# Minimal code example for solving AST3220 project 2

Jakob Borg

March 26, 2024

This is a minimal code example demonstrating a possible outline for the code you are developing in this project. The general idea is that the number of particles we consider is given as an input argument, so the system of equations are easily extended. This greatly reduces the time required to debug and test the reaction equations as we can test the code while gradually increasing the number of species one by one.

Note, this is not a functioning code and it will not run as some of the class methods are missing, as well as the modules for the background equations and reaction rates. I've tried to make the example as minimal and simple as possible to show of the design regarding particle species, but still replicate the structure of my actual code.

```python
# Jakob Borg, UiO, AST3220 Cosmology I, 2023
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp

# Own imports:
from reaction_rates import ReactionRates  # Module with the equations for each reaction
from background import Background, cgs  # Module background functions and constants


class BBN:
    """
    General Big Bang nucleosynthesis class solving the Boltzmann equations for a collection
    of particle species in the early universe.
    """

    def __init__(self, NR_interacting_species: int = 2, **background_kwargs) -> None:
        """
        The class takes NR_interacting_species as input on initialization. This is a number
        between 2 and 8, determining the collection of particles to be considered:

        Keyword Arguments:
            NR_interacting_species {int} -- The number of particle species (default: {2}),
                                            max = 8
            background_kwargs {dict} -- Arguments passed to the background, e.g. Neff and
                                            Omega_b0
        """

        self.NR_species = NR_interacting_species  # The number of particle species
        self.species_labels = ["n", "p", "D", "T", "He3", "He4", "Li7", "Be7"]  # The
                                        particle names
        self.mass_number = [1, 1, 2, 3, 3, 4, 7, 7]  # The particle atomic numbers
        self.NR_points = 1001  # the number of points for our solution arrays
        self.RR = ReactionRates()  # Initiate the set of reaction rates equations
        # Initiate the background equations, e.g. equations for the Hubble parameter and
                                        rho_b:
        self.background = Background(**background_kwargs)

    def get_ODE(self, lnT: float, Y: np.ndarray) -> np.ndarray:
        """
        Computes the right hand side of the ODE for Boltzmanns equation
        for elements/particle species in Y

        Arguments:
            lnT {float} -- natural log of temperature
            Y {np.ndarray} -- array of relative abundance for each species.
                              With all included species Y takes the form:
                              n, p, D, T, He3, He4, Li7, Be7 = Y
        """
        T = np.exp(lnT)
        T9 = T / 1e9
        Hubble = self.background.get_Hubble(T)
        rho_b = self.background.get_rho_b(T)
        dY = np.zeros_like(Y)  # differential for each species following the shape of Y,
        # e.i. dY[0] corresponds to dY_n. Initialized to zero for all species.

        # Weak interactions, always included ~ (n <-> p) (a 1-3)
        Y_n, Y_p = Y[0], Y[1]
        lambda_n, lambda_p = self.RR.get_rate_weak(T9)
        # The change to particles on the left hand side:
        change_LHS = Y_p * lambda_p - Y_n * lambda_n
        dY[0] += change_LHS  # Update the change to neutron fraction
        dY[1] -= change_LHS  # Update the change to proton fraction (opposite sign)

        if self.NR_species > 2:  # Include deuterium
            Y_D = Y[2]
            # (n+p <-> D+gamma) (b.1)
            Y_np = Y_n * Y_p
```

```python
            rate_np, rate_D = self.RR.get_np_to_D(T9, rho_b)
            change_LHS = Y_D * rate_D - Y_np * rate_np  # left hand side changes
            dY[0] += change_LHS  # Update the change to neutron fraction
            dY[1] += change_LHS  # Update the change to proton fraction
            dY[2] -= change_LHS  # Update the change to deuterium fraction

        if self.NR_species > 3:  # Include tritium
            Y_T = Y[3]
            # (n+D <-> T+gamma) (b.3)
            Y_nD = Y_n * Y_D
            rate_nD, rate_T = self.RR.get_nD_to_T(T9, rho_b)
            change_LHS = Y_T * rate_T - Y_nD * rate_nD
            dY[0] += change_LHS
            dY[2] += change_LHS
            dY[3] -= change_LHS

            # (D+D <-> p + T) (b.8)
            Y_DD = Y_D * Y_D
            Y_pT = Y_p * Y_T
            rate_DD, rate_pT = self.RR.get_DD_to_pT(T9, rho_b)
            change_LHS = 2 * Y_pT * rate_pT - Y_DD * rate_DD
            change_RHS = 0.5 * Y_DD * rate_DD - Y_pT * rate_pT
            dY[2] += change_LHS
            dY[1] += change_RHS
            dY[3] += change_RHS

        if self.NR_species > 4:  # Include He3
            ...
        if self.NR_species > 5:  # Include He4
            ...
        if self.NR_species > 6:  # Include Li7
            ...
        if self.NR_species > 7:  # Include Be7
            ...

        # Each reaction equation above should be multiplied with (-1/Hubble) before return:
        return -dY / Hubble

    def get_IC(self, T_init: float):
        """
        Defines the initial condition array used in solve_BBN.
        The only nonzero values are for neutrons and protons, but the shape of self.Yinit is
        determined by the number of particles included.

        Arguments:
            T_init {float} -- the initial temperature
        """
        Y_init = np.zeros(self.NR_species)  # Initialize all species to zero
        Yn_init, Yp_init = self.get_np_equil(T_init)  # solves equations (16-17)
        Y_init[0] = Yn_init
        Y_init[1] = Yp_init

        return Y_init

    def solve_BBN(self, T_init: float = 100e9, T_end: float = 0.01e9):
        """
        Solves the BBN-system for a given range of temperature values

        Keyword Arguments:
            T_init {float} -- the initial temperature (default: {100e9})
            T_end {float} -- the final temperature (default: {0.01e9})
        """
        sol = solve_ivp(  # solve the ODE-system using scipy.solve_ivp
            self.get_ODE,
            [np.log(T_init), np.log(T_end)],  # our equations are defined over ln(T),
            y0=self.get_IC(T_init),
            method="Radau",
            rtol=1e-12,
            atol=1e-12,
```

```python
            dense_output=True,  # this allows us to extract the solvables following the
                                                    procedure below
        )
        # Define linearly spaced logarithmic temperatures using points from the solver:
        lnT = np.linspace(sol.t[0], sol.t[-1], self.NR_points)
        self.Y_i = sol.sol(lnT)  # use ln(T) to extract the solved solutions
        # Use ln(T) to create a corresponding logarithmically spaced T array
        self.T = np.exp(lnT)  # array used for plotting, points corresponds to self.Y


if __name__ == "__main__":
    # Example use:
    # Initiate the system including 3 species (neutrons, protons and deuterium):
    bbn = BBN(NR_interacting_species=3)
    bbn.solve_BBN(T_end=0.1e9)  # solve the system until end temperature 0.1*10^9 K

    # Plot the mass fraction for each species:
    fig, ax = plt.subplots()
    for i, y in enumerate(bbn.Y_i):
        ax.loglog(bbn.T, bbn.mass_number[i] * y, label=bbn.species_labels[i])
```