AST5220 - Cosmology 2
Milestone 4

Alex Ho

May 23, 2017

The program, plots and the report can be found in the following Github page:

`https://github.com/AHo94/AST5220_Projects/tree/master/Project4`

Mathematics

We would first like to compute the *source function* $\tilde{S}$, given as

$$\tilde{S}(k,x) = \tilde{g}\left[\Theta_0 + \Psi + \frac{1}{4}\Pi\right] + e^{-\tau}[\Psi' - \Phi'] - \frac{1}{k}\frac{d}{dx}(\mathcal{H}\tilde{g}v_b) + \frac{3}{4k^2}\frac{d}{dx}\left[\mathcal{H}\frac{d}{dx}(\mathcal{H}\tilde{g}\Pi)\right] \tag{1}$$

Without polarization, $\Pi = \Theta_2$. One thing to note is the dimensionality of the third and final term. Let us first consider the third term. The variable k has the physical dimension $1/m$, whereas the dimension of $\mathcal{H}$ is $1/s$. All the other parameters, i.e $\tilde{g}$ and v_b are dimensionless. The physical dimension of the third term is then m/s. We want the source function to dimensionless, so we have to divide with a factor of c (i.e, light speed) to make it dimensionless. Similarly, the final term has the dimension m^2/s^2, so we simply divide by c^2.

The final expression for the source function is then

$$\tilde{S}(k,x) = \tilde{g}\left[\Theta_0 + \Psi + \frac{1}{4}\Pi\right] + e^{-\tau}[\Psi' - \Phi'] - \frac{1}{ck}\frac{d}{dx}(\mathcal{H}\tilde{g}v_b) + \frac{3}{4(ck)^2}\frac{d}{dx}\left[\mathcal{H}\frac{d}{dx}(\mathcal{H}\tilde{g}\Pi)\right] \tag{2}$$

We know what the first and second term tells us, but the third term is a Doppler term, which says that the photons either gain ore lose energy due to their peculiar velocities. The fourth term is a correction term, where the correction comes from the polarization. The transfer function can then be used to compute the transfer function $\Theta_l(k, x = 0)$ which is given as

$$\Theta_l(k, x = 0) = \int_{-\infty}^{0} \tilde{S}(k,x) j_l[k(\eta_0 - \eta(x))]dx \tag{3}$$

where $j_l(x)$ is the spherical Bessel function. We see that the transfer function require an integration in the range $x \in [-\infty, 0]$. However, our x-grid is a finite grid, which limits our integration down to our x-grid. Now that we have the transfer function, we can use it to compute the actual power spectrum C_l as

$$C_l = \int_0^\infty \left(\frac{ck}{H_0}\right)^{n-1} \Theta_l^2(k)\frac{dk}{k} \tag{4}$$

where we choose $n = 0.96$, which corresponds roughly to the best fit WMAP value. Like the transfer function, our k-grid is a finite grid, so the integration will only be over our given k values.

Numerics

A small change has been made to the computation of the Boltzmann-Einstein equations, but it has a significant impact on the computational time. The program now, for one value of k, creates a (cubic) spline, with `Scipy`'s function `interpolate.CubicSpline`, of the optical depth τ and conformal time η. When computing the Boltzmann equations, it will simply just call the spline and compute the value for a given point x. Before this change, the program would create a spline and then compute the spline for every iteration in the ODE solver. The program now runs roughly 4 times faster with

the new interpolation method. Another thing to note, the interpolation now *actually* uses Cubic interpolation. Using `interpolate.CubicSpline` also gives more stable results compared to the previous interpolation method. One example is the interpolated values of $|\tau''|$ being a lot more stable.[1]

`time_mod` now also saves the computed derivatives, of the relevant variables, as well. The class `Plotter` saves all relevant data to a text file, based on the k value. We can then simply read off the text files to obtain the desired values from the Boltzmann equations, which would be a lot faster than computing everything from scratch.

A new class `Power_Spectrum` has been made specifically used to compute power spectrum C_l. It will first read off the saved values of the different variables and the relevant derivatives. It will then compute the source function $\tilde{S}(x, k)$ and then interpolate the computed values of $\tilde{S}$ over a larger x and k grid, both with 5000 points, which results to a 5000×5000 matrix.

`Scipy` has its own Bessel function `special.spherical_jn`, which takes in two arguments. The first argument is the order of the Bessel function, i.e the l values. The second argument is the actual function argument, i.e the x values. One thing to note here is that computing the Bessel functions over the values $k(\eta_0 - \eta(x))$, takes roughly 30 seconds for one value of k over the x interval. Computing this for 5000 k values will take days. A different approach is then to create a spline over the Bessel functions.

To create a spline over the Bessel function $j_l(x)$, we first compute it over the values $x \in [0, 5400]$ for all the l values we are interested in. In this case, the l values we use are the same as the ones as Callin. That is, we use $l = 2, 3, 4, 6, 8, 10, 12, 15$ and 20 and the every 10th l up to $l = 100$, every 20th to $l = 200$, every 25th up to $l = 300$ and every 50th up to $l = l_{\max} = 1200$. This means we have 44 l points in our computation.

We will then use this to create 44 1D splines. Each spline corresponds to one value of l. We can then use each spline to compute the source function $\tilde{S}$ for one value of k. That is, we interpolate the spline over the values $k(\eta_0 - \eta(x))$ for one k value. This is done for all 5000 k values. However, the computed

[1] Recall from milestone 2 that $|\tau''|$ was very unstable.

Bessel functions will not be saved in an array, because the end array would be a 44×5000×5000 array, which will use up all the memory. Instead, for each k, we interpolate the Bessel function, and then compute transfer function $\Theta_l(k)$, which integrates over the whole x grid. The output array is then a 44×5000 array.

The integration of both the transfer function $\tilde{S}$ and the power spectrum C_l will be done by using the *trapezoidal rule* [2]. `Scipy` has its own trapezoidal function `scipy.trapz`, which integrates a dataset over a (finite) grid x (or k, when we integrate the power spectrum).

Once we have integrated the transfer function, we will have to increase the number of sample points of the power spectrum C_l as well as the transfer functions themselves. This is easily done by simply creating a spline for each transfer function and the power spectrum and interpolate a new l grid with more points. We choose $l \in [0, 1200]$, with 3000 points between this interval. Finally, we normalize our Power spectrum so that the max value of $l(l+1)C_l/2\pi$ is $5775\mu K^2$. This is simply done by finding the max value over the power spectrum array, and then compute $N = 5775.0/\max(C_l)$, where N is the normalization factor and $C_l = l(l+1)C_l/2\pi$.

Best fit model

We will here try to adjust some of the cosmological parameters to get a power spectrum that best fits the one from Planck. The power spectrum from our model is found in figure 1. The first peak fits perfectly with the one from Planck, but the peaks after does not fit as well as the first one. Notable changes to the power spectrum, when we change one of the parameters while keeping the others unchanged (note: Ω_Λ changes as we change one of the Ω's), include:

- **Change of** $\Omega_m = 0.024$: Not a whole lot changed to the power spectrum when we changed Ω_m. When $\Omega_m = 0.220$ and $\Omega_m = 0.23$, the

[2] Simpson's rule did not work as well as the trapezoidal rule, due to division of zeros in the calculations. Reasons for this is unknown to me.

Ω_m	Ω_b	Ω_r	h_0
0.24	0.09	8.8×10^{-5}	0.66

Table 1: The cosmological parameters chosen for our model to best fit the Planck data

second, third and fourth peak increased a little bit, but the change was barley noticeable.

- **Change of** $\Omega_b = 0.046$: The amplitudes of the second, third and fourth peak decreased when $\Omega_b = 0.05$. On the other hand, the amplitudes of the same peaks increased when $\Omega_b = 0.042$

- **Change of** $\Omega_r = 8.3e^{-5}$: Nothing particularly interesting happened here. The second peak decreased a little bit and the third and fourth peak were shifted slightly to the left when $\Omega_r = 8.1e^{-5}$. Opposite effect happened when $\Omega_r = 8.5e^{-5}$.

- **Change of** $h_0 = 0.7$ [3]: When decrease h_0 to $h_0 = 0.66$, the amplitude of the second, third and fourth peak decreased just a tiny bit. We get the same results when we increase to $h_0 = 0.74$.

Table 1 contains the parameters that was changed in order to get the best fit power spectrum compared to the one from Planck. The plot of this can be found in figure 3. The first and second peak fits incredibly well, but the third and fourth peak are both a little off. Nevertheless, this is till a fairly good model.

[3]This is a change of the parameter h_0 in the Hubble parameter H_0

Plots

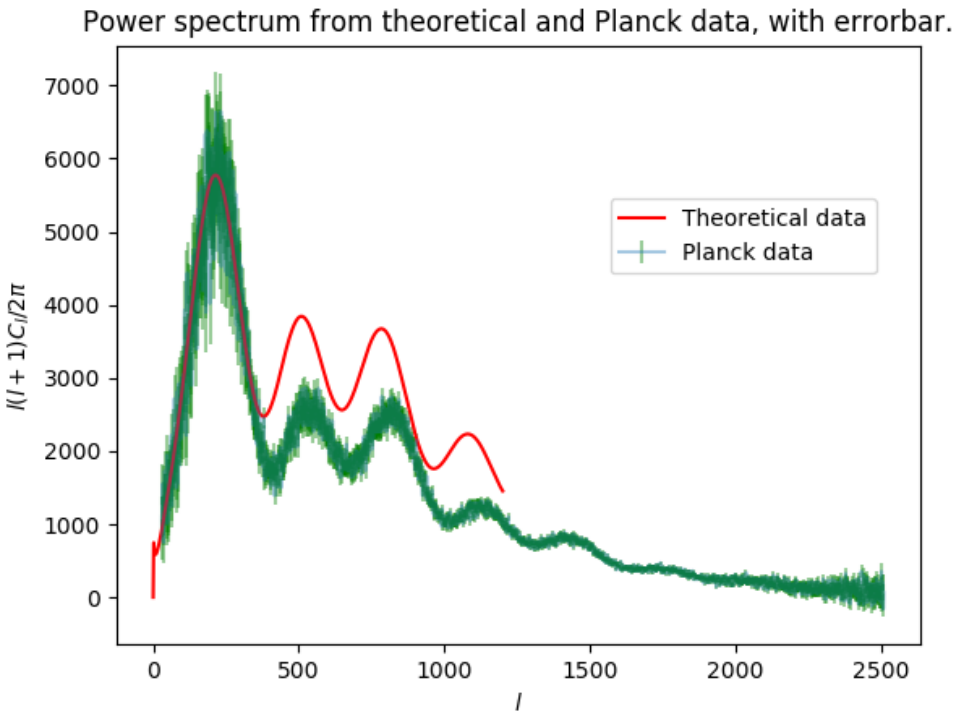

Figure 1: Plot of the power spectrum from our theoretical model and Planck's data with error bars.

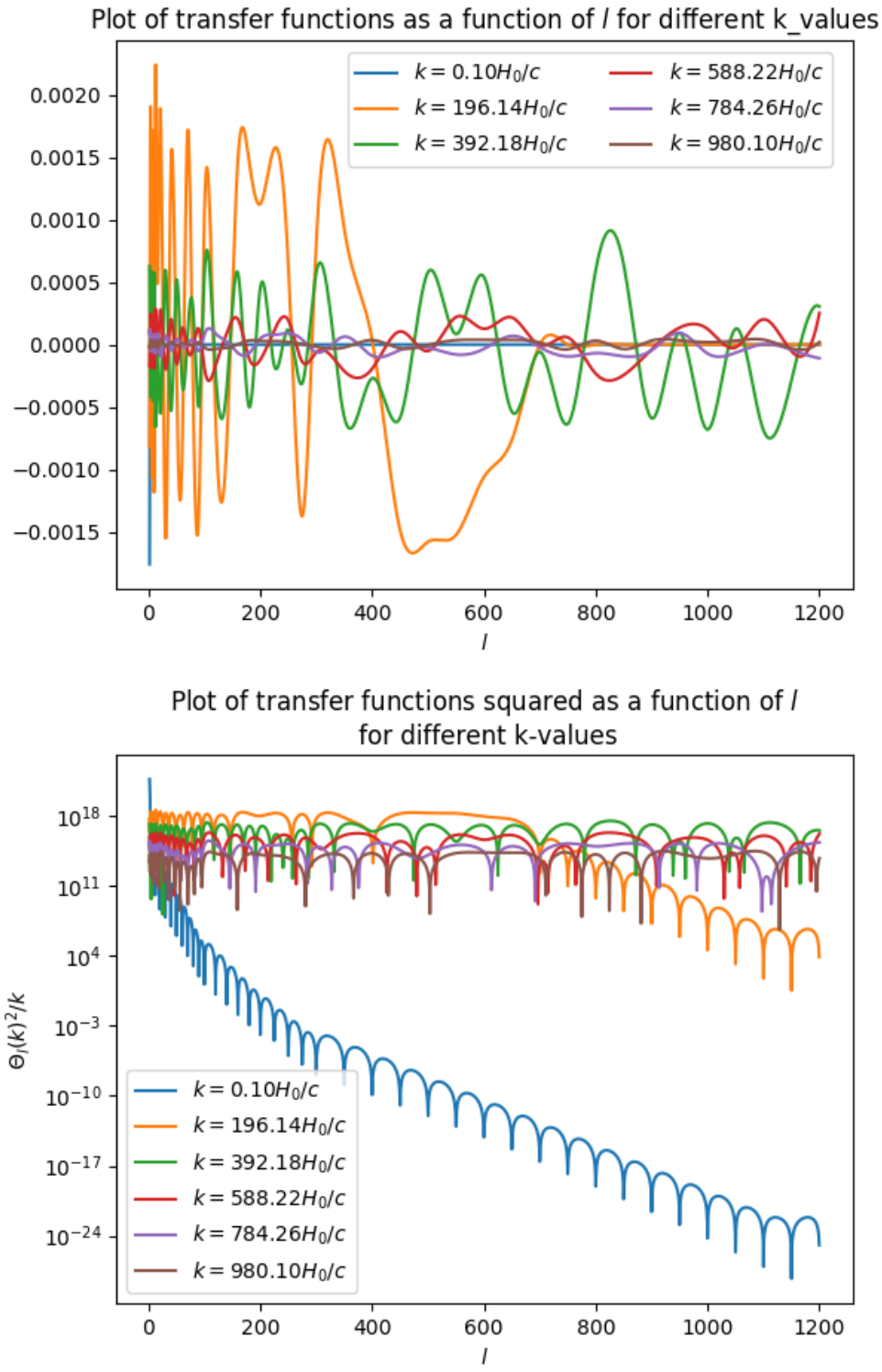

Figure 2: Plot of $\Theta_l(k)$ (top image) and $\Theta_l(k)^2$ (bottom image) as functions of l with 6 different k-values. These plots are for our theoretical model with no adjustments to the cosmological parameters. The first four points on both plots were removed due to weird behaviours.

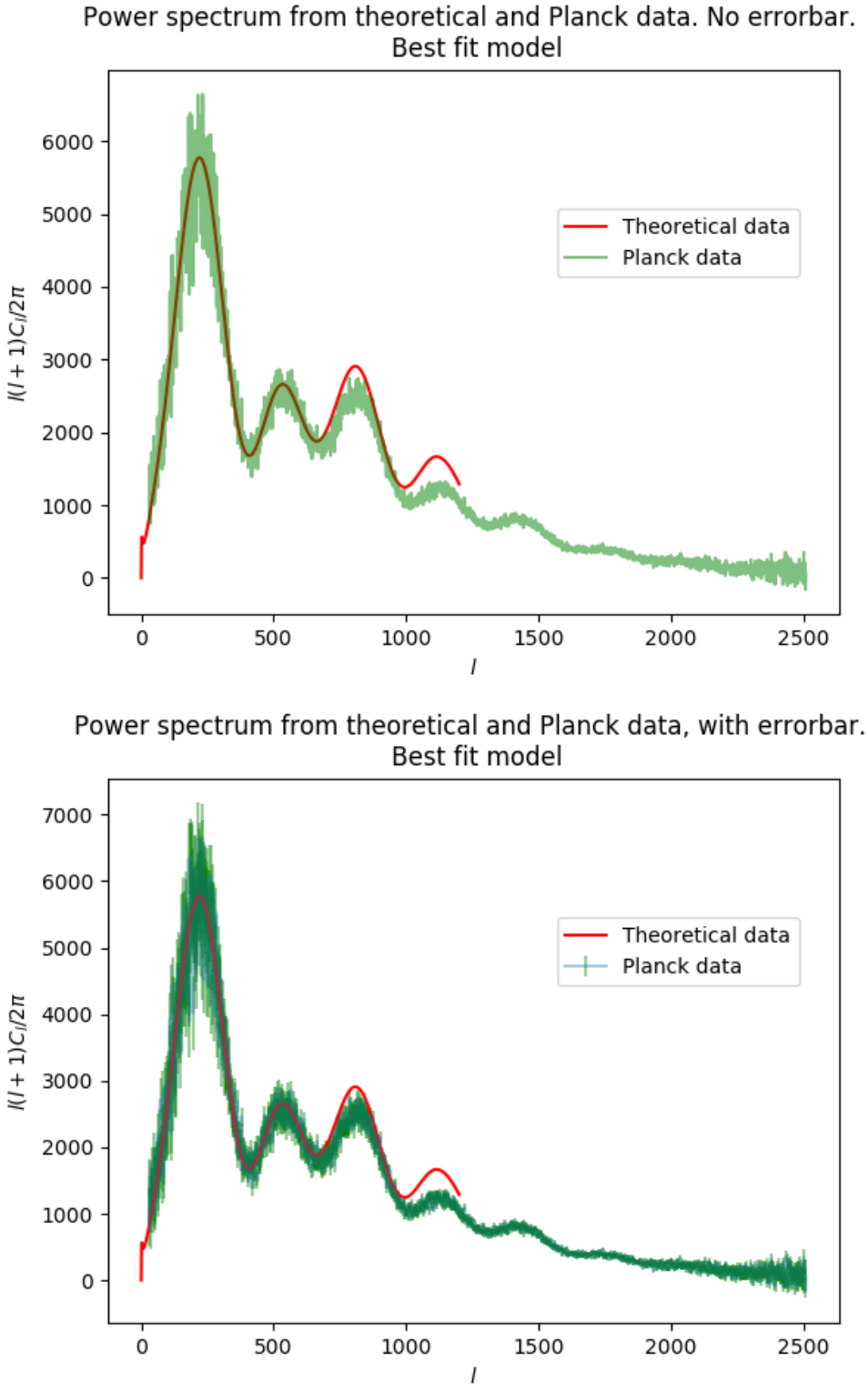

Figure 3: Power spectrum of the best fit model plotted with the one from Planck. Bottom image includes error bars. Parameters used for the best fit model is found in table 1.

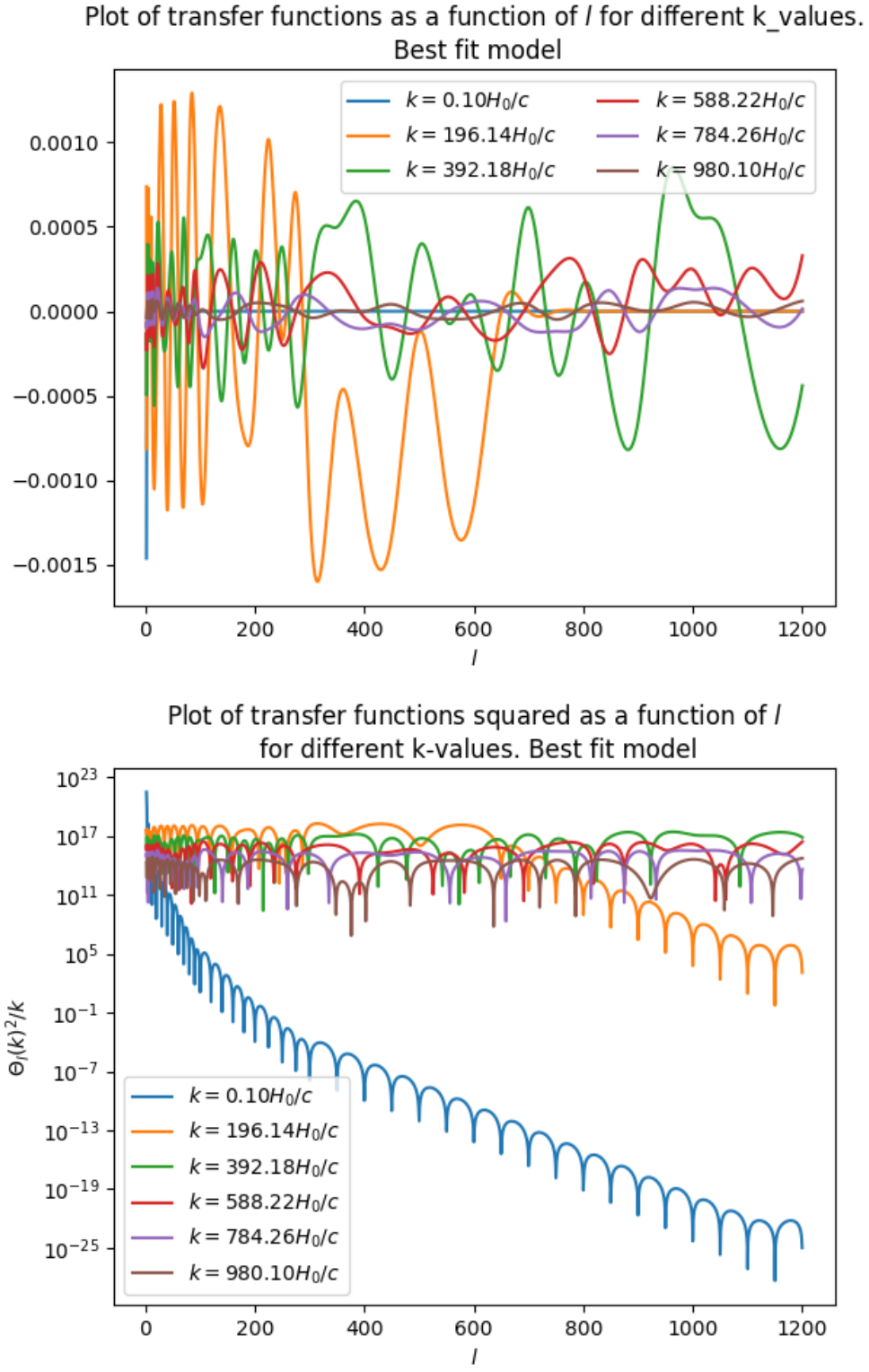

Figure 4: Plot of $\Theta_l(k)$ (top image) and $\Theta_l(k)^2$ (bottom image) as functions of l with 6 different k-values. This is the plot for the best fit model, with the adjusted cosmological parameters given in table 1. Like the previous one, the first four points on both plots were removed due to weird behaviours.

The code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
from scipy import integrate
from scipy import special
import time
import multiprocessing as mp
import sys
import os

# Global constants
# Units
eV = 1.60217647e-19
Mpc = 3.08568025e22

# Cosmological parameters
Omega_b = 0.046
Omega_m = 0.224
Omega_r = 8.3e-5
Omega_nu = 0.0
Omega_lambda = 1.0 - Omega_m - Omega_b - Omega_r -
   Omega_nu
T_0 = 2.725
n_s = 0.96
A_s = 1.0
#h0 = 0.7    # Standard parameter
h0 = 0.66    # Best fit parameter
H_0 = h0*100.0*1e3/Mpc

# General constants
c = 2.99792458e8
epsilon_0 = 13.605698*eV
m_e = 9.10938188e-31
m_H = 1.673534e-27
sigma_T = 6.652462e-29
G_grav = 6.67258e-11
```

```
rho_c0 = (3.0*H_0**2)/(8*np.pi*G_grav)
alpha = 7.29735308e-3
hbar = 1.05457148e-34
k_b = 1.3806503e-23

# Density Parameters today
rho_m0 = Omega_m*rho_c0
rho_b0 = Omega_b*rho_c0
rho_r0 = Omega_r*rho_c0
rho_lambda0 = Omega_lambda*rho_c0

# Precalculate certain factors to reduce number of
   float point operations
Saha_b_factor = ((m_e*T_0*k_b)/(2*np.pi*hbar**2))
   **(3.0/2.0)   # Factor in front of 'b' in Saha
   equation
rhoCrit_factor = 3.0/(8*np.pi*G_grav)
         # Used for critical density at arbitrary
   times

# Constant used for Peebles equation and some
   constant factors that can be precalculated
Lambda_2sto1s = 8.227
alpha_factor = ((64.0*np.pi)/(np.sqrt(27.0*np.pi)))
   *((alpha/m_e)**2.0)*(hbar**2.0/c)
beta_factor = (((m_e*T_0*k_b)/(2.0*np.pi))
   **(3.0/2.0))*(1.0/hbar**3.0)
Lambda_alpha_factor = ((3.0*epsilon_0/(hbar*c))
   **3.0)/(8*np.pi)**2.0
EpsTemp_factor = epsilon_0/(k_b*T_0)

# Other precalculated factors
H_0Squared = H_0*H_0
c_Squared = c*c
PsiPrefactor = 12.0*H_0*H_0/(c*c)

class time_mod():
   def __init__(self, l_max, kVAL):
```

```
        self.kVal = kVAL

        self.n1 = 400
        self.n2 = 600
        self.n_t = self.n1 + self.n2

        self.z_start_rec = 1630.4
        self.z_end_rec = 614.2
        self.z_0 = 0.0
        self.x_start_rec = -np.log(1.0 + self.
           z_start_rec)
        self.x_end_rec = -np.log(1.0 + self.z_end_rec)
        self.x_0 = 0.0
        self.a_start_rec = 1.0/(1.0 + self.z_start_rec
           )
        self.a_end_rec = 1.0/(1.0 + self.z_end_rec)

        # Used for the x-values for the conformal time
        self.n_eta = 3000
        self.a_init = 1e-8
        self.x_eta_init = np.log(self.a_init)
        self.x_eta_end = 0

        # Set up grid
        self.x_t = np.linspace(self.x_eta_init, self.
           x_0, self.n_t)

        # Set up grid of x-values for the integrated
           eta
        self.x_eta = np.linspace(self.x_eta_init, self
           .x_eta_end, self.n_eta)   # X-values for
           the conformal time
        self.x_tau = np.linspace(self.x_eta_end, self.
           x_eta_init, self.n_eta)   # Reversed array,
            used to calculate tau

        self.l_max = l_max
        self.lValues = np.linspace(2, l_max-1, l_max
```

```
            -2)
        self.NumVariables = self.l_max + 1 + 5
        k_min = 0.1*H_0/c
        k_max = 1000*H_0/c
        self.k_N = 100
        self.k = np.array([k_min + (k_max-k_min)*(i
            /100.0)**2 for i in range(self.k_N)])
        self.k_squared = self.k*self.k
        self.ck = c*self.k

        # Arrays/lists that contains the variables for
            all values of k
        self.Theta0 = []
        self.Theta1 = []
        self.Theta2 = []
        self.Theta3 = []
        self.Theta4 = []
        self.Theta5 = []
        self.Theta6 = []
        self.delta = []
        self.deltab = []
        self.v = []
        self.vb = []
        self.Phi = []

        self.Theta1Der = np.zeros(self.n_t)
        self.Theta2Der = np.zeros(self.n_t)
        self.Theta3Der = np.zeros(self.n_t)
        self.PhiDer = np.zeros(self.n_t)
        self.vbDer = np.zeros(self.n_t)

        self.CHECKER =0

    def Get_Hubble_param(self, x):
        """ Function returns the Hubble parameter for
            a given x """
        return H_0*np.sqrt((Omega_b + Omega_m)*np.exp
            (-3*x) + Omega_r*np.exp(-4*x) +
```

```
        Omega_lambda)

    def Get_Hubble_prime(self, x):
        """ Function returns the scaled Hubble
            parameter for a given x value. See report 1
             """
        return H_0*np.sqrt((Omega_b + Omega_m)*np.exp
            (-x) + Omega_r*np.exp(-2*x) + Omega_lambda*
            np.exp(2*x))

    def Get_Hubble_prime_derivative(self, x):
        """ Function returns the derivative of the
            scaled Hubble parameter. See report 1 """
        return -H_0Squared*(0.5*(Omega_b + Omega_m)*np
            .exp(-x) + Omega_r*np.exp(-2.0*x) -
            Omega_lambda*np.exp(2.0*x))/(self.
            Get_Hubble_prime(x))

    def Get_Omegas(self, x):
        """
        Calculates the omegas as a function of
            redshift
        Will first have to calculate the energy
            densities today, which is then used to
            calculate the energy density
        for an arbitrary time. See report 1
        """
        H = self.Get_Hubble_param(x)
        rho_c = rhoCrit_factor*H**2
        Omega_m_z = rho_m0*np.exp(-3*x)/rho_c
        Omega_b_z = rho_b0*np.exp(-3*x)/rho_c
        Omega_r_z = rho_r0*np.exp(-4*x)/rho_c
        Omega_lambda_z = rho_lambda0/rho_c

        return Omega_m_z, Omega_b_z, Omega_r_z,
            Omega_lambda_z

    def Diff_eq_eta(self, eta, x_0):
```

```
        """ Returns the right hand side of the
           differential equation for the conformal
           time eta """
        dEtada = c/(self.Get_Hubble_prime(x_0))
        return dEtada

    def Cubic_Spline(self, x_values, y_values,
       n_points, x_start=np.log(1e-8), x_end=0):
        """
        Cubic spline interpolation, zeroth derivative.
            Returns interpolated values of any
           variables, for a given range of x-values
        """
        Temp_interp = interpolate.splrep(x_values,
           y_values)
        x_new = np.linspace(x_start, x_end, n_points)
        y_new = interpolate.splev(x_new, Temp_interp,
           der=0)
        return x_new, y_new

    def Cubic_Spline_OnePoint(self, x_values,
       y_values, x_point):
        """ Cubic spline for one specific point """
        Temp_interp = interpolate.splrep(x_values,
           y_values)
        y_new = interpolate.splev(x_point, Temp_interp
           , der=0)
        return y_new

    def Spline_Derivative(self, x_values, y_values,
       n_points, derivative, x_start=np.log(1e-8),
       x_end=0):
        """ Spline derivative for any functions. Using
            natural spline for the second derivative
           """
        if derivative < 1:
           raise ValueError("Derivative input in
              Spline_Derivative less than 1. Use
```

```
            Cubic_spline instead.")
        Temp_interp = interpolate.splrep(x_values,
           y_values)
        x_new = np.linspace(x_start, x_end, n_points)
        yDerivative = interpolate.splev(x_new,
           Temp_interp, der=derivative)
        if derivative == 2:
           yDerivative[0] = 0
           yDerivative[-1] = 0
        return yDerivative

    def Get_Index_Interpolation(self, X_init, X_end):
        """
        Finds the array index/component of x for a
           given x-value
        This is specifically used to zoom into the
           interpolated segment
        """
        EtaIndex1 = (np.abs(self.x_eta - X_init)).
           argmin()
        EtaIndex2 = (np.abs(self.x_eta - X_end)).
           argmin()
        if EtaIndex1-1 <= 0:
           EtaIndex1 = 0
        else:
           EtaIndex1 -= 1

        if EtaIndex2+1 >= self.n_eta:
           EtaIndex2 = self.n_eta-1
        else:
           EtaIndex2 += 1

        return EtaIndex1, EtaIndex2

    def Get_n_b(self, x):
        """ Calculate n_b (or n_H) at a given 'time' x
            """
        n_b = Omega_b*rho_c0*np.exp(-3.0*x)/m_H
```

```
        return n_b

    def Saha_equation(self, x):
        """ Solves the Saha equation. Uses numpy.roots
            solver, see report 2. Only returns the
           positive valued X_e """
        Exponential = np.exp(x)
        a = 1
        b = (Saha_b_factor/self.Get_n_b(x))*np.exp(-
           EpsTemp_factor*Exponential - 3.0*x/2.0)
        c = -b
        X_e = np.roots(np.array([a,b,c]))
        if X_e[0] > 0:
           return X_e[0]
        else:
           return X_e[1]

    def Peebles_equation(self, X_e, x_0):
        """ Solves the right hand side of the Peebles
           equation """
        n_b = self.Get_n_b(x_0)
        H = self.Get_Hubble_param(x_0)
        exp_factor = EpsTemp_factor*np.exp(x_0)
        phi2 = 0.448*np.log(exp_factor)
        alpha2 = alpha_factor*np.sqrt(exp_factor)*phi2
        beta = alpha2*beta_factor*np.exp(-3.0*x_0/2.0-
           exp_factor)
        beta2 = alpha2*beta_factor*np.exp(-3.0*x_0
           /2.0-exp_factor/4.0)
        Lambda_alpha = H*Lambda_alpha_factor/((1.0-X_e
           )*n_b)
        C_r = (Lambda_2sto1s + Lambda_alpha)/(
           Lambda_2sto1s + Lambda_alpha + beta2)
        dXedx = (C_r/H)*(beta*(1.0-X_e) - n_b*alpha2*
           X_e**2.0)

        return dXedx
```

```
    def Calculate_Xe(self):
        """ Function that calculates X_e. Initial
            condition X_e = 1 """
        X_e_TempArray = [1]
        Peeble = False
        for i in range(0,self.n_eta-1):
            if X_e_TempArray[i] > 0.99:
                X_e_TempArray.append(self.Saha_equation(
                    self.x_eta[i]))
            else:
                PeebleXe = integrate.odeint(self.
                    Peebles_equation, X_e_TempArray[i],
                    self.x_eta[i:])
                break
        PeebleXe2 = []
        for i in range(0, len(PeebleXe)-1):
            PeebleXe2.append(PeebleXe[i][0])
        self.X_e_array = np.concatenate([np.array(
            X_e_TempArray),np.array(PeebleXe2)])

    def Diff_eq_tau(self, tau, x_0):
        """
        Solves the differential equation of tau. This
            is the right hand side of the equation
        Finds the n_e value that corresponds to the x
            value, since we use a reversed x-array.
        """
        n_e = np.exp(self.Cubic_Spline_OnePoint(self.
            x_eta, np.log(self.n_e), x_0))
        dTaudx = -n_e*sigma_T*c/self.Get_Hubble_param(
            x_0)
        return dTaudx

    def Visibility_func(self, x, tau, tauDerv):
        """ Computes the visibility function (tilde)
            """
        g = np.zeros(len(tau))
        for i in range(0, len(tau)-1):
```

```
            g[i] = -tauDerv[i]*np.exp(-tau[i])
        return g

    def Kronecker_Delta_2(self, l):
        """ Kronecker delta that only returns 1 if l =
            2 """
        if l == 2:
            return 1
        else:
            return 0

    def BoltzmannEinstein_InitConditions(self, k):
        """ Initial conditions for the Boltzmann
            equations """
        Phi = 1.0
        delta_b = 3.0*Phi/2.0
        HPrime_0 = self.Get_Hubble_param(self.
            x_eta_init)
        InterpolateTauDerivative = self.
            Spline_Derivative(self.x_eta, self.Taus, 1,
             derivative = 1, x_start = self.x_eta_init,
             x_end =self.x_eta_init)
        v_b = c*k*Phi/(2.0*HPrime_0)
        Theta_0 = 0.5*Phi
        Theta_1 = -c*k*Phi/(6.0*HPrime_0)

        self.BoltzmannTightCoupling = np.array([
            Theta_0, Theta_1, delta_b, delta_b, v_b,
            v_b, Phi])
        self.NumVarTightCoupling = len(self.
            BoltzmannTightCoupling)

    def BoltzmannEinstein_InitConditions_AfterTC(self
        , k):
        """
        Properly set up all variables into a parameter
            in the tight coupling regime
        Also sets up initial conditions of the
```

```
            different parameters, that is to be
            calculated for time after recombination
        """
        Transposed = np.transpose(self.EBTightCoupling
            )
        Hprimed = self.Get_Hubble_prime(self.x_TC_grid
            )
        TauDer = self.Spline_Derivative(self.x_eta,
            self.Taus, len(Transposed[0]), derivative
            =1, x_start=self.x_TC_grid[0], x_end=self.
            x_TC_grid[-1])
        self.Theta0TC = Transposed[0]
        self.Theta1TC = Transposed[1]
        self.Theta2TC = -20.0*c*k*self.Theta1TC/(45.0*
            Hprimed*TauDer)
        self.Theta3TC = -3.0*c*k*self.Theta2TC/(7.0*
            Hprimed*TauDer)
        self.Theta4TC = -4.0*c*k*self.Theta3TC/(9.0*
            Hprimed*TauDer)
        self.Theta5TC = -5.0*c*k*self.Theta4TC/(11.0*
            Hprimed*TauDer)
        self.Theta6TC = -6.0*c*k*self.Theta5TC/(13.0*
            Hprimed*TauDer)
        self.deltaTC = Transposed[2]
        self.deltabTC = Transposed[3]
        self.vTC = Transposed[4]
        self.vbTC = Transposed[5]
        self.PhiTC = Transposed[6]

        self.BoltzmannVariablesAFTERTC_INIT = np.array
            ([self.Theta0TC[-1], self.Theta1TC[-1],
            self.Theta2TC[-1], self.Theta3TC[-1], self.
            Theta4TC[-1],
             self.Theta5TC[-1], self.Theta6TC[-1], self.
                deltaTC[-1], self.deltabTC[-1], self.vTC
                [-1], self.vbTC[-1], self.PhiTC[-1]])

    def BoltzmannEinstein_Equations(self, variables,
```

```
x_0, k):
""" Solves Boltzmann Einstein equations """
Theta_0, Theta_1, Theta_2, Theta_3, Theta_4,
   Theta_5, Theta_6, delta, delta_b, v, v_b,
   Phi = variables
# Calculating some prefactors
Hprimed = self.Get_Hubble_prime(x_0)
Hprimed_Squared = Hprimed*Hprimed
ck_Hprimed = c*k/Hprimed
# Interpolating Conformal time and Optical
   depth at the point x_0
InterTauDerivative = self.TauSpline(x_0, 1)[0]
InterEta = self.EtaSpline(x_0)[0]

R = 4.0*Omega_r/(3.0*Omega_b*np.exp(x_0))
Psi = -Phi - PsiPrefactor*(np.exp(-2.0*x_0)/(k
   *k))*Omega_r*Theta_2
ck_HprimedPsi = ck_Hprimed*Psi

dPhidx = Psi - (ck_Hprimed**2/3.0)*Phi\
      + (H_0Squared/(2.0*Hprimed_Squared))*(
         Omega_m*np.exp(-x_0)*delta + Omega_b*
         np.exp(-x_0)*delta_b + 4.0*Omega_r*np
         .exp(-2.0*x_0)*Theta_0)

ThetaDerivatives = np.zeros(self.l_max+1)
Thetas = np.array([Theta_0, Theta_1, Theta_2,
   Theta_3, Theta_4, Theta_5, Theta_6])
ThetaDerivatives[0] = -ck_Hprimed*Theta_1 -
   dPhidx
ThetaDerivatives[1] = (ck_Hprimed/3.0)*Theta_0
    - ((2.0*ck_Hprimed)/3.0)*Theta_2 \
         + (ck_HprimedPsi/3.0) +
            InterTauDerivative*(Theta_1 + v_b
            /3.0)
for l in range(2, self.l_max):
   ThetaDerivatives[l] = l*ck_Hprimed/(2.0*l
      +1.0)*Thetas[l-1] - ck_Hprimed*((l+1.0)
```

```
            /(2.0*l+1.0))*Thetas[l+1] \
                  + InterTauDerivative*(Thetas[l] -
                     0.1*Thetas[l]*self.
                     Kronecker_Delta_2(l))
      ThetaDerivatives[self.l_max] = ck_Hprimed*
         Thetas[self.l_max-1] - c*((self.l_max + 1)
         /(Hprimed*InterEta))*Thetas[self.l_max]\
                  + InterTauDerivative*Thetas[self.
                     l_max]

      dDeltadx = ck_Hprimed*v - 3.0*dPhidx
      dDeltabdx = ck_Hprimed*v_b - 3.0*dPhidx
      dvdx = -v - ck_HprimedPsi
      dvbdx = -v_b - ck_HprimedPsi +
         InterTauDerivative*R*(3.0*Theta_1 + v_b)

      derivatives = np.array([ThetaDerivatives[0],
         ThetaDerivatives[1], ThetaDerivatives[2],
         ThetaDerivatives[3], ThetaDerivatives[4] ,
         ThetaDerivatives[5]\
               , ThetaDerivatives[6], dDeltadx,
                  dDeltabdx, dvdx, dvbdx, dPhidx])
      return derivatives

   def TightCouplingRegime(self, variables, x_0, k):
      """ Boltzmann equation in the tight coupling
         regime """
      Theta_0, Theta_1, delta, delta_b, v, v_b, Phi
         = variables
      # Calculating some prefactors
      Hprimed = self.Get_Hubble_prime(x_0)
      HprimedDer = self.Get_Hubble_prime_derivative(
         x_0)
      HprimeDer_Hprime = HprimedDer/Hprimed
      Hprimed_Squared = Hprimed*Hprimed
      ck_Hprimed = c*k/Hprimed
      # Interpolating Conformal time and Optical
         depth (its derivatives) at the point x_0
```

```
        InterTauDerivative = self.TauSpline(x_0, 1)[0]
        InterTauDoubleDer = self.TauSpline(x_0, 2)[0]
        InterEta = self.EtaSpline(x_0)[0]

        Theta_2 = -20.0*ck_Hprimed*Theta_1/(45.0*
           InterTauDerivative)
        R = 4.0*Omega_r/(3.0*Omega_b*np.exp(x_0))
        Psi = -Phi - PsiPrefactor*Omega_r*Theta_2/(k*k
           *np.exp(2.0*x_0))
        dPhidx = Psi - (ck_Hprimed**2/3.0)*Phi\
              + (H_0Squared/(2.0*Hprimed_Squared))*(
                 Omega_m*np.exp(-x_0)*delta + Omega_b*
                 np.exp(-x_0)*delta_b + 4.0*Omega_r*np
                 .exp(-2.0*x_0)*Theta_0)
        dTheta0dx = -ck_Hprimed*Theta_1 - dPhidx
        q = -(((1.0 - 2.0*R)*InterTauDerivative + (1.0
            + R)*InterTauDoubleDer)*(3.0*Theta_1 + v_b
           ) - ck_Hprimed*Psi +
             (1.0-HprimeDer_Hprime)*ck_Hprimed*(-
                Theta_0 + 2.0*Theta_2) - ck_Hprimed*
                dTheta0dx)/((1.0+R)*InterTauDerivative
                + HprimeDer_Hprime - 1.0)

        dDeltadx = ck_Hprimed*v - 3.0*dPhidx
        dDeltabdx = ck_Hprimed*v_b - 3.0*dPhidx
        dvdx = -v - ck_Hprimed*Psi
        dvbdx = (-v_b - ck_Hprimed*Psi + R*(q +
           ck_Hprimed*(-Theta_0 + 2.0*Theta_2) -
           ck_Hprimed*Psi))/(1.0+R)
        dTheta1dx = (q-dvbdx)/3.0
        derivatives = np.array([dTheta0dx, dTheta1dx,
           dDeltadx, dDeltabdx, dvdx, dvbdx, dPhidx])
        return np.reshape(derivatives, len(derivatives
           ))

    def MergeAndFinalize(self):
        """ Merges computed values of the variables in
            and after tight coupling. Saves them to
```

```
        their respective arrays defined in the
        initializer """
    Transposed_AFTERTC = np.transpose(self.
        EBAfterTC)
    Theta0Merge = np.concatenate([self.Theta0TC,
        Transposed_AFTERTC[0]])
    Theta1Merge = np.concatenate([self.Theta1TC,
        Transposed_AFTERTC[1]])
    Theta2Merge = np.concatenate([self.Theta2TC,
        Transposed_AFTERTC[2]])
    Theta3Merge = np.concatenate([self.Theta3TC,
        Transposed_AFTERTC[3]])
    Theta4Merge = np.concatenate([self.Theta4TC,
        Transposed_AFTERTC[4]])
    Theta5Merge = np.concatenate([self.Theta5TC,
        Transposed_AFTERTC[5]])
    Theta6Merge = np.concatenate([self.Theta6TC,
        Transposed_AFTERTC[6]])
    deltaMerge = np.concatenate([self.deltaTC,
        Transposed_AFTERTC[7]])
    deltabMerge = np.concatenate([self.deltabTC,
        Transposed_AFTERTC[8]])
    vMerge = np.concatenate([self.vTC,
        Transposed_AFTERTC[9]])
    vbMerge = np.concatenate([self.vbTC,
        Transposed_AFTERTC[10]])
    PhiMerge = np.concatenate([self.PhiTC,
        Transposed_AFTERTC[11]])

    self.Theta0.append(Theta0Merge)
    self.Theta1.append(Theta1Merge)
    self.Theta2.append(Theta2Merge)
    self.Theta3.append(Theta3Merge)
    self.Theta4.append(Theta4Merge)
    self.Theta5.append(Theta5Merge)
    self.Theta6.append(Theta6Merge)
    self.delta.append(deltaMerge)
    self.deltab.append(deltabMerge)
```

```
        self.v.append(vMerge)
        self.vb.append(vbMerge)
        self.Phi.append(PhiMerge)

        self.AllVariables = np.array([self.Theta0,
            self.Theta1, self.Theta2, self.Theta3, self
            .Theta4, self.Theta5, self.Theta6,
                                self.delta, self.deltab,
                                    self.v, self.vb, self.
                                    Phi])

    def Get_TC_end(self, k):
        """ Computes the time when tight coupling ends
            . See report. """
        TauDeriv = self.Spline_Derivative(self.x_eta,
            self.Taus, self.n_eta, derivative=1,
            x_start=self.x_eta[0], x_end=self.x_eta
            [-1])
        kHprimedTau = c*k/(self.Get_Hubble_prime(self.
            x_eta)*TauDeriv)

        Condition1 = np.where(np.fabs(kHprimedTau)
            >0.1)[0]
        Condition2 = np.where(np.fabs(TauDeriv) >
            10.0)[0]
        indexList = np.intersect1d(Condition1,
            Condition2)
        if len(indexList) == 0:
            index = Condition2[-1]
        else:
            index = indexList[0]

        if self.x_eta[index] > self.x_start_rec:
            return self.x_start_rec
        else:
            return self.x_eta[index]

    def Compute_derivatives_TC(self, x_0, k):
```

```
""" Computes and saves the left hand side of
   the diff. eqs for tight coupling """
Hprimed = self.Get_Hubble_prime(x_0)
HprimedDer = self.Get_Hubble_prime_derivative(
   x_0)
HprimeDer_Hprime = HprimedDer/Hprimed
Hprimed_Squared = Hprimed*Hprimed
ck_Hprimed = c*k/Hprimed
# Interpolating Conformal time and Optical
   depth (its derivatives) at the point x_0
InterTauDerivative = np.transpose(self.
   TauSpline(x_0, 1))[0]
InterTauDoubleDer = np.transpose(self.
   TauSpline(x_0, 2))[0]
InterEta = np.transpose(self.EtaSpline(x_0))
   [0]

Theta_2 = -20.0*ck_Hprimed*self.Theta1TC
   /(45.0*InterTauDerivative)
R = 4.0*Omega_r/(3.0*Omega_b*np.exp(x_0))
Psi = -self.PhiTC - PsiPrefactor*Omega_r*self.
   Theta2TC/(k*k*np.exp(2.0*x_0))
dPhidx = Psi - (ck_Hprimed**2/3.0)*self.PhiTC\
      + (H_0Squared/(2.0*Hprimed_Squared))*(
         Omega_m*np.exp(-x_0)*self.deltaTC +
         Omega_b*np.exp(-x_0)*self.deltabTC\
      + 4.0*Omega_r*np.exp(-2.0*x_0)*self.
         Theta0TC)
dTheta0dx = -ck_Hprimed*self.Theta1TC - dPhidx
q = -(((1.0 - 2.0*R)*InterTauDerivative + (1.0
    + R)*InterTauDoubleDer)*(3.0*self.Theta1TC
    + self.vbTC) - ck_Hprimed*Psi +
    (1.0-HprimeDer_Hprime)*ck_Hprimed*(-self.
       Theta0TC + 2.0*self.Theta2TC) -
       ck_Hprimed*dTheta0dx)/((1.0+R)*
       InterTauDerivative + HprimeDer_Hprime
       -1.0)
dvbdx = (-self.vbTC - ck_Hprimed*Psi + R*(q +
```

```
            ck_Hprimed*(-self.Theta0TC + 2.0*self.
               Theta2TC) - ck_Hprimed*Psi))/(1.0+R)
        dTheta1dx = (q-dvbdx)/3.0


        self.Theta1Der[0:len(x_0)] = dTheta1dx
        self.Theta2Der[0:len(x_0)] = dTheta1dx
        self.Theta3Der[0:len(x_0)] = dTheta1dx
        self.PhiDer[0:len(x_0)] = dPhidx
        self.vbDer[0:len(x_0)] = dvbdx

    def Compute_derivatives_AFTERTC(self, x_0, k):
        """ Computes the left hand side of the diff.
           eqs after tight coupling """
        # Calculating some prefactors
        Hprimed = self.Get_Hubble_prime(x_0)
        Hprimed_Squared = Hprimed*Hprimed
        ck_Hprimed = c*k/Hprimed
        # Interpolating Conformal time and Optical
           depth at the point x_0
        InterTauDerivative = np.transpose(self.
           TauSpline(x_0, 1))[0]
        InterEta = np.transpose(self.EtaSpline(x_0))
           [0]

        R = 4.0*Omega_r/(3.0*Omega_b*np.exp(x_0))
        Psi = -np.array(self.Phi[0][self.n1:]) -
           PsiPrefactor*(np.exp(-2.0*x_0)/(k*k))*
           Omega_r*np.array(self.Theta2[0][self.n1:])
        ck_HprimedPsi = ck_Hprimed*Psi
        dPhidx = Psi - (ck_Hprimed**2/3.0)*np.array(
           self.Phi[0][self.n1:])\
               + (H_0Squared/(2.0*Hprimed_Squared))*(
                  Omega_m*np.exp(-x_0)*np.array(self.
                  delta[0][self.n1:]) \
               + Omega_b*np.exp(-x_0)*np.array(self.
                  deltab[0][self.n1:]) + np.array(4.0*
                  Omega_r*np.exp(-2.0*x_0)*self.Theta0
```

```
            [0][self.n1:]))
    ThetaDerivatives = [0,0,0,0,0,0,0]
    Thetas = np.array([self.Theta0[0][self.n1:],
       self.Theta1[0][self.n1:], self.Theta2[0][
       self.n1:], self.Theta3[0][self.n1:],
              self.Theta4[0][self.n1:], self.
                 Theta5[0][self.n1:], self.Theta6
                 [0][self.n1:]])
    ThetaDerivatives[0] = -ck_Hprimed*np.array(
       self.Theta1[0][self.n1:]) - dPhidx
    ThetaDerivatives[1] = (ck_Hprimed/3.0)*np.
       array(self.Theta0[0][self.n1:]) - ((2.0*
       ck_Hprimed)/3.0)*np.array(self.Theta2[0][
       self.n1:]) \
             + (ck_HprimedPsi/3.0) +
                InterTauDerivative*(np.array(self.
                Theta1[0][self.n1:]) + np.array(
                self.vb[0][self.n1:])/3.0)
    for l in range(2, self.l_max):
       ThetaDerivatives[l] = l*ck_Hprimed/(2.0*l
          +1.0)*Thetas[l-1] - ck_Hprimed*((l+1.0)
          /(2.0*l+1.0))*Thetas[l+1] \
                + InterTauDerivative*(Thetas[l] -
                   0.1*Thetas[l]*self.
                   Kronecker_Delta_2(l))
    ThetaDerivatives[self.l_max] = ck_Hprimed*
       Thetas[self.l_max-1] - c*((self.l_max + 1)
       /(Hprimed*InterEta))*Thetas[self.l_max]\
                + InterTauDerivative*Thetas[self.
                   l_max]
    dvbdx = -np.array(self.vb[0][self.n1:]) -
       ck_HprimedPsi + InterTauDerivative*R*(3.0*
       np.array(self.Theta1[0][self.n1:]) + np.
       array(self.vb[0][self.n1:]))

    self.Theta1Der[self.n1:] = ThetaDerivatives[1]
    self.Theta2Der[self.n1:] = ThetaDerivatives[2]
    self.Theta3Der[self.n1:] = ThetaDerivatives[3]
```

```
        self.PhiDer[self.n1:] = dPhidx
        self.vbDer[self.n1:] = dvbdx

    def Compute_Results(self, n_interp_points,
        x_start = -np.log(1.0 + 1630.4), x_end = -np.
        log(1.0 + 614.2)):
        """ Computes all the relevant results of the
            Boltzmann-Einstein equations """
        self.ScipyEta = integrate.odeint(self.
            Diff_eq_eta, 0, self.x_eta)
        self.EtaSpline = interpolate.CubicSpline(self.
            x_eta, self.ScipyEta)
        # Calculate X_e, and n_e
        self.Calculate_Xe()
        self.n_e = self.X_e_array*self.Get_n_b(self.
            x_eta)
        # Calculates tau
        self.Taus = integrate.odeint(self.Diff_eq_tau,
             0, self.x_tau)[::-1]  # Calculate tau and
            reverse array
        self.TauSpline = interpolate.CubicSpline(self.
            x_eta, self.Taus)

        self.BoltzmannEinstein_InitConditions(self.
            kVal)
        x_tc_end = self.Get_TC_end(self.kVal)
        self.x_TC_grid = np.linspace(self.x_eta_init,
            x_tc_end, self.n1)
        x_afterTC_grid = np.linspace(x_tc_end, self.
            x_eta_end, self.n2)
        self.EBTightCoupling = integrate.odeint(self.
            TightCouplingRegime, np.transpose(self.
            BoltzmannTightCoupling),
                self.x_TC_grid, args=(self.kVal,))
        self.BoltzmannEinstein_InitConditions_AfterTC(
            self.kVal)

        self.Compute_derivatives_TC(self.x_TC_grid,
```

```
            self.kVal)

        self.EBAfterTC = integrate.odeint(self.
            BoltzmannEinstein_Equations, self.
            BoltzmannVariablesAFTERTC_INIT,
                x_afterTC_grid, args=(self.kVal,))
        self.MergeAndFinalize()
        self.Compute_derivatives_AFTERTC(
            x_afterTC_grid, self.kVal)

        return self.AllVariables, [self.Theta1Der,
            self.Theta2Der, self.Theta3Der, self.PhiDer
            , self.vbDer]

    def Compute_tau_and_g(self):
        """ Computes optical depth and visibility
            function and returns it to Power spectrum
            class """
        ScipyEta = integrate.odeint(self.Diff_eq_eta,
            0, self.x_eta)
        self.Calculate_Xe()
        self.n_e = self.X_e_array*self.Get_n_b(self.
            x_eta)
        Taus = integrate.odeint(self.Diff_eq_tau, 0,
            self.x_tau)[::-1]
        TauDerivative = self.Spline_Derivative(self.
            x_eta, Taus, self.n_eta, derivative=1)

        g_tilde = self.Visibility_func(self.x_eta,
            Taus, TauDerivative)
        new_x_grid, Taus_smallerGrid = self.
            Cubic_Spline(self.x_eta, Taus, self.n_t)
        new_x_grid, g_tilde_smallerGrid = self.
            Cubic_Spline(self.x_eta, g_tilde, self.n_t)
        new_x_grid, Eta_smallerGrid = self.
            Cubic_Spline(self.x_eta, ScipyEta, self.n_t
            )
```

```
        return Taus_smallerGrid, g_tilde_smallerGrid,
            Eta_smallerGrid

class Plotter:
    def __init__(self, savefile, k_array, variables):
        self.savefile = savefile    # If savefile = 0,
            plots the data. If savefile = 1, saves the
            plots into a pdf
        self.k = k_array
        self.variables = variables
        if savefile != 0 and savefile != 1:
            print 'Current value of savefile = ',
                savefile
            raise ValueError('Argument savefig not
                properly set. Try savefile = 1 (saves as
                 pdf) or savefile = 0 (do not save as
                pdf)')

        self.n1 = 400
        self.n2 = 600
        self.n_t = self.n1 + self.n2
        self.a_init = 1e-8
        self.x_init = np.log(self.a_init)
        self.x_0 = 0.0
        # Set up x grid
        self.x_t = np.linspace(self.x_init, self.x_0,
            self.n_t)

        # Arrays/lists that contains the variables for
            all values of k
        self.Theta0 = []
        self.Theta1 = []
        self.Theta2 = []
        self.Theta3 = []
        self.Theta4 = []
        self.Theta5 = []
        self.Theta6 = []
        self.delta = []
```

```
        self.deltab = []
        self.v = []
        self.vb = []
        self.Phi = []
        self.Theta1Deriv = []
        self.Theta2Deriv = []
        self.Theta3Deriv = []
        self.PhiDeriv = []
        self.vbDeriv = []

    def Sort_Arrays(self):
        """ Sorts the variables to their respective
            arrays """
        for i in range(len(self.k)):
            self.Theta0.append(self.variables[i][0][0])
            self.Theta1.append(self.variables[i][0][1])
            self.Theta2.append(self.variables[i][0][2])
            self.Theta3.append(self.variables[i][0][3])
            self.Theta4.append(self.variables[i][0][4])
            self.Theta5.append(self.variables[i][0][5])
            self.Theta6.append(self.variables[i][0][6])
            self.delta.append(self.variables[i][0][7])
            self.deltab.append(self.variables[i][0][8])
            self.v.append(self.variables[i][0][9])
            self.vb.append(self.variables[i][0][10])
            self.Phi.append(self.variables[i][0][11])

            self.Theta1Deriv.append(self.variables[i
                ][1][0])
            self.Theta2Deriv.append(self.variables[i
                ][1][1])
            self.Theta3Deriv.append(self.variables[i
                ][1][2])
            self.PhiDeriv.append(self.variables[i
                ][1][3])
            self.vbDeriv.append(self.variables[i
                ][1][4])
```

```
    def Write_Outfile(self, filedir, filename, k,
        k_index):
        """ Saves data to a text file """
        results_dir = filedir
        if not os.path.exists(results_dir):
            os.makedirs(results_dir)

        text_file = open(os.path.join(results_dir,
            filename), "w")
        text_file.write(("Theta0, Theta1, Theta2,
            Theta3, Theta4, Theta5, Theta6, delta,
            delta_b, v, v_b, phi, Theta1Der. Theta2Der,
             Theta3Der, vbDer, PhiDer, k=%.4e H_0/c\n")
            \
                    %(self.k[k_index]*c/H_0))
        for i in range(self.n_t):
            text_file.write(("%.7e %.7e %.7e %.7e %.7e
                %.7e %.7e %.7e %.7e %.7e %.7e %.7e %.7e
                %.7e %.7e %.7e %.7e\n") \
                % (self.Theta0[k_index][0][i], self.
                    Theta1[k_index][0][i], self.Theta2[
                    k_index][0][i], self.Theta3[k_index
                    ][0][i],
                self.Theta4[k_index][0][i], self.Theta5[
                    k_index][0][i], self.Theta6[k_index
                    ][0][i], self.delta[k_index][0][i],
                self.deltab[k_index][0][i], self.v[
                    k_index][0][i], self.vb[k_index][0][i
                    ], self.Phi[k_index][0][i],
                self.Theta1Deriv[k_index][i], self.
                    Theta2Deriv[k_index][i], self.
                    Theta3Deriv[k_index][i], self.vbDeriv
                    [k_index][i], self.PhiDeriv[k_index][
                    i]))
        text_file.close()

    def Plot_results(self, filedir, filename):
        """ Plots the results """
```

```
        self.Sort_Arrays()
        for i in range(len(self.k)):
            filename2 = filename + str(i) + '.txt'
            self.Write_Outfile(filedir, filename2, self
                .k[i], i)

        print "Nothing to plot here!"
        if self.savefile == 1:
            a=1
        else:
            plt.show()

class Power_Spectrum():
    def __init__(self, save_figure, k_array,
        file_directory, variable_filename):
        self.k = k_array
        self.filedir = file_directory
        self.save_figure = save_figure

        self.n1 = 400
        self.n2 = 600
        self.n_t = self.n1 + self.n2

        # Set up x grid
        self.a_init = 1e-8
        self.x_init = np.log(self.a_init)
        self.x_0 = 0.0
        self.x_t = np.linspace(self.x_init, self.x_0,
            self.n_t)

        # Set up larger grid
        self.x_LargeGrid = np.linspace(self.x_init,
            self.x_0, 5000)
        self.k_LargeGrid = np.linspace(self.k[0], self
            .k[-1], 5000)

        # Values of l, used for the Bessel function
        self.l_full_grid = np.linspace(0,1200, 3000)
```

```
        self.l_values = []
        self.l_val_grid = np.array
           ([2,3,4,6,8,10,12,15,20,30,40,50,60,70,80,90,100,120,140,16

                 ,180,200,225,250,275,300,350,400,450,500,550,600,650,

        for ls in self.l_val_grid:
           self.l_values.append([ls])

        self.Theta0 = []
        self.Theta1 = []
        self.Theta2 = []
        self.Theta3 = []
        self.Theta4 = []
        self.Theta5 = []
        self.Theta6 = []
        self.delta = []
        self.deltab = []
        self.v = []
        self.vb = []
        self.Phi = []

        self.Theta1Deriv = []
        self.Theta2Deriv = []
        self.Theta3Deriv = []
        self.vbDeriv = []
        self.PhiDeriv = []

        read_start = time.clock()
        for i in range(len(k)):
           filename = variable_filename + str(i) + '.
              txt'
           self.read_Boltzmann_variables(filename)
        print 'Read file time: ', time.clock() -
           read_start, 's'

    def read_Boltzmann_variables(self, filename):
        """ Reads data computed from the Boltzmann-
```

```
   Einstein equations """
datafile = open(self.filedir+filename,'r')
SkipFirstLine = 0
Theta0_temp = []
Theta1_temp = []
Theta2_temp = []
Theta3_temp = []
Theta4_temp = []
Theta5_temp = []
Theta6_temp = []
delta_temp = []
deltab_temp = []
v_temp = []
vb_temp = []
Phi_temp = []
Theta1Der_temp = []
Theta2Der_temp = []
Theta3Der_temp = []
vbDer_temp = []
PhiDer_temp = []

for line in datafile:
   data_set = line.split()
   if SkipFirstLine == 0:
      SkipFirstLine = 1
   else:
      Theta0_temp.append(float(data_set[0]))
      Theta1_temp.append(float(data_set[1]))
      Theta2_temp.append(float(data_set[2]))
      Theta3_temp.append(float(data_set[3]))
      Theta4_temp.append(float(data_set[4]))
      Theta5_temp.append(float(data_set[5]))
      Theta6_temp.append(float(data_set[6]))
      delta_temp.append(float(data_set[7]))
      deltab_temp.append(float(data_set[8]))
      v_temp.append(float(data_set[9]))
      vb_temp.append(float(data_set[10]))
      Phi_temp.append(float(data_set[11]))
```

```
            Theta1Der_temp.append(float(data_set
               [12]))
            Theta2Der_temp.append(float(data_set
               [13]))
            Theta3Der_temp.append(float(data_set
               [14]))
            vbDer_temp.append(float(data_set[15]))
            PhiDer_temp.append(float(data_set[16]))

        self.Theta0.append(np.array(Theta0_temp))
        self.Theta1.append(np.array(Theta1_temp))
        self.Theta2.append(np.array(Theta2_temp))
        self.Theta3.append(np.array(Theta3_temp))
        self.Theta4.append(np.array(Theta4_temp))
        self.Theta5.append(np.array(Theta5_temp))
        self.Theta6.append(np.array(Theta6_temp))
        self.delta.append(np.array(delta_temp))
        self.deltab.append(np.array(deltab_temp))
        self.v.append(np.array(v_temp))
        self.vb.append(np.array(vb_temp))
        self.Phi.append(np.array(Phi_temp))
        self.Theta1Deriv.append(np.array(
           Theta1Der_temp))
        self.Theta2Deriv.append(np.array(
           Theta2Der_temp))
        self.Theta3Deriv.append(np.array(
           Theta3Der_temp))
        self.vbDeriv.append(np.array(vbDer_temp))
        self.PhiDeriv.append(np.array(PhiDer_temp))

    def Get_SourceFunction(self, x_1, k, k_index):
        """ Computes the source function for a given k
            value """
        x = self.Get_x_grid_with_TC(k, largeGrid=0)
        Hprimed = self.timemod_instance.
           Get_Hubble_prime(x)
        Hprimed_Derivative = self.timemod_instance.
           Get_Hubble_prime_derivative(x)
```

```
Hspline = interpolate.CubicSpline(self.x_t,
   Hprimed)
HprimedDoubleDer = Hspline(x)
TauSpline = interpolate.CubicSpline(self.x_t,
   self.Tau)
TAUU = TauSpline(x)
InterTauDerivative = TauSpline(x, 1)
InterTauDoubleDer = TauSpline(x, 2)
ck_Hprimed = c*k/Hprimed
HprimeDer_Hprime = Hprimed_Derivative/Hprimed
k_squared = k*k

Pi = self.Theta2[k_index]
Psi = -self.Phi[k_index] - PsiPrefactor*
   Omega_r*self.Theta2[k_index]/(np.exp(2.0*x)
   *k_squared)

Theta2Der = self.Theta2Deriv[k_index]
PsiDer = -self.PhiDeriv[k_index] -
   PsiPrefactor*Omega_r*(-2.0*np.exp(-2.0*x)*
   self.Theta2[k_index] + self.Theta2Deriv[
   k_index]*np.exp(-2.0*x))/k_squared

Pi_derivative = self.Theta2Deriv[k_index]
Pi_doubleDer = (2.0*ck_Hprimed/5.0)*(-
   HprimeDer_Hprime*self.Theta1[k_index] +
   self.Theta1Deriv[k_index]) \
         + 3.0*(InterTauDoubleDer*Pi +
            InterTauDerivative*Pi_derivative)
            /10.0\
         - (3.0*ck_Hprimed/5.0)*(-
            HprimeDer_Hprime*self.Theta3[
            k_index] + self.Theta3Deriv[
            k_index])

g_spline = interpolate.CubicSpline(self.x_t,
   self.g_tilde)
g_tilde = g_spline(x)
```

```
        g_tilde_derivative = g_spline(x, 1)
        g_tilde_doubleDer = g_spline(x, 2)

        dHpHpderdx = (Hprimed_Derivative**2.0 +
           Hprimed*HprimedDoubleDer)
        ThirdTermDerivative = Hprimed_Derivative*
           g_tilde*self.vb[k_index] + Hprimed*
           g_tilde_derivative*self.vb[k_index] \
                       + Hprimed*g_tilde*self.vbDeriv[
                          k_index]
        LastTermDerivative = g_tilde*Pi*dHpHpderdx +
           3.0*Hprimed*Hprimed_Derivative*(
           g_tilde_derivative*Pi + g_tilde*
           Pi_derivative) \
                       + (Hprimed**2.0)*(
                          g_tilde_doubleDer*Pi + 2.0*
                          g_tilde_derivative*
                          Pi_derivative + g_tilde*
                          Pi_doubleDer)

        S_tilde = g_tilde*(self.Theta0[k_index] + Psi
           + Pi/4.0) + np.exp(-TAUU)*(PsiDer - self.
           PhiDeriv[k_index]) \
                  - ThirdTermDerivative/(c*k) + 3.0*
                     LastTermDerivative/(4.0*c_Squared*
                     k_squared)
        return S_tilde

    def Interpolate_LargerGrid(self, SourceFunctions)
       :
        """ Interpolates the source function to a
           larger grid.  """
        # Interpolate k grid
        Interpolated_SourceFunc_unsorted = []
        for i in range(self.n_t):
           for j in range(len(k)):
              S_x_grid = [Sfunc_values[i] for
                 Sfunc_values in SourceFunctions]
```

```
            Temp_spline = interpolate.CubicSpline(self.
                k, S_x_grid)
            SourceFunc_k_new = Temp_spline(self.
                k_LargeGrid)
            Interpolated_SourceFunc_unsorted.append(
                SourceFunc_k_new)
        Interpolated_k_grid = np.transpose(
            Interpolated_SourceFunc_unsorted)

        # Interpolate x grid
        Interpolated_SourceFunc = []
        for i in range(len(Interpolated_k_grid)):
            xgrid_s1, xgrid_s2 = self.
                Get_x_grid_with_TC(self.k_LargeGrid[i],
                largeGrid=0, split_grid=1)
            x_grid_s1, x_grid_s2 = self.
                Get_x_grid_with_TC(self.k_LargeGrid[i],
                largeGrid=1, split_grid=1)

            Spline_s1 = interpolate.CubicSpline(
                xgrid_s1, Interpolated_k_grid[i][0:self.
                n1])
            Spline_s2 = interpolate.CubicSpline(
                xgrid_s2, Interpolated_k_grid[i][self.n1
                :])
            SourceFunc_x_new_s1 = Spline_s1(x_grid_s1)
            SourceFunc_x_new_s2 = Spline_s2(x_grid_s2)
            SourceFunc_x_new = np.concatenate([
                SourceFunc_x_new_s1, SourceFunc_x_new_s2
                ])
            Interpolated_SourceFunc.append(np.array(
                SourceFunc_x_new))
        return np.array(Interpolated_SourceFunc)

    def Get_x_grid_with_TC(self, k, largeGrid=0,
        split_grid = 0):
        TauDeriv = self.Spline_Derivative(self.x_t,
            self.Tau, self.x_LargeGrid, derivative=1)
```

```
kHprimedTau = c*k/(self.timemod_instance.
   Get_Hubble_prime(self.x_LargeGrid)*TauDeriv
   )

Condition1 = np.where(np.fabs(kHprimedTau)
   >0.1)[0]
Condition2 = np.where(np.fabs(TauDeriv) >
   10.0)[0]
indexList = np.intersect1d(Condition1,
   Condition2)
if len(indexList) == 0:
   index = Condition2[-1]
else:
   index = indexList[0]

if self.x_LargeGrid[index] > self.
   timemod_instance.x_start_rec:
   x_tc_end = self.timemod_instance.
      x_start_rec
else:
   x_tc_end = self.x_LargeGrid[index]

if largeGrid == 0:
   Xinit_TC = np.linspace(self.x_init,
      x_tc_end, self.n1)
   Xend_TC = np.linspace(x_tc_end, self.x_0,
      self.n2)
   X_grid_w_TC = np.concatenate([Xinit_TC,
      Xend_TC])
else:
   Xinit_TC = np.linspace(self.x_init,
      x_tc_end, 2000)
   Xend_TC = np.linspace(x_tc_end, self.x_0,
      3000)
   X_grid_w_TC = np.concatenate([Xinit_TC,
      Xend_TC])

if split_grid == 0:
```

```
            return X_grid_w_TC
        else:
            return Xinit_TC, Xend_TC
    def Spline_Derivative(self, x_values, y_values,
        xgrid, derivative):
        """ Cubic spline for any functions. Using
            natural spline for the second derivative
            """
        Spline = interpolate.CubicSpline(x_values,
            y_values)
        Interpolated_value = Spline(xgrid, derivative)
        if derivative == 2:
            Interpolated_value[0] = 0
            Interpolated_value[-1] = 0
        return Interpolated_value

    def Compute_transfer_function(self,
        theta_directory, filename_theta):
        """ Computes the transfer function and saves
            the computed values to a text file """
        # Get optical depth and visibility function
            from time_mod class
        self.timemod_instance = time_mod(l_max=6, kVAL
            =self.k)
        self.Tau, self.g_tilde, self.Eta_smallgrid =
            self.timemod_instance.Compute_tau_and_g()
        Eta_spline = interpolate.CubicSpline(self.x_t,
             self.Eta_smallgrid)
        self.Eta = Eta_spline(self.x_LargeGrid)

        # Compute source function
        Source_functions_smallgrid = []
        Stemp = []
        start = time.clock()
        for j in range(len(k)):
            X_grid_w_TC = self.Get_x_grid_with_TC(self.
                k[j])
            S_tilde = self.Get_SourceFunction(
```

```
            X_grid_w_TC, self.k[j], j)
        Source_functions_smallgrid.append(S_tilde)
    print 'Computing source function time: ', time
        .clock() - start, 's'

    # Interpolate source function over larger grid
    start2 = time.clock()
    self.Interpolated_SourceFunction = self.
        Interpolate_LargerGrid(
        Source_functions_smallgrid)
    print 'Interpolation time: ', time.clock() -
        start2, 's'

    # Save Bessel arguments
    self.BesselArgs = []
    self.X_grids = []
    for ks in self.k_LargeGrid:
        X_TT = self.Get_x_grid_with_TC(ks,
            largeGrid=1)
        ETA = Eta_spline(X_TT)
        self.X_grids.append(X_TT)
        self.BesselArgs.append(ks*(ETA[-1] - ETA))

    # Compute Bessel splines
    time_Bess = time.clock()
    x_bessel_grid = np.linspace(0, 5400, 10000)
    Bessel_functions = special.spherical_jn(self.
        l_values, x_bessel_grid)
    print "Bessel function time: ", time.clock() -
         time_Bess, "s"
    SPLINES = []
    for i in range(len(self.l_values)):
        B_spline = interpolate.CubicSpline(
            x_bessel_grid, Bessel_functions[i])
        SPLINES.append(B_spline)

    # Compute transfer functions
    TrTime = time.clock()
```

```
        Transfer_funcs_k = []
        for ls in range(len(self.l_values)):
            Integrals = []
            Bessel_spline = SPLINES[ls]
            for ks in range(len(self.k_LargeGrid)):
                New_bessel = Bessel_spline(self.
                    BesselArgs[ks])
                Integrand = self.
                    Interpolated_SourceFunction[ks]*
                    New_bessel
                TransferFunc_integral = integrate.trapz(
                    Integrand, self.X_grids[ks])
                Integrals.append(TransferFunc_integral)

            Transfer_funcs_k.append(np.array(Integrals)
                )
        print "Transfer function time: ", time.clock()
            - TrTime, "s"

        # Saves data of the transfer functions to text
            file
        results_dir = theta_directory
        if not os.path.exists(results_dir):
            os.makedirs(results_dir)

        textfile = open(os.path.join(results_dir,
            filename_theta), 'w')
        textfile.write("# Values along the columns are
            Theta_l. Different value of l along the
            rows \n")
        for j in range(len(self.l_values)):
            for i in range(len(Transfer_funcs_k[j])):
                textfile.write("%.8e " %(
                    Transfer_funcs_k[j][i]))
            textfile.write("\n")
        return np.array(Transfer_funcs_k)

    def Read_transfer_func_data(self, Theta_dir,
```

```
        filename_theta):
        """ Reads transfer function data from a text
            file """
        if not os.path.exists(Theta_dir):
            raise ValueError('Filedirectory', Theta_dir
                , 'does not exist! Save the values first
                .')
        datafile = open(os.path.join(Theta_dir,
            filename_theta), 'r')
        Transferfunctions = []
        SkipFirstLine = 0
        for line in datafile:
            data_set = line.split()
            if SkipFirstLine == 0:
                SkipFirstLine = 1
            else:
                Temp_transfer_array = []
                for i in range(len(data_set)):
                    Temp_transfer_array.append(float(
                        data_set[i]))
                Transferfunctions.append(np.array(
                    Temp_transfer_array))
        return np.array(Transferfunctions)

    def Compute_power_spectrum(self, Theta_dir,
        filename_theta, read_data=1, save_data=0):
        """
        Computes the power spectrum C_l. Returns a
            normalized and interpolated power spectrum
            over a large grid.
        If read_data = 0, then program computes the
            transfer function from scratch. read_data =
             1 by default
        """
        if read_data == 0:
            save_data = 1
        else:
            Transfer_functions = self.
```

```
            Read_transfer_func_data(Theta_dir,
            filename_theta)

        if save_data == 1:
            Transfer_functions = self.
               Compute_transfer_function(Theta_dir,
               filename_theta)

        Integrand = (((c*self.k_LargeGrid/H_0)**(n_s
           -1.0))/self.k_LargeGrid)*Transfer_functions
           **2.0
        Power_spectrum_smallGrid = integrate.trapz(
           Integrand, self.k_LargeGrid)
        Power_spec_spline = interpolate.CubicSpline(
           self.l_val_grid, Power_spectrum_smallGrid)
        Power_spectrum = Power_spec_spline(self.
           l_full_grid)
        Normalization_factor = 5775.0/np.max(self.
           l_full_grid*(self.l_full_grid+1)*
           Power_spectrum/(2.0*np.pi))

        Transfer_func_splines = []
        for ks in range(len(self.k_LargeGrid)):
            T_spline = interpolate.CubicSpline(self.
               l_val_grid, np.transpose(
               Transfer_functions)[ks])
            Transfer_func_splines.append(T_spline)

        return Normalization_factor*Power_spectrum,
           Transfer_func_splines

    def Read_planck_data(self):
        """ Reads data from Planck """
        CMB_data_filename = "COM_PowerSpect_CMB-TT-hiL
           -full_R2.02.txt"
        datafile = open(os.path.join("../Planck_data",
            CMB_data_filename), 'r')
        SkipLines = 0
```

```
        PLanck_l_values = []
        Planck_PS = []
        Planck_PS_Err = []
        for line in datafile:
            if SkipLines <= 2:
                SkipLines += 1
            else:
                data_set = line.split()
                PLanck_l_values.append(float(data_set
                    [0]))
                Planck_PS.append(float(data_set[1]))
                Planck_PS_Err.append(float(data_set[2]))

        self.PLanck_l_values = np.array(
            PLanck_l_values)
        self.Planck_PS = np.array(Planck_PS)
        self.Planck_PS_Err = np.array(Planck_PS_Err)

    def Plot_results(self, Theta_dir, filename_theta,
         PlotDir, BestFitModel, r_data=1):
        """ Plots the results """
        self.Read_planck_data()
        Power_spectrum, Transfer_func_splines = self.
            Compute_power_spectrum(Theta_dir,
            filename_theta, read_data=r_data)

        fig1 = plt.figure()
        ax1 = plt.subplot(111)
        plt.hold("on")
        ax1.plot(self.l_full_grid, self.l_full_grid*(
            self.l_full_grid+1)*Power_spectrum/(2.0*np.
            pi), 'r-', label="Theoretical data")
        ax1.errorbar(self.PLanck_l_values, self.
            Planck_PS, yerr=self.Planck_PS_Err, ecolor=
            'g', alpha=0.4, label="Planck data")
        ax1.legend(loc = 'lower left', bbox_to_anchor
            =(0.6,0.6), ncol=1, fancybox=True)
        plt.xlabel('$l$')
```

```
plt.ylabel(r'$l(l+1)C_l/2\pi$')

fig2 = plt.figure()
ax2 = plt.subplot(111)
plt.hold("on")
ax2.plot(self.l_full_grid, self.l_full_grid*(
   self.l_full_grid+1)*Power_spectrum/(2.0*np.
   pi), 'r-', label="Theoretical data")
ax2.plot(self.PLanck_l_values, self.Planck_PS,
    'g', alpha=0.5, label="Planck data")
ax2.legend(loc = 'lower left', bbox_to_anchor
   =(0.6,0.6), ncol=1, fancybox=True)
plt.xlabel('$l$')
plt.ylabel(r'$l(l+1)C_l/2\pi$')

fig3 = plt.figure()
ax3 = plt.subplot(111)
plt.hold("on")
ax3.plot(self.l_full_grid[5:],
   Transfer_func_splines[0](self.l_full_grid
   [5:]), label='$k= %.2f H_0/c$' %(self.
   k_LargeGrid[0]*c/H_0))
ax3.plot(self.l_full_grid[5:],
   Transfer_func_splines[1000](self.
   l_full_grid[5:]), label='$k= %.2f H_0/c$'
   %(self.k_LargeGrid[1000]*c/H_0))
ax3.plot(self.l_full_grid[5:],
   Transfer_func_splines[2000](self.
   l_full_grid[5:]), label='$k= %.2f H_0/c$'
   %(self.k_LargeGrid[2000]*c/H_0))
ax3.plot(self.l_full_grid[5:],
   Transfer_func_splines[3000](self.
   l_full_grid[5:]), label='$k= %.2f H_0/c$'
   %(self.k_LargeGrid[3000]*c/H_0))
ax3.plot(self.l_full_grid[5:],
   Transfer_func_splines[4000](self.
   l_full_grid[5:]), label='$k= %.2f H_0/c$'
   %(self.k_LargeGrid[4000]*c/H_0))
```

```
ax3.plot(self.l_full_grid[5:],
   Transfer_func_splines[-1](self.l_full_grid
   [5:]), label='$k= %.2f H_0/c$' %(self.
   k_LargeGrid[-1]*c/H_0))
ax3.legend(loc = 'lower left', bbox_to_anchor
   =(0.3,0.75), ncol=2, fancybox=True)
plt.xlabel('$l$')
plt.ylabel('$\Theta_l(k)$')

fig4 = plt.figure()
ax4 = plt.subplot(111)
plt.hold("on")
ax4.semilogy(self.l_full_grid[5:], (
   Transfer_func_splines[0](self.l_full_grid
   [5:]))**2.0/(self.k_LargeGrid[0]),\
        label='$k= %.2f H_0/c$' %(self.
           k_LargeGrid[0]*c/H_0))
ax4.semilogy(self.l_full_grid[5:], (
   Transfer_func_splines[1000](self.
   l_full_grid[5:]))**2.0/(self.k_LargeGrid
   [1000]),\
        label='$k= %.2f H_0/c$' %(self.
           k_LargeGrid[1000]*c/H_0))
ax4.semilogy(self.l_full_grid[5:], (
   Transfer_func_splines[2000](self.
   l_full_grid[5:]))**2.0/(self.k_LargeGrid
   [2000]),\
        label='$k= %.2f H_0/c$' %(self.
           k_LargeGrid[2000]*c/H_0))
ax4.semilogy(self.l_full_grid[5:], (
   Transfer_func_splines[3000](self.
   l_full_grid[5:]))**2.0/(self.k_LargeGrid
   [3000]),\
        label='$k= %.2f H_0/c$' %(self.
           k_LargeGrid[3000]*c/H_0))
ax4.semilogy(self.l_full_grid[5:], (
   Transfer_func_splines[4000](self.
   l_full_grid[5:]))**2.0/(self.k_LargeGrid
```

```
    [4000]),\
        label='$k= %.2f H_0/c$' %(self.
            k_LargeGrid[4000]*c/H_0))
ax4.semilogy(self.l_full_grid[5:], (
    Transfer_func_splines[-1](self.l_full_grid
    [5:]))**2.0/(self.k_LargeGrid[-1]),\
        label='$k= %.2f H_0/c$' %(self.
            k_LargeGrid[-1]*c/H_0))
ax4.legend(loc = 'lower left', bbox_to_anchor
    =(0.0,0.0), ncol=1, fancybox=True)
plt.xlabel('$l$')
plt.ylabel('$\Theta_l(k)^2/k$')

fig5 = plt.figure()
ax5 = plt.subplot(111)
ax5.plot(self.l_full_grid, self.l_full_grid*(
    self.l_full_grid+1)*Power_spectrum/(2.0*np.
    pi), label="Theoretical data")
plt.xlabel('$l$')
plt.ylabel(r'$l(l+1)C_l/2\pi$')

if BestFitModel == 0:
    ax1.set_title('Power spectrum from
        theoretical and Planck data, with
        errorbar.')
    ax2.set_title('Power spectrum from
        theoretical and Planck data. No errorbar
        .')
    ax3.set_title('Plot of transfer functions
        as a function of $l$ for different
        k_values')
    ax4.set_title('Plot of transfer functions
        squared as a function of $l$ \n for
        different k-values')
    ax5.set_title('Power spectrum of the
        theoretical data.')

elif BestFitModel == 1:
```

```
            ax1.set_title('Power spectrum from
               theoretical and Planck data, with
               errorbar. \n Best fit model')
            ax2.set_title('Power spectrum from
               theoretical and Planck data. No errorbar
               . \n Best fit model')
            ax3.set_title('Plot of transfer functions
               as a function of $l$ for different
               k_values. \n Best fit model')
            ax4.set_title('Plot of transfer functions
               squared as a function of $l$ \n for
               different k-values. Best fit model')
            ax5.set_title('Power spectrum of the
               theoretical data. Best fit model')

        if not os.path.exists(PlotDir):
            os.makedirs(PlotDir)
        if self.save_figure == 1:
            if BestFitModel == 0:
                fig1.savefig(PlotDir + '
                   PowerSpectrumVsPlanckwError.png')
                fig2.savefig(PlotDir + '
                   PowerSpectrumVsPlanck.png')
            elif BestFitModel == 1:
                fig1.savefig(PlotDir + '
                   PowerSpectrumVsPlanckwErrorBestFit.
                   png')
                fig2.savefig(PlotDir + '
                   PowerSpectrumVsPlanckBestFit.png')
            fig3.savefig(PlotDir + 'Thetal.png')
            fig4.savefig(PlotDir + 'ThetaSquaredk.png')
            fig5.savefig(PlotDir + '
               PowerSpectrumTheoretical.png')
        else:
            plt.show()

def SolveEquations(k):
    """ Function used to call the solver class for
```

```
        different values of k """
    solver = time_mod(l_max=6, kVAL=k)
    ComputedVariables = solver.Compute_Results(100)
    return ComputedVariables

if __name__ == '__main__':
    print 'Starting program'
    # Defines the range of k
    k_min = 0.1*H_0/c
    k_max = 1000.0*H_0/c
    k_N = 100
    k = np.array([k_min + (k_max-k_min)*(i/100.0)**2
        for i in range(k_N)])

    num_processes = 4    # Sets number of proceses to
        compute in parallel
    Compute_BoltzmannEquations = 0   # Computes all
        of Boltzmann equations from scratch if set to
        1
    BestFitModel = 0  # Set to 1 if testing a best
        fit model by modifiying cosmological
        parameters.
                    # Will compute Boltzmann variables
                        from scratch if set to 1
    Read_theta_data = 1 # Reads from saved theta data
        from Power_Spectrum class. Set to 1 if data
        is pre-computed

    if BestFitModel == 0:
        # Directories for the standard, unchanged,
            model
        Variable_dir = '../VariableData/'
        Variable_filename = 'BoltzmannVariables_k'
        Theta_dir = '../ThetaData/'
        Plot_dir = '../Plots/'
    elif BestFitModel == 1:
        # Directories for the best fit model
        Compute_BoltzmannEquations = 1
```

```
    Read_theta_data = 0
    Variable_dir = '../VariableDataBestFit/'
    Variable_filename = 'BoltzmannVariables_k'
    Theta_dir = '../ThetaDataBestFit/'
    Plot_dir = '../PlotsBestFit/'
    ### Change of Omegas parameters below. Other
       cosmological parameters to be changed at
       the top.
    Omega_m = 0.24
    Omega_b = 0.09
    Omega_r = 8.8e-5

if Compute_BoltzmannEquations == 1:
    print 'Computing Boltzmann equations ...'
    p = mp.Pool(num_processes)
    time_start = time.clock()
    Solution = p.map(SolveEquations, k)
    print "time elapsed: ",  time.clock() -
       time_start, "s"
    PlotInstance = Plotter(savefile=1, k_array=k,
       variables=Solution)
    PlotInstance.Plot_results(Variable_dir,
       Variable_filename)

PS_solver = Power_Spectrum(save_figure=1, k_array
   =k, file_directory=Variable_dir,
   variable_filename=Variable_filename)
PS_solver.Plot_results(Theta_dir, 'Theta_data.txt
   ', Plot_dir, BestFitModel, r_data=
   Read_theta_data)
```

References

[1] P. Callin, https://arxiv.org/abs/astro-ph/0606683.