AST5220 - Cosmology 2
Milestone 3

Alex Ho

April 26, 2017

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

`https://github.com/AHo94/AST5220_Projects/tree/master/Project3`

Mathematics

Tight coupling regime

The tight coupling regime is when $kc/(\mathcal{H}\tau') << 1$. During this regime, we do not have to compute the derivatives of θ_l, for $l \in [2, l_{max}]$. Because of this, the end of tight coupling will vary for different values of k. Like in Callin, we let tight coupling end when $|kc/(\mathcal{H}\tau')| > 0.1$ and while $|\tau'| > 10$. We also put the constraint that tight coupling has to end before recombination starts (or at the time when recombination starts).

Initial conditions

The initial conditions, tight coupling, are only set for $\Theta_0, \Theta_1, \delta, \delta_b, v, v_b$ and Φ variables. The values of the Θ_l, (for $l > 1$) are not computed during this time period. However, once tight coupling ends, we use the computed values of Θ_1 to compute Θ_2, which can then be used to compute Θ_3 and so on. The initial conditions for the 'non tight coupling' regime will be the last computed values from tight coupling.

Numerics

The program

The program is a further continuation from the previous project. Two new functions, `TightCouplingRegime` and `BoltzmannEinstein_Equations` contains the right hand side of the differential equations during and after tight coupling time respectively. The initial conditions, for tight coupling, is set using the `BoltzmannEinstein_InitConditions` function, and the initial conditions for the time after tight coupling is set with the `BoltzmannEinstein_InitConditions_AfterTC` function.
The function `Get_TC_end` computes the value of x when tight coupling ends based on the constraints described in the previous section. Once the computation is completed, the function `MergeAndFinalize` will sort all the computed variables to their respective arrays.

`Scipy`'s odeint is once again used to compute the differential equations. However, with the increased number of variables that we have to compute, the time it takes to finish the computation increases significantly. The program is thus parallel programmed using `Python`'s `multiprocessing` library. Doing this reduces the computation time significantly, but still takes a little while to run [1].

[1] Takes roughly 10 minutes to run on one of the UiO terminals with 4 CPUs and roughly 40 minutes with 1 CPU

Each value of k returns a multidimensional array, which contains all the computed variables for all values of x. The multiprocessing function `map` uses all the k values to compute and returns all these multidimensional arrays into one single array. For this case, the resulting array is an $100 \times 12 \times$(number of x-points) array. This array is sorted in the `Plotter` class which then plots the results.

Plots

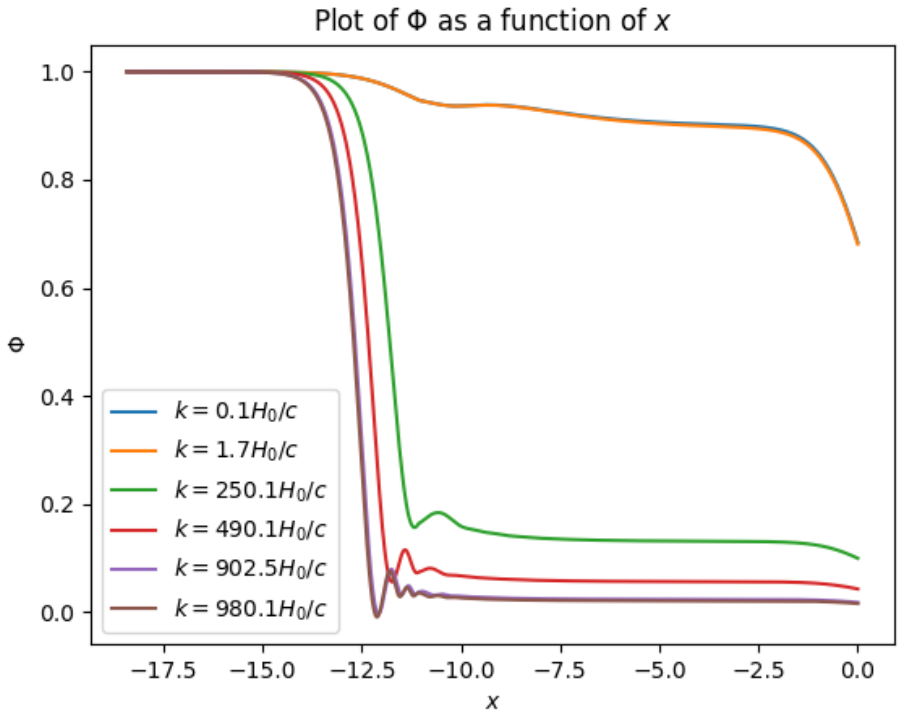

Figure 1: Plot of the gravitational potential Φ as a function of x.

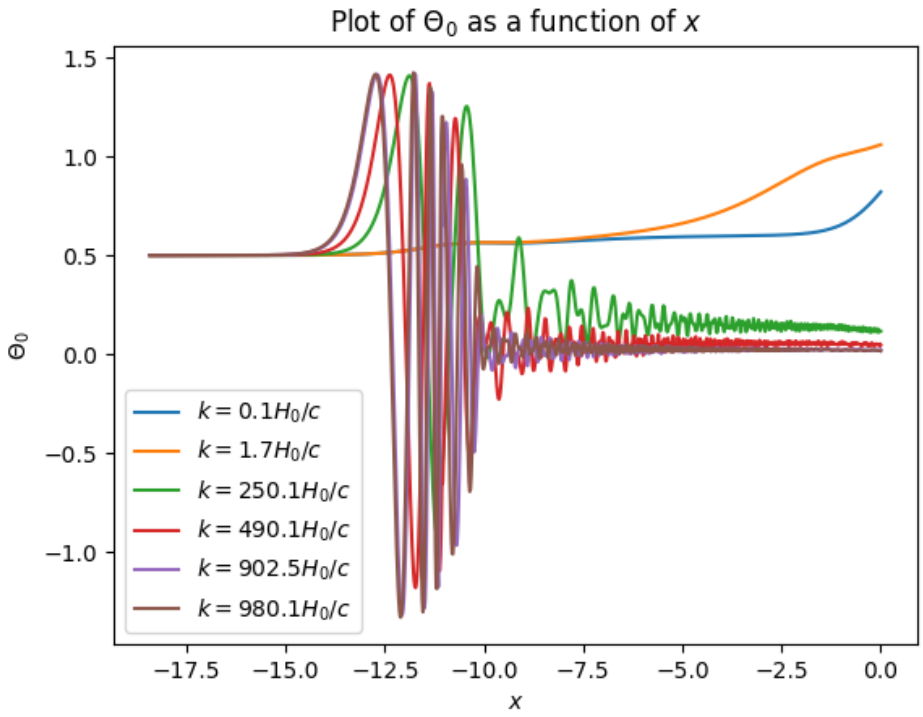

Figure 2: The monopole Θ_0 as a function of x.

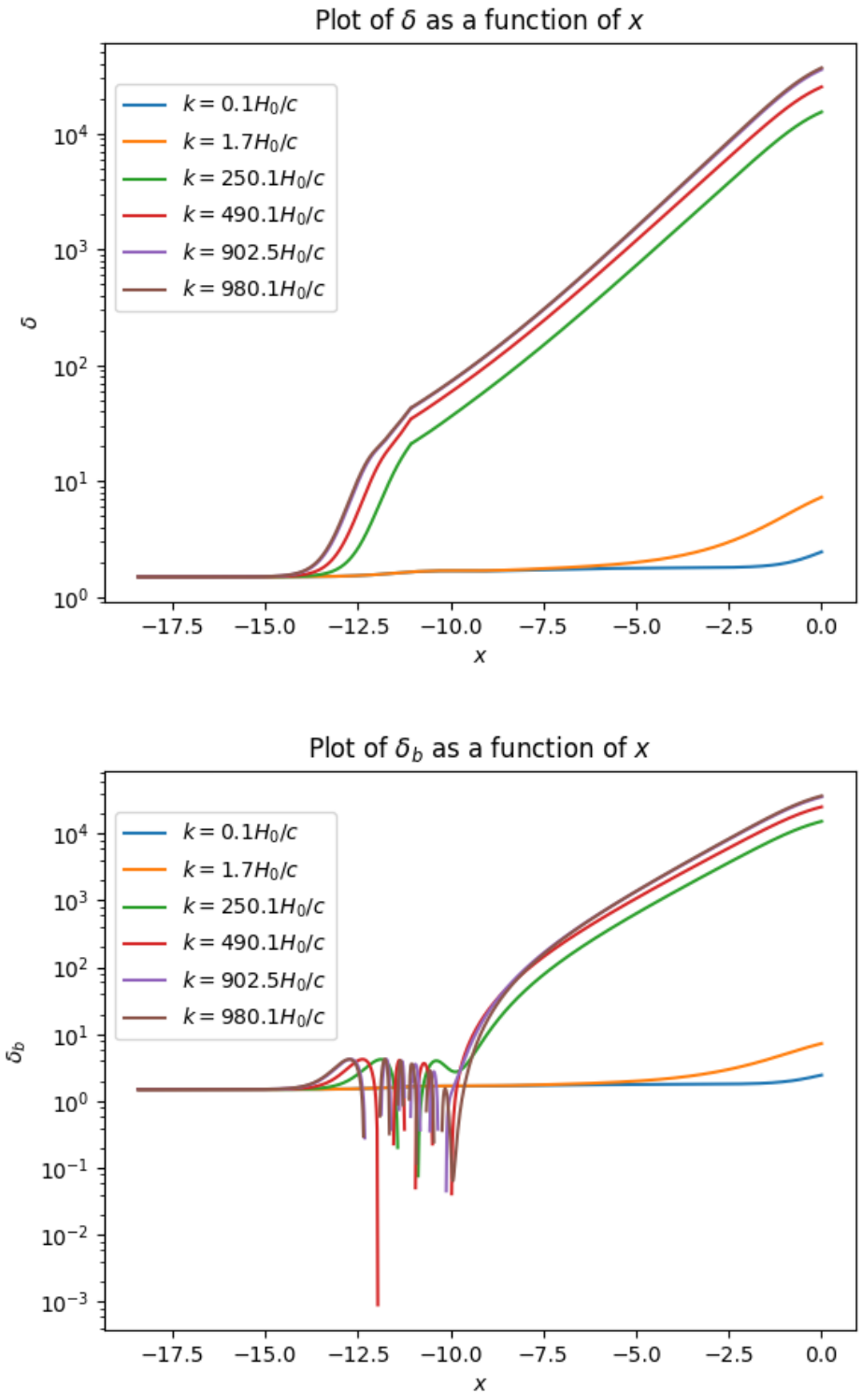

Figure 3: Density perturbation for dark matter δ (top image) and for baryons δ_b (bottom image) as a function of x.

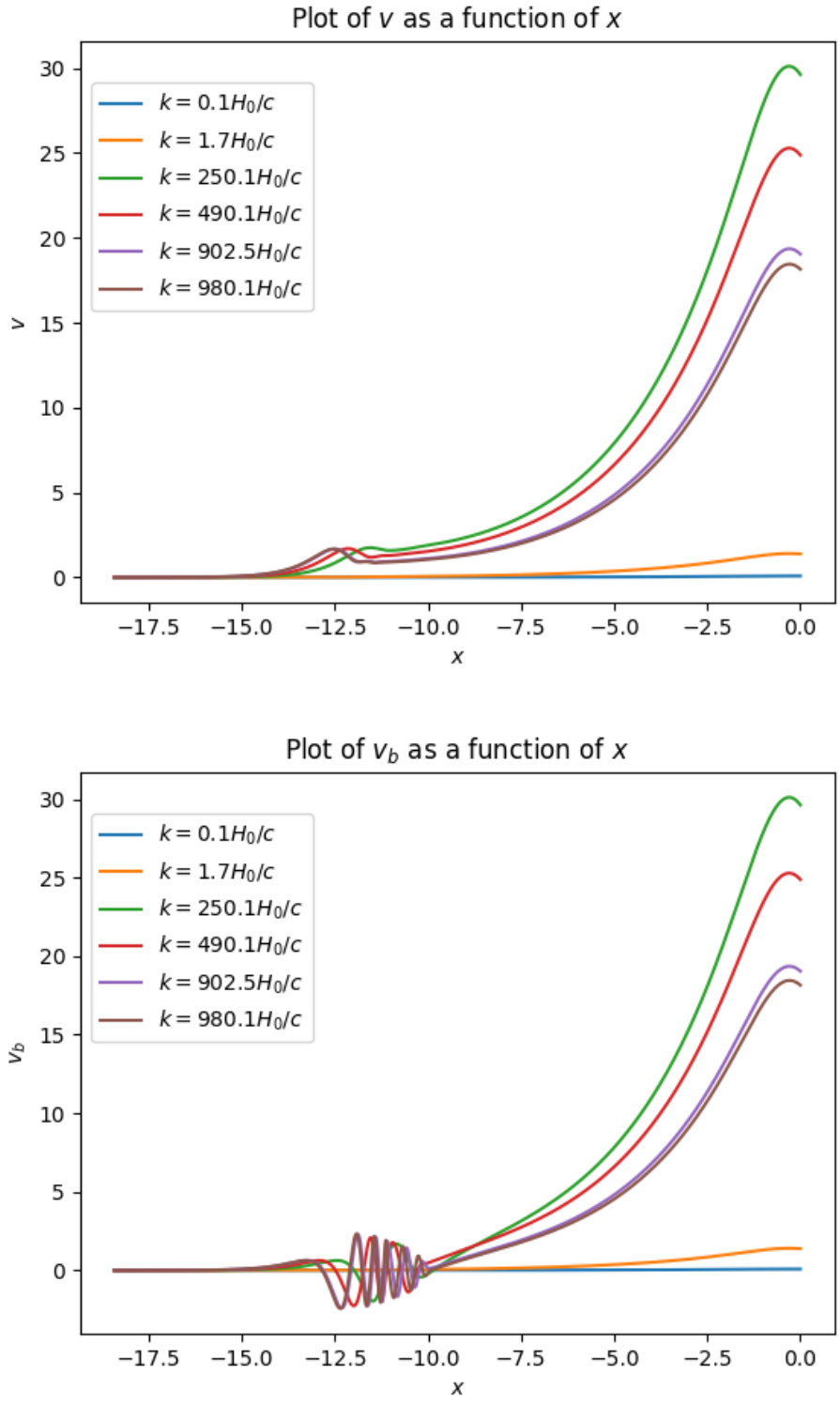

Figure 4: The velocity of dark matter v (top image) and velocity of baryons v (bottom image) as a function of x.

The code

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

# 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 = 1.0
A_s = 1.0
h0 = 0.7
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, savefile, l_max, kVAL):
      self.savefile = savefile      # If savefig =
         0, plots the data. If savefig = 1, saves
         the plots into a pdf
```

```
        self.kVal = kVAL

        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.time_start = time.clock()
        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.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.Spline_Derivative(
       self.x_eta, self.Taus, 1, derivative=1,
       x_start=x_0, x_end=x_0)
    InterEta = self.Cubic_Spline_OnePoint(self.
       x_eta, self.ScipyEta, x_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.Spline_Derivative(
   self.x_eta, self.Taus, 1, derivative=1,
   x_start=x_0, x_end=x_0)
InterTauDoubleDer = self.Spline_Derivative(
   self.x_eta, self.Taus, 1, derivative=2,
   x_start=x_0, x_end=x_0)
InterEta = self.Cubic_Spline_OnePoint(self.
   x_eta, self.ScipyEta, x_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 Write_Outfile(self, filename, variables, k):
        """ Saves data to a text file """
        Transposed = variables
        text_file = open(filename, "w")
        text_file.write(("Theta0, Theta1, Theta2,
            Theta3, Theta4, Theta5, Theta6, delta,
            delta_b, v, v_b, phi, k=%.8e \n") %self.k[k
            ])
        for i in range(self.n_t):
            text_file.write(("%.6e %.6e %.6e %.6e %.6e
                %.6e %.6e %.6e %.6e %.6e %.6e \n")
                \
            %(Transposed[k][i], Transposed[k+self.k_N][
                i], Transposed[k+self.k_N*2][i],
                Transposed[k+self.k_N*3][i],\
             Transposed[k+self.k_N*4][i], Transposed[k+
                self.k_N*5][i], Transposed[k+self.k_N
                *6][i], Transposed[k+self.k_N*7][i],\
             Transposed[k+self.k_N*8][i], Transposed[k+
                self.k_N*9][i], Transposed[k+self.k_N
                *10][i], Transposed[k+self.k_N*11][i]))
        text_file.close()

    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 """
        self.ScipyEta = integrate.odeint(self.
            Diff_eq_eta, 0, self.x_eta)
```

```
        # Calculate X_e, n_e and interpolates n_e as a
            test
        self.Calculate_Xe()
        self.n_e = self.X_e_array*self.Get_n_b(self.
           x_eta)
        x_eta_new, n_e_NewLogarithmic = self.
           Cubic_Spline(self.x_eta, np.log(self.n_e),
           n_interp_points)
        # Calculates tau
        self.Taus = integrate.odeint(self.Diff_eq_tau,
            0, self.x_tau)[::-1]  # Calculate tau and
           reverse array

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


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 = []

    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])
            self.Theta1.append(self.variables[i][1])
            self.Theta2.append(self.variables[i][2])
            self.Theta3.append(self.variables[i][3])
            self.Theta4.append(self.variables[i][4])
            self.Theta5.append(self.variables[i][5])
            self.Theta6.append(self.variables[i][6])
            self.delta.append(self.variables[i][7])
            self.deltab.append(self.variables[i][8])
            self.v.append(self.variables[i][9])
            self.vb.append(self.variables[i][10])
            self.Phi.append(self.variables[i][11])

    def Plot_results(self):
        """ Plots the results """
        self.Sort_Arrays()

        fig1 = plt.figure()
        ax1 = plt.subplot(111)
        plt.hold("on")
        ax1.plot(self.x_t, self.Phi[0][0], label=r'$k
            = %.1f H_0/c$' %(self.k[0]*c/H_0))
        ax1.plot(self.x_t, self.Phi[4][0], label=r'$k
            = %.1f H_0/c$' %(self.k[4]*c/H_0))
        ax1.plot(self.x_t, self.Phi[50][0], label=r'$k
            = %.1f H_0/c$' %(self.k[50]*c/H_0))
        ax1.plot(self.x_t, self.Phi[70][0], label=r'$k
            = %.1f H_0/c$' %(self.k[70]*c/H_0))
        ax1.plot(self.x_t, self.Phi[-5][0], label=r'$k
            = %.1f H_0/c$' %(self.k[-5]*c/H_0))
        ax1.plot(self.x_t, self.Phi[-1][0], label=r'$k
            = %.1f H_0/c$' %(self.k[-1]*c/H_0))
        ax1.legend(loc='lower left', bbox_to_anchor
            =(0,0), ncol=1, fancybox=True)
        plt.xlabel('$x$')
        plt.ylabel('$\Phi$')
```

```
plt.title('Plot of $\Phi$ as a function of $x$
    ')

fig2 = plt.figure()
ax2 = plt.subplot(111)
plt.hold("on")
ax2.plot(self.x_t, self.Theta0[0][0], label='
    $k = %.1f H_0/c$' %(self.k[0]*c/H_0))
ax2.plot(self.x_t, self.Theta0[4][0], label='
    $k = %.1f H_0/c$' %(self.k[4]*c/H_0))
ax2.plot(self.x_t, self.Theta0[50][0], label='
    $k = %.1f H_0/c$' %(self.k[50]*c/H_0))
ax2.plot(self.x_t, self.Theta0[70][0], label='
    $k = %.1f H_0/c$' %(self.k[70]*c/H_0))
ax2.plot(self.x_t, self.Theta0[-5][0], label='
    $k = %.1f H_0/c$' %(self.k[-5]*c/H_0))
ax2.plot(self.x_t, self.Theta0[-1][0], label='
    $k = %.1f H_0/c$' %(self.k[-1]*c/H_0))
ax2.legend(loc = 'lower left', bbox_to_anchor
    =(0,0), ncol=1, fancybox=True)
plt.xlabel('$x$')
plt.ylabel(r'$\Theta_0$')
plt.title(r'Plot of $\Theta_0$ as a function
    of $x$')

fig3 = plt.figure()
ax3 = plt.subplot(111)
plt.hold("on")
ax3.semilogy(self.x_t, self.delta[0][0], label
    ='$k = %.1f H_0/c$' %(self.k[0]*c/H_0))
ax3.semilogy(self.x_t, self.delta[4][0], label
    ='$k = %.1f H_0/c$' %(self.k[4]*c/H_0))
ax3.semilogy(self.x_t, self.delta[50][0],
    label='$k = %.1f H_0/c$' %(self.k[50]*c/H_0
    ))
ax3.semilogy(self.x_t, self.delta[70][0],
    label='$k = %.1f H_0/c$' %(self.k[70]*c/H_0
    ))
```

```
ax3.semilogy(self.x_t, self.delta[-5][0],
   label='$k = %.1f H_0/c$' %(self.k[-5]*c/H_0
   ))
ax3.semilogy(self.x_t, self.delta[-1][0],
   label='$k = %.1f H_0/c$' %(self.k[-1]*c/H_0
   ))
ax3.legend(loc = 'lower left', bbox_to_anchor
   =(0,0.5), ncol=1, fancybox=True)
plt.xlabel('$x$')
plt.ylabel(r'$\delta$')
plt.title(r'Plot of $\delta$ as a function of
   $x$')

fig4 = plt.figure()
ax4 = plt.subplot(111)
plt.hold("on")
ax4.semilogy(self.x_t, self.deltab[0][0],
   label='$k = %.1f H_0/c$' %(self.k[0]*c/H_0)
   )
ax4.semilogy(self.x_t, self.deltab[4][0],
   label='$k = %.1f H_0/c$' %(self.k[4]*c/H_0)
   )
ax4.semilogy(self.x_t, self.deltab[50][0],
   label='$k = %.1f H_0/c$' %(self.k[50]*c/H_0
   ))
ax4.semilogy(self.x_t, self.deltab[70][0],
   label='$k = %.1f H_0/c$' %(self.k[70]*c/H_0
   ))
ax4.semilogy(self.x_t, self.deltab[-5][0],
   label='$k = %.1f H_0/c$' %(self.k[-5]*c/H_0
   ))
ax4.semilogy(self.x_t, self.deltab[-1][0],
   label='$k = %.1f H_0/c$' %(self.k[-1]*c/H_0
   ))
ax4.legend(loc = 'lower left', bbox_to_anchor
   =(0,0.5), ncol=1, fancybox=True)
plt.xlabel('$x$')
plt.ylabel(r'$\delta_b$')
```

```
plt.title(r'Plot of $\delta_b$ as a function
   of $x$')

fig5 = plt.figure()
ax5 = plt.subplot(111)
plt.hold("on")
ax5.plot(self.x_t, self.v[0][0], label='$k =
   %.1f H_0/c$' %(self.k[0]*c/H_0))
ax5.plot(self.x_t, self.v[4][0], label='$k =
   %.1f H_0/c$' %(self.k[4]*c/H_0))
ax5.plot(self.x_t, self.v[50][0], label='$k =
   %.1f H_0/c$' %(self.k[50]*c/H_0))
ax5.plot(self.x_t, self.v[70][0], label='$k =
   %.1f H_0/c$' %(self.k[70]*c/H_0))
ax5.plot(self.x_t, self.v[-5][0], label='$k =
   %.1f H_0/c$' %(self.k[-5]*c/H_0))
ax5.plot(self.x_t, self.v[-1][0], label='$k =
   %.1f H_0/c$' %(self.k[-1]*c/H_0))
ax5.legend(loc = 'lower left', bbox_to_anchor
   =(0,0.5), ncol=1, fancybox=True)
plt.xlabel('$x$')
plt.ylabel(r'$v$')
plt.title(r'Plot of $v$ as a function of $x$')


fig6 = plt.figure()
ax6 = plt.subplot(111)
plt.hold("on")
ax6.plot(self.x_t, self.vb[0][0], label='$k =
   %.1f H_0/c$' %(self.k[0]*c/H_0))
ax6.plot(self.x_t, self.vb[4][0], label='$k =
   %.1f H_0/c$' %(self.k[4]*c/H_0))
ax6.plot(self.x_t, self.vb[50][0], label='$k =
    %.1f H_0/c$' %(self.k[50]*c/H_0))
ax6.plot(self.x_t, self.vb[70][0], label='$k =
    %.1f H_0/c$' %(self.k[70]*c/H_0))
ax6.plot(self.x_t, self.vb[-5][0], label='$k =
    %.1f H_0/c$' %(self.k[-5]*c/H_0))
```

```
        ax6.plot(self.x_t, self.vb[-1][0], label='$k =
            %.1f H_0/c$' %(self.k[-1]*c/H_0))
        ax6.legend(loc = 'lower left', bbox_to_anchor
           =(0,0.5), ncol=1, fancybox=True)
        plt.xlabel('$x$')
        plt.ylabel(r'$v_b$')
        plt.title(r'Plot of $v_b$ as a function of $x$
           ')

        if self.savefile == 1:
            fig1.savefig('../Plots/Phi.png')
            fig2.savefig('../Plots/Theta0.png')
            fig3.savefig('../Plots/delta.png')
            fig4.savefig('../Plots/deltaBaryon.png')
            fig5.savefig('../Plots/velocity.png')
            fig6.savefig('../Plots/velocityBaryon.png')
        else:
            plt.show()


def SolveEquations(k):
    """ Function used to call the solver class for
       different values of k """
    solver = time_mod(savefile=1, l_max=6, kVAL=k)
    ComputedVariables = solver.Compute_Results(100)
    return ComputedVariables

if __name__ == '__main__':
    # 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)])
    # Sets number of proceses and starts computing in
        parallell
    num_processes = 4
    print 'Computing ...'
```

```
time_start = time.clock()
p = mp.Pool(num_processes)
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()
```

References

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