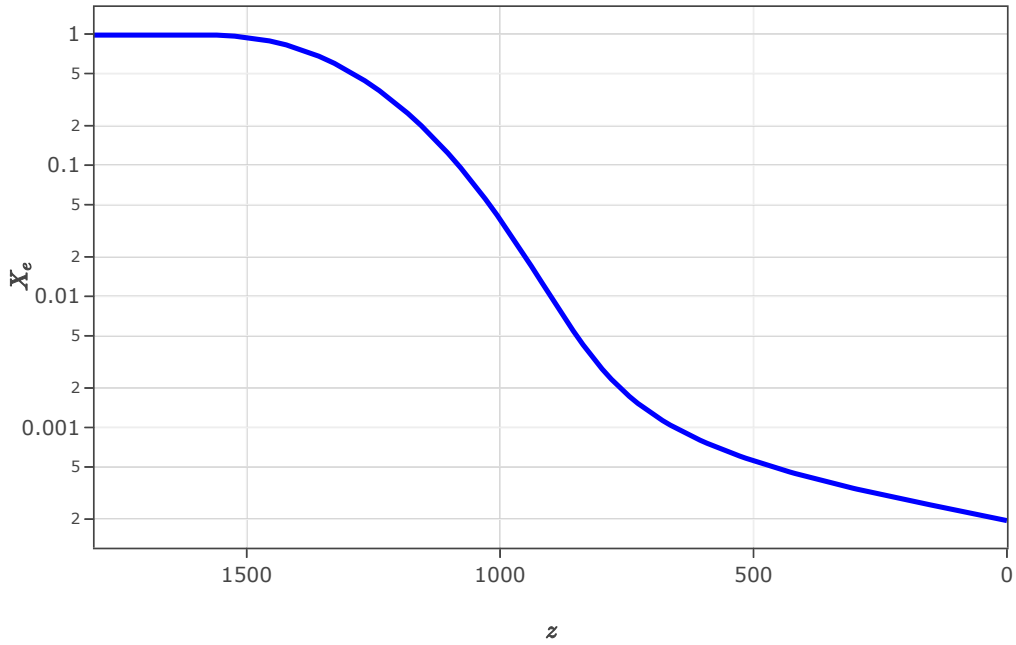

Figure 4.1: The free electron fraction X_e as a function of redshift, using the Saha (2.3) and Peebles (2.7) approximations. The switch between the two comes at $X_e = 0.99$, which roughly corresponds to redshift $z = 1559$.

To make evaluation possible, I set the cosmological parameters to be (Ref. [2])

$$
\begin{aligned}
\text{Hobble constant}: \quad & h && = 0.7, \\
\text{Hobble rate}: \quad & H_0 && = h \cdot 100\,\mathrm{km\,s^{-1}\,Mpc^{-1}}, \\
\text{CMB temperature}: \quad & T_0 && = 2.725\,\mathrm{K}, \\
\Rightarrow \quad & \Omega_r && = 5.04 \cdot 10^{-5}, \\
& \Omega_\nu && = 8.3 \cdot 10^{-5} - \Omega_r \\
& && = 3.26 \cdot 10^{-5}, \\
\text{Baryon density}: \quad & \Omega_b && = 0.046, \\
\text{CDM density}: \quad & \Omega_m && = 0.224, \\
\text{Vacuum density}: \quad & \Omega_\Lambda && = 1 - (\Omega_m + \Omega_b + \Omega_r + \Omega_\nu) \\
& && = 0.72992.
\end{aligned}
$$

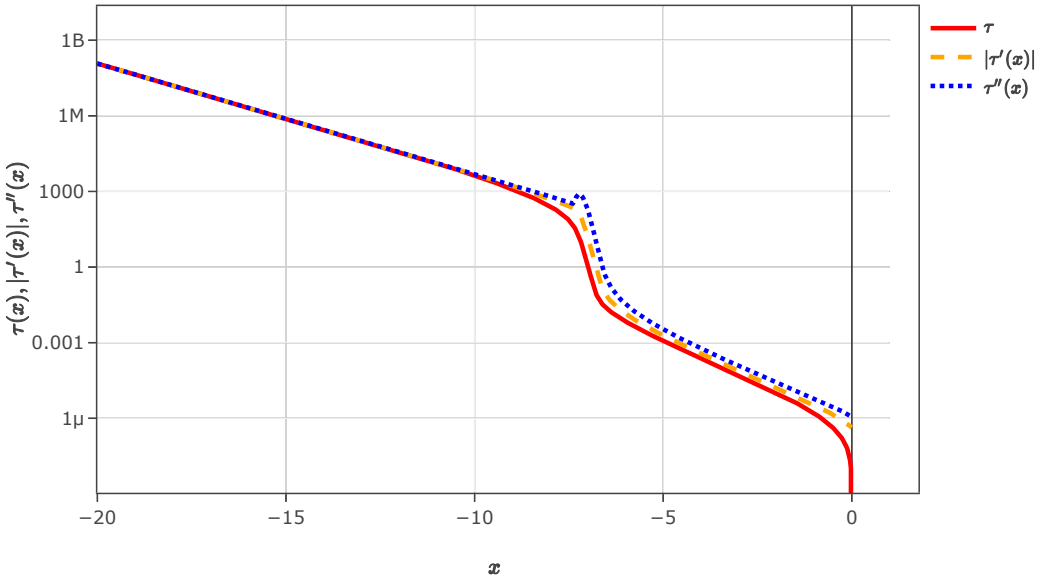

Figure 4.2: The optical depth $\tau(x)$ (solid red line) of the universe as seen from present day, together with its first, $|\tau'(x)|$, (dashed orange line) and second, $\tau''(x)$, (dotted blue line) derivatives.

which, together with other constants, are stored in "params.f90" file.

The precomputed values of $\tau(x)$ and its derivatives are shown in the figure 4.2, whereas figure 4.3 shows the precomputed $\tilde{g}(x)$ and its derivatives scaled to fit into the same graph.

The peak of $\tilde{g}(x)$ is at $x = -6.9838$ and corresponds to a redshift $z \sim 1100$ (i.e. the last scattering surface discussed in Introduction).

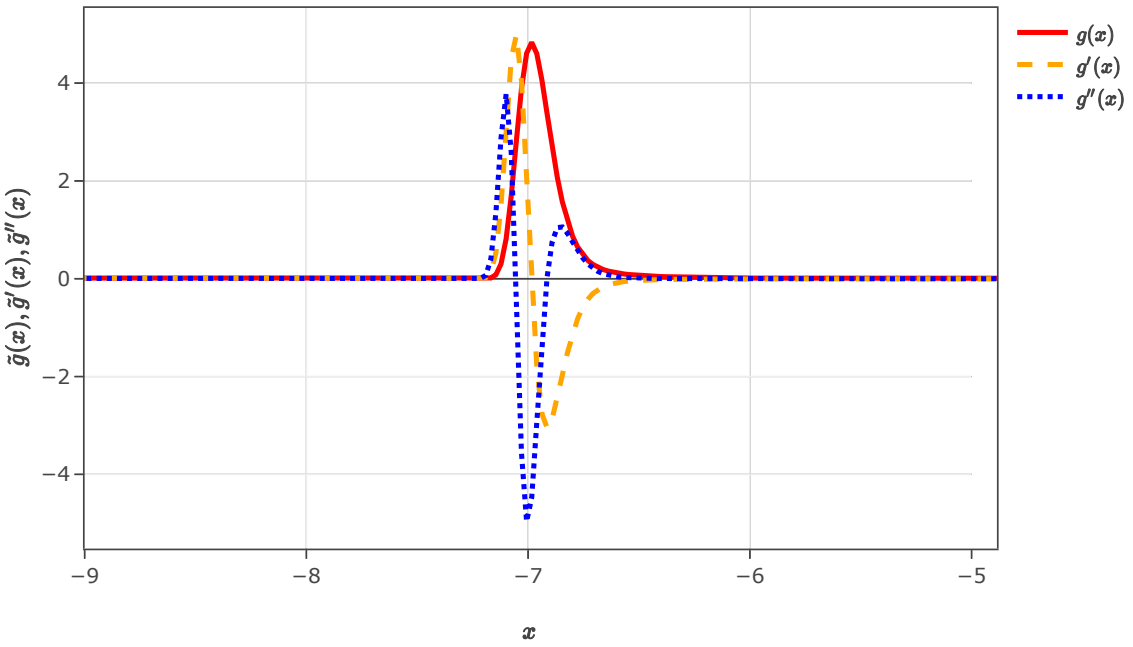

Figure 4.3: The visibility function $\tilde{g}$ (solid red line) of the universe, its first derivative $\tilde{g}'/10$ (dashed orange line) and second derivative $\tilde{g}''/200$ (dotted blue line) as functions of x.

REFERENCES

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

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

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

[4] H. K. Eriksen, *Milestone 2: The recombination history of the universe* (2014b).

CODE

Only the most relevant parts of the code present, while others can be found in

```
~hke/AST5220/v14/src/cmbspec
```

Listing 1: rec_mod.f90

```
module rec_mod
  use healpix_types
  use params
  use time_mod
  use ode_solver
  use spline_1D_mod
  implicit none

  integer(i4b),                          private :: n                 ! Number of grid
      ↪ points
  real(dp), allocatable, dimension(:), private :: x_rec               ! Grid
  real(dp), allocatable, dimension(:), private :: tau, tau2, tau22    ! Splined tau and
      ↪ second derivatives
  real(dp), allocatable, dimension(:), private :: n_e, n_e2           ! Splined (log of)
      ↪ electron density, n_e
  real(dp), allocatable, dimension(:), private :: g, g2, g22          ! Splined
      ↪ visibility function

  ! Variables I've defined
  real(dp), allocatable, dimension(:), private :: a_rec ! scale factor

contains

  subroutine initialize_rec_mod
    implicit none

    integer(i4b) :: i, j, k
    real(dp)     :: saha_limit, y, T_b, n_b, dydx, xmin, xmax, dx, f, n_e0, X_e0,
        ↪ xstart, xstop
    logical(lgt) :: use_saha
    real(dp), allocatable, dimension(:) :: X_e ! Fractional electron density, n_e / n_H

    ! Variables I've defined
    real(dp) :: saha_const               ! The constant factor in Saha equation
                                                                  ! which is an array
                                                                      ↪  of constants
                                                                      ↪  (due
                                                                  ! to different
                                                                      ↪ values of a)
    real(dp) :: lambda_2s1s, H, h1, h_min, eps, Xe_i(1), tau_i(1), yp1, ypn
    lambda_2s1s = 8.227d0 ! s**-1

    ! Defined variables
    saha_limit = 0.99d0       ! Switch from Saha to Peebles when X_e < 0.99
    xstart     = log(1.d-10)  ! Start grids at a = 10^-10
    xstop      = 0.d0         ! Stop  grids at a = 1
    n          = 1000         ! Number of grid points between xstart and xstopo

    !write(*,*) pi

    ! Scale factor
    allocate(a_rec(n))
    ! The grid
    allocate(x_rec(n))
    ! Fractional electron density
    allocate(X_e(n))
    ! Optical depth and its derivatives
```

```
    allocate(tau(n))
    allocate(tau2(n))
    allocate(tau22(n))
    ! Electron number density
    allocate(n_e(n))
    allocate(n_e2(n))
    ! Visibility function and its derivatives
    allocate(g(n))
    allocate(g2(n))
    allocate(g22(n))

    !=========================================!
    !             Filling the Grid            !
    !=========================================!
    ! Task: Fill in x (rec) grid
    dx = (xstop - xstart) / (n - 1)
    x_rec(1) = xstart
    do i = 2, n
       x_rec(i) = x_rec(1) + dx * (i - 1)
    end do
    a_rec = exp(x_rec)

    !=========================================!
    !               Computing Xe              !
    !=========================================!
    ! Task: Compute X_e and n_e at all grid times
    use_saha = .true.
    ! for calculating Peebles eq. through ode solver
    h1 = 1.d-6
    h_min = 0.d0
    eps = 1.d-9
    do i = 1, n
       ! Baryon temperature
       T_b = T_0 / exp(x_rec(i))
       ! Baryon number density
       n_b = (Omega_b * rho_c) / (m_H * exp(3.d0 * x_rec(i)))
       !print *, n_b
       if (use_saha) then
          ! Calculate constant in Saha equation
          saha_const = (m_e * k_b * T_b / (2.d0 * pi))**1.5d0 * exp(-epsilon_0 / (k_b *
              ↪  T_b)) / (n_b * hbar**3.d0)
          ! Use Saha equation
          X_e(i) = (-saha_const + sqrt(saha_const**2.d0 + 4.d0 * saha_const)) / 2.d0
          if (X_e(i) < saha_limit) then
             use_saha = .false.
             Xe_i(1) = X_e(i)
          end if
       else
          ! Use the Peebles equation
          call odeint(Xe_i, x_rec(i-1), x_rec(i), eps, h1, h_min, Peebles, bsstep,
              ↪ output1)
          X_e(i) = Xe_i(1)
          !print *, "Peebles gives", X_e(i)
       end if
    end do

    !print *, X_e
    ! Saving computed X_e(z) into a file
    open(1, file = "X_e_z.dat", action = "write", status = "replace")
    do i = 1, n
       write(1,*) (1 - a_rec(i)) / a_rec(i), X_e(i)
    end do
    close(1)
    !print *, "X_e_z.dat has been saved"

    !print *, X_e
```

```
!=========================================!
!         Electron Number Density         !
!=========================================!

! Task: Compute splined (log of) electron density function
! we use log because it spans many orders of magnitude, later on
! then we will use exp to recover the original value
n_e = log(Omega_b * rho_c / (m_H * exp(3.d0 * x_rec)) * X_e)
!print *, "n_e is ", n_e
! Spline to get the double derivative of n_e
yp1 = 1.d30
ypn = 1.d30
call spline(x_rec, n_e, yp1, ypn, n_e2)
!print *, "splined n_e is ", n_e2

!=========================================!
!              Optical Depth              !
!=========================================!
! Task: Compute optical depth at all grid points
tau(n) = 0.d0
tau_i(1) = tau(n)
! We should set h1 again because it could be changed during previous integration to
    ↪  optimize runtime
h1 = 1.d-6
h_min = 0.d0
eps = 1.d-9
do i = n - 1, 1, -1
   call odeint(tau_i, x_rec(i+1), x_rec(i), eps, h1, h_min, dtau_int, bsstep,
       ↪ output1)
   tau(i) = tau_i(1)
end do
!print *, "tau is ", tau


! Task: Compute splined (log of) optical depth
! Compute yhe second derivative of tau (tau2) on a grid
! using an analytic expression of tau' stored in dtau_function
call spline(x_rec, tau, dtau_function(x_rec(1)), dtau_function(x_rec(n)), tau2)
!print *, "Second derivative of tau is ", tau2
! Checking the values of "recovered tau"
!do i = 1, n
!   print *, "The recovered tau is ", get_tau(x_rec(i))
!end do

! Checking the value of "recovered" derivative of tau
!do i = 1, n
!   print *, "The recovered dtau is ", get_dtau(x_rec(i))
!end do

! Task: Compute splined second derivative of (log of) optical depth
!print *, tau2
!yp1 = 1.d30
!ypn = 1.d30
call spline(x_rec, tau2, yp1, ypn, tau22)
!print *, tau22

! Checking the value of "recovered" derivative of tau
! do i = 1, n
!    print *, "The recovered ddtau is ", tau22
! end do

! Checking the value of "recovered" derivative of tau
!do i = 1, n
!   print *, "The recovered ddtau is ", get_ddtau(x_rec(i))
!end do
```

```
!=========================================!
! Saving tau and its derivatives to a file !
!=========================================!

open(2, file = "tau_x.dat", action = "write", status = "replace")
do i = 1, n
   write(2,*) x_rec(i),"␣", get_tau(x_rec(i))
end do
close(2)
!print *, "tau_x.dat has been saved"

open(3, file = "dtau_x.dat", action = "write", status = "replace")
do i = 1, n
   write(3,*) x_rec(i),"␣", get_dtau(x_rec(i))
end do
close(3)
!print *, "dtau_x.dat has been saved"

open(4, file = "ddtau_x.dat", action = "write", status = "replace")
do i = 1, n
   write(4,*) x_rec(i),"␣", get_ddtau(x_rec(i))
end do
close(4)
!print *, "ddtau_x.dat has been saved"

!=========================================!
!           Visibility Function           !
!=========================================!

! Task: Compute splined visibility function
! Visibility function as defined in Callin
!open(5, file = "g_x.dat", action = "write", status = "replace")
do i = 1, n
   g(i) = -get_dtau(x_rec(i)) * exp(-get_tau(x_rec(i)))
end do
!close(5)
!print *, "g_x.dat has been saved"

! Getting second derivative through spline
call spline(x_rec, g, yp1, ypn, g2)

! Task: Compute splined second derivative of visibility function
call spline(x_rec, g2, yp1, ypn, g22)

!=========================================!
!  Saving g and its derivatives to a file  !
!=========================================!

open(5, file = "g_x.dat", action = "write", status = "replace")
do i = 1, n
   write(5,*) x_rec(i),"␣", get_g(x_rec(i))
end do
close(5)
!print *, "g_x.dat has been saved"

open(6, file = "dg_x.dat", action = "write", status = "replace")
do i = 1, n
   write(6,*) x_rec(i),"␣", get_dg(x_rec(i))
end do
close(6)
!print *, "dtau_x.dat has been saved"

open(7, file = "ddg_x.dat", action = "write", status = "replace")
do i = 1, n
   write(7,*) x_rec(i),"␣", get_ddg(x_rec(i))
```

```
    end do
    close(7)
    !print *, "ddg_x.dat has been saved"

  end subroutine initialize_rec_mod

  ! Create subroutine to calculate deta/dx
    subroutine Peebles(x, X_e, dXedx)
      use healpix_types
      implicit none
      real(dp),                intent(in)  :: x
      real(dp), dimension(:), intent(in)  :: X_e
      real(dp), dimension(:), intent(out) :: dXedx
      ! Other variables
      integer(i4b)                         :: i
      ! Preliminary variables
      real(dp)                             :: H, T_b, dummy_const, phi2, n_b
      ! Calculatable variables
      real(dp)                             :: alpha2, lambda_2s1s
      real(dp)                             :: beta, beta2
      real(dp), dimension(:), allocatable :: lambda_alpha, n_1s, C_r
      ! lambda_2s->1s [s^-1]
      lambda_2s1s = 8.227d0
      ! Taking the value of Hubble constant from time_mod (milestone 1) [s^-1]
      H = get_H(x)
      ! Baryon temperature [K]
      T_b = T_0 / exp(x)
      ! Baryon number density [m^-3]
      n_b = Omega_b * rho_c / (m_H * exp(3 * x))
      ! e/kT constant [dimensionless]
      dummy_const = epsilon_0 / (k_b * T_b)
      ! phi2 [dimensionless]
      phi2 = 0.448d0 * log(dummy_const)
      ! alpha2 [kg^-2]
      alpha2 = (64 * pi / (sqrt(27 * pi))) * (alpha / m_e)**2 * sqrt(dummy_const) *
          ↪ phi2
      ! beta [s^-1]
      beta = (alpha2 / (hbar * c)) * (m_e * k_b * T_b / (2 * pi))**1.5d0 * exp(-
          ↪ dummy_const)
      ! beta2 [s^-1]
      beta2 = (alpha2 / (hbar * c)) * (m_e * k_b * T_b / (2 * pi))**1.5 * exp(-
          ↪ dummy_const / 4.d0)
      ! n1s [m^-3]
      n_1s = (1 - X_e) * n_b
      ! Calculating Lambda_alpha [s^-1]
      lambda_alpha = H * (3 * epsilon_0)**3 / (n_1s * (8 * pi)**2 * (hbar * c)**3)
      ! C_r [dimensionless]
      C_r = (lambda_2s1s + lambda_alpha) / (lambda_2s1s + lambda_alpha + beta2)
      ! dX_e/dx by Peebles [dimensionless]
      dXedx = (C_r / H) * (beta * (1 - X_e) - n_b * alpha2 * X_e**2 * hbar**2 / c)
    end subroutine Peebles

    ! To integrate tau with odeint we need a subroutine
    subroutine dtau_int(x, y, dydx)
      use healpix_types
      implicit none
      real(dp),                intent(in)  :: x
      real(dp), dimension(:), intent(in)  :: y
      real(dp), dimension(:), intent(out) :: dydx

      dydx = -c * get_n_e(x) * sigma_T * exp(x) / get_H_p(x)
    end subroutine dtau_int

    ! Task: Complete routine for computing n_e at arbitrary x, using precomputed
        ↪ information
    ! Hint: Remember to exponentiate...
```

```
function get_n_e(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: get_n_e

  ! n_e = log(electron density)
  ! n_e2 = its double derivative
  get_n_e = exp(splint(x_rec, n_e, n_e2, x))
  !print *, "The splint of n_e is", get_n_e
  ! It spans many orders of magnitude
end function get_n_e

! Task: Complete routine for computing tau at arbitrary x, using precomputed
    ↪ information
function get_tau(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: get_tau

  ! In this way we "recover" tau(x)
  !get_tau = splint(x_rec, tau, tau2, x)
  get_tau = splint(x_rec, tau, tau2, x)
  !get_tau = 1.d0
  !print *, "Recovered tau is ", get_tau
end function get_tau

! The "analytic" (from text book) expression for tau
function dtau_function(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: dtau_function

  dtau_function = -c * get_n_e(x) * sigma_T * exp(x) / get_H_p(x)
end function dtau_function

! Task: Complete routine for computing the derivative of tau at arbitrary x, using
    ↪ precomputed information
function get_dtau(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: get_dtau

  ! "recovering" thw first derivative of tau
  get_dtau = splint_deriv(x_rec, tau, tau2, x)
end function get_dtau

! Task: Complete routine for computing the second derivative of tau at arbitrary x,
! using precomputed information
function get_ddtau(x)
  implicit none

  real(dp), intent(in) :: x
  real(dp)             :: get_ddtau

  get_ddtau = splint(x_rec, tau2, tau22, x)
end function get_ddtau

! Task: Complete routine for computing the visibility function, g, at arbitray x
function get_g(x)
  implicit none

  real(dp), intent(in) :: x
```

```
    real(dp)              :: get_g

    get_g = splint(x_rec, g, g2, x)
  end function get_g

  ! Task: Complete routine for computing the derivative of the visibility function, g
    ↪ , at arbitray x
  function get_dg(x)
    implicit none

    real(dp), intent(in) :: x
    real(dp)              :: get_dg

    get_dg = splint_deriv(x_rec, g, g2, x)
  end function get_dg

  ! Task: Complete routine for computing the second derivative of the visibility
    ↪ function, g, at arbitray x
  function get_ddg(x)
    implicit none

    real(dp), intent(in) :: x
    real(dp)              :: get_ddg

    get_ddg = splint(x_rec, g2, g22, x)
  end function get_ddg

end module rec_mod
```

Listing 2: spline_1D_mod.f90

```
module spline_1D_mod
  use healpix_types
  implicit none

  interface locate
     module procedure locate_int, locate_dp
  end interface

contains

  ! Routines from Numerical Recipes
  subroutine spline(x, y, yp1, ypn, y2)
    implicit none

    real(dp),                intent(in)  :: yp1, ypn
    real(dp), dimension(:), intent(in)  :: x, y
    real(dp), dimension(:), intent(out) :: y2

    integer(i4b) :: i, n, m
    real(dp), dimension(:), allocatable :: a, b, c, r

    n = size(x)
    allocate(a(n),b(n),c(n),r(n))

    c(1:n-1) = x(2:n)-x(1:n-1)
    r(1:n-1) = 6.d0*((y(2:n)-y(1:n-1))/c(1:n-1))
    r(2:n-1) = r(2:n-1)-r(1:n-2)
    a(2:n-1) = c(1:n-2)
    b(2:n-1) = 2.d0*(c(2:n-1)+a(2:n-1))
    b(1)     = 1.d0
    b(n)     = 1.d0

    if (yp1 > 0.99d30) then
       r(1) = 0.d0
       c(1) = 0.d0
    else
       r(1) = (3.d0 / (x(2)-x(1))) * ((y(2)-y(1))/(x(2)-x(1)) - yp1)
       c(1) = 0.5d0
    end if

    if (ypn > 0.99d30) then
       r(n) = 0.d0
       a(n) = 0.d0
    else
       r(n) = (-3.d0 / (x(n)-x(n-1))) * ((y(n)-y(n-1))/(x(n)-x(n-1)) - ypn)
       a(n) = 0.5d0
    end if

    call tridag(a(2:n), b(1:n), c(1:n-1), r(1:n), y2(1:n))
    deallocate(a,b,c,r)
  end subroutine spline

  function splint(xa, ya, y2a, x)
    implicit none

    real(dp),                intent(in)  :: x
    real(dp), dimension(:), intent(in)  :: xa, ya, y2a
    real(dp)                             :: splint

    integer(i4b) :: khi, klo, n
    real(dp)     :: a, b, h

    n = size(xa)

    klo = max(min(locate(xa,x),n-1),1)
```

```
    khi = klo+1
    h   = xa(khi) - xa(klo)
    a   = (xa(khi) - x) / h
    b   = (x - xa(klo)) / h

    splint = a*ya(klo) + b*ya(khi) + ((a**3-a)*y2a(klo) + (b**3-b)*y2a(khi))*(h**2)/6.
        ↪ d0

  end function splint


  function splint_deriv(xa, ya, y2a, x)
    implicit none

    real(dp),               intent(in)  :: x
    real(dp), dimension(:), intent(in)  :: xa, ya, y2a
    real(dp)                            :: splint_deriv

    integer(i4b) :: khi, klo, n
    real(dp)     :: a, b, h

    n = size(xa)

    klo = max(min(locate(xa,x),n-1),1)
    khi = klo+1
    h   = xa(khi) - xa(klo)
    a   = (xa(khi) - x) / h
    b   = (x - xa(klo)) / h

    splint_deriv = (ya(khi) - ya(klo)) / h - (3.d0 * a**2 - 1.d0) / 6.d0 * h * y2a(klo)
        ↪  + &
         & (3.d0 * b**2 - 1.d0) / 6.d0 * h * y2a(khi)

  end function splint_deriv

end module spline_1D_mod
```

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

Listing 3: Milestone2_DataPlots.py

```
import numpy as np
# library for plotting
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


plt.init_notebook_mode(connected = True)

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

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



# Milestone 1
# Omega(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, 'Omega_x.pdf')
pio.write_image(etaFigure,   'eta_x.pdf')
pio.write_image(HxFigure,    'H_x.pdf')
pio.write_image(HzFigure,    'H_z.pdf')


# Milestone 2
XeDataFile       = np.loadtxt("X_e_z.dat")

tauDataFile      = np.loadtxt("tau_x.dat")
dtauDataFile     = np.loadtxt("dtau_x.dat")
ddtauDataFile    = np.loadtxt("ddtau_x.dat")

gtildeDataFile   = np.loadtxt("g_x.dat")
dgtildeDataFile  = np.loadtxt("dg_x.dat")
ddgtildeDataFile = np.loadtxt("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, 'X_e_z.pdf')
pio.write_image(tauFigure, 'tau_x.pdf')
pio.write_image(gtildeFigure, 'g_x.pdf')
```