Documentation

Introduction

This program tries to solve numerically the transport equation in the diffusion approximation:

$$\frac{\partial}{\partial\tau}f(\tau,p)=\frac{\hat{q}}{4p^2}\frac{\partial}{\partial p}\left[p^2\left(\frac{\partial f}{\partial\tau}+\frac{1}{T^*}f(1+f)\right)\right] \tag{1}$$

With this purpose is useful to introduce the quantity

$$n[i]\equiv\int_{p_s[i-1]}^{p_s[i]}dpp^2f\approx\frac{1}{3}(p_s[i]^3-p_s[i-1]^3)f_i\ , \tag{2}$$

which measures the particle number in the volume with momentum in the interval $(p_s\left[i-1\right],p_s\left[i\right])$. Then, we can integrate (1) and rewrite it as

$$\frac{\partial}{\partial\tau}n(\tau,p)=\frac{\hat{q}}{4p^2}\left[\frac{\partial f}{\partial\tau}+\frac{1}{T^*}f(1+f)\right]\equiv\mathcal{B}(f)\ . \tag{3}$$

And this is trivially integrable via the Euler method

$$n[j]=n[j-1]+\Delta t\mathcal{B}(f)[j-1], \tag{4}$$

where the j index runs over the time discretization.

With this purpose, we present three different Python classes for achieve this computation. The first of them defines the lattice in which we will made our calculations. The second one takes this lattice and implements the evolution of the transport equation. The last one it is just a utility for construct an animation with the data of the time evolution. Let us see how works each of them.

`Lattice`

This class defines a lattice in the momentum space that we can use for perform our computations. It takes three **arguments** and use them to construct the lattice as a numpy array.

- `pmin` : Minimum value of the momentum.

- `pmax` : Maximum value of the momentum.
- `size` : Number of elements of the lattice

`Lattice` also implements a few **methods** in order to access to some useful elements related with the lattice.

- `get_lattice` : Returns the numpy array which contains the lattice.
- `get_p_lattice` : Returns the mean values for each interval of the lattice, which correspond with the momentum value of each volume.
- `number(i, f)` : Computes the number of particles in the volume of the phase space which momentum is in the interval (p[i-1], p[i]). It takes two arguments:
  - `i` : index of the lattice on which we want to compute the number of particles.
  - `f` : distribution function associated with the lattice.
- `energy(i, f)` : Computes the energy in the volume of the phase space which momentum is in the interval (p[i-1], p[i]). It takes two arguments:
  - `i` : index of the lattice on which we want to compute the number of particles.
  - `f` : distribution function associated with the lattice.
- `entropy(i, f)` : Computes the entropy in the volume of the phase space which momentum is in the interval (p[i-1], p[i]). It takes two arguments:
  - `i` : index of the lattice on which we want to compute the number of particles.
  - `f` : distribution function associated with the lattice.
- `save_lattice` : Save both `get_lattice` and `get_p_lattice` in each .txt file.

`Time_evolution`

This class is used to compute the time evolution of the transport equation in the lattice. It takes the following **arguments**:

- `deltat` : time step for the time evolution of the transport equation.
- `nt_steps` : number of steps computed in the time evolution.
- `pmin` : Minimum value of the momentum.
- `pmax` : Maximum value of the momentum.
- `size` : Number of elements of the lattice.

- `Qs` : Saturation momentum. As default it is set $Q_s = 0,1$.

`Time_evolution` implements the following **methods**.

- `initial_condition` : Set the initial values of the distribution with a simple model of a step-function with saturation momentum Q_s. It returns both the function distribution and the number density as numpy arrays for each interval of the lattice. Notice that this values are constructed as the correspondant for the mean momentum value of each inteval, so if the lattice is set to have n elements, the distribution function (and the number density) will have $n - 1$.
- `derivative` : Computes the value of the time derivative for the number distribution except for the last one for the current time step. Returns:
  - numpy array with the derivatives.
  - float with the I_a integral result.
  - float with the I_b integral result.
  - float with the $T^* = I_a/I_b$ result.
- `next_step` : Update the next time step distribution function and number density values. It returns:
  - numpy array with the new distribution function values.
  - numpy array with the new number density values.
  - list containing the values of I_a, I_b and T^* for the current time step.
- `evolve(save, plot)` : Compute the time evolution of the number distribution for the total number `nt_steps`. It has two variables:
  - `save` : number of iteration after which data is saved via the `save` method. If `False` (default), no data is saved.
  - `plot` : if `True`, distribution is plot after each time step. Uses the `plot_evo` method in order to follow the evolution in real time.
- `save(iter, additional)` : Saves the data of the current step in different text files. It takes two arguments:
  - `iter` : Current iteration of the transport equation evolution.
  - `additional` : As default it is set as `False` and will save only the distribution function and number density of each step. If not, it must be a list [I_a, I_b, T^*], and it will also save this values in `integrals.txt` and the total number, energy and entropy of the system in `stats.txt`.

- `conservation` : Test if the conservation of number and energy was achieved during the simulation. For this purpose, we print the difference between the maximum and minimum value of each quantity, and its quotient with respect the maximum value. This results are saved in `conservation.txt`.
- `plot_evo (iter)` : Plots the current distribution function and number density.

`Animate`

Utility that takes the save files of the `Time_evolution.save` method and creates an animation of the transport equation evolution by calling `Animation.animate()`.