pychangcooper Documentation

J. Michael Burgess

Dec 31, 2020

Contents

A simple numerical solver for Fokker-Planck style equations of the form:

$$\frac{\partial N\left(\gamma,t\right)}{\partial t}=\frac{\partial}{\partial\gamma}\left[B\left(\gamma,t\right)N\left(\gamma,t\right)+C\left(\gamma,t\right)\frac{\partial N\left(\gamma,t\right)}{\partial\gamma}\right]$$

designed with an object-oriented interface to allow for easy problem specification via subclassing.

Contents:

[1]:
```
import numpy as np

%matplotlib notebook
import matplotlib.pyplot as plt

plt.style.use("ggplot")

from pychangcooper import ChangCooper
```

CHAPTER 1

Specifying a problem

1.1 Defining heating and dispersion terms.

The ChangCooper class automatically specifies the appropriate difference scheme for any time-independent heating and acceleration terms.

[2]:
```
class MySolver(ChangCooper):
    def __init__(self):

        # we have no injection, so we must have an
        # initial non-zero distribution function

        init_distribution = np.ones(100)

        # must pass up to the super class so that
        # all terms are setup after initialization

        super(MySolver, self).__init__(
            n_grid_points=100,
            delta_t=1.0,
            max_grid=1e5,
            initial_distribution=init_distribution,
            store_progress=True,  # store each time step
        )

    def _define_terms(self):

        # energy dependent heating and dispersion terms
        # must be evaluated at half grid points.

        # These half grid points are automatically
        # calculated about object creation.

        self._heating_term = self._half_grid
```

(continues on next page)

(continued from previous page)

```
        self._dispersion_term = self._half_grid2
```

To run the solver, simply call the solution method. If the store_progress option has been set, then each solution is stored in the objects history.

[3]:
```
solver = MySolver()

# amount of time that has gone by
print(solver.current_time)

# number of
print(solver.n_iterations)

# current solution
print(solver.n)
```

```
0.0
0
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1.]
```

[4]:
```
solver.solve_time_step()


# amount of time that has gone by
print(solver.current_time)

# number of
print(solver.n_iterations)

# current solution
print(solver.n)
```

```
1.0
1
[10.28587875  9.38231525  8.65268405  8.06122973  7.57954714  7.18507042
  6.85987258  6.58971189  6.36327433  6.17157169  6.00746348  5.86527709
  5.74050603  5.62957022  5.52962553  5.43841252  5.35413619  5.27537057
  5.20098283  5.13007298  5.06192609  4.99597417  4.93176599  4.8689431
  4.80722071  4.7463726   4.68621905  4.62661726  4.56745381  4.5086386
  4.45010004  4.39178125  4.33363706  4.27563157  4.21773627  4.15992851
  4.1021903   4.04450733  3.98686825  3.92926404  3.87168753  3.81413302
  3.756596    3.69907286  3.64156076  3.58405742  3.52656104  3.4690702
  3.41158375  3.3541008   3.29662061  3.23914263  3.1816664   3.12419156
  3.06671783  3.00924497  2.95177281  2.89430121  2.83683004  2.77935922
  2.72188869  2.66441837  2.60694822  2.54947822  2.49200833  2.43453852
  2.37706879  2.31959911  2.26212947  2.20465987  2.1471903   2.08972075
  2.03225121  1.9747817   1.91731219  1.85984269  1.8023732   1.74490371
  1.68743423  1.62996475  1.57249527  1.5150258   1.45755632  1.40008685
  1.34261738  1.28514791  1.22767845  1.17020898  1.11273951  1.05527005
  0.99780058  0.94033111  0.88286165  0.82539218  0.76792271  0.71045325
  0.65298378  0.59551432  0.53804485  0.48057539]
```

We can plot the evolution of the solution if we have been storing it.

[5]:
```
for i in range(10):

    solver.solve_time_step()

solver.plot_evolution(alpha=0.8)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

1.2 Adding source and escape terms

The general Chang and Cooper scheme does not specify injection and esacpe. But we can easily add them on. In this case, the Fokker-Planck equation reads:

$$\frac{\partial N(\gamma,t)}{\partial t} = \frac{\partial}{\partial \gamma}\left[B(\gamma,t)\,N(\gamma,t) + C(\gamma,t)\,\frac{\partial N(\gamma,t)}{\partial \gamma}\right] - E(\gamma,t) + Q(\gamma,t)\,.$$

In order ot include these terms, we simply need to define a source and escape function which will be evaluated on the grid at each iteration of the solution.

[6]:
```
class MySolver(ChangCooper):
    def __init__(self):

        # must pass up to the super class so that
        # all terms are setup after initialization

        super(MySolver, self).__init__(
            n_grid_points=100,
            delta_t=1.0,  # the time step of the solution
            max_grid=1e5,
            initial_distribution=None,
            store_progress=True,
        )

    def _define_terms(self):

        # energy dependent heating and dispersion terms
        # must be evaluated at half grid points.

        # These half grid points are automatically
        # calculated about object creation.

        self._heating_term = self._half_grid

        self._dispersion_term = self._half_grid2

    def _source_function(self, gamma):

        # power law injection
        return gamma ** 2

    def _escape_function(self, gamma):
```

(continues on next page)

(continued from previous page)

```
        # constant, energy-independent escape term
        return 0.5 * np.ones_like(gamma)
```

Upon object creation, the source and escape terms are automatically evaluated.

[7]:
```
solver = MySolver()

for i in range(10):

    solver.solve_time_step()

solver.plot_evolution(alpha=0.8, cmap="winter")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
```

[1]:
```
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt
from jupyterthemes import jtplot

jtplot.style(context="talk", fscale=1.4, spines=False, gridlines="--")

from pychangcooper import CoolingAcceleration
```

CHAPTER 2

Generic Cooling and Acceleration

For a generic heating/cooling and acceleration problem, we rewrite the Fokker-Planck equation as:

$$\frac{\partial N(\gamma,t)}{\partial t}=\frac{\partial}{\partial\gamma}\left[\left(C(\gamma)-A(\gamma)\right)N(\gamma,t)+D(\gamma)\frac{\partial N(\gamma,t)}{\partial\gamma}\right]$$

where:

$$A(\gamma)=\frac{2}{\gamma}D(\gamma).$$

Here we specify $C(\gamma)=C_0\gamma^a$ and $D(\gamma)=\frac{1}{2t_{\rm acc}}\gamma^b$ where the acceleration time is $t_{\rm acc}$. The steady-state solution for this problem (given no injection or escape) is

$$N(\gamma)\propto\gamma^2\exp\left[\frac{2t_{\rm acc}C_0\left(\gamma^{b+1-a}\right)}{a-1-b}\right]$$

.

If we let $a=b=2$, an electron of energy γ will cool in a characteristic time $t_{\rm cool}(\gamma)=1/\left(C_0\gamma\right)$, thus when the cooling time is equal to the acceleration time, and electron will have an equilibrium energy $\gamma_{\rm e}=\frac{1}{t_{\rm acc}C_0}$.

2.1 Solving the equation

2.1.1 Setting an initial distribution

We will start with an initial flat electron distribution at low energy and let it evolve for $50\cdot t_{\rm acc}$.

[2]:
```
n_grid_points = 300

init_distribution = np.zeros(n_grid_points)


for i in range(30):

    init_distribution[i + 1] = 1.0
```

2.1.2 Create the solver

We set $C_0 = 1$ and $t_{\rm acc} = 10^{-4}$ and thus $\gamma_{\rm e} = 10^4$.

[3]:
```
generic_ca = CoolingAcceleration(
    n_grid_points=n_grid_points,
    C0=1.0,
    t_acc=1e-4,
    cooling_index=2.0,
    acceleration_index=2.0,
    initial_distribution=init_distribution,
    store_progress=True,
)
```

Run the solver:

[4]:
```
for i in range(45):

    generic_ca.solve_time_step()
```

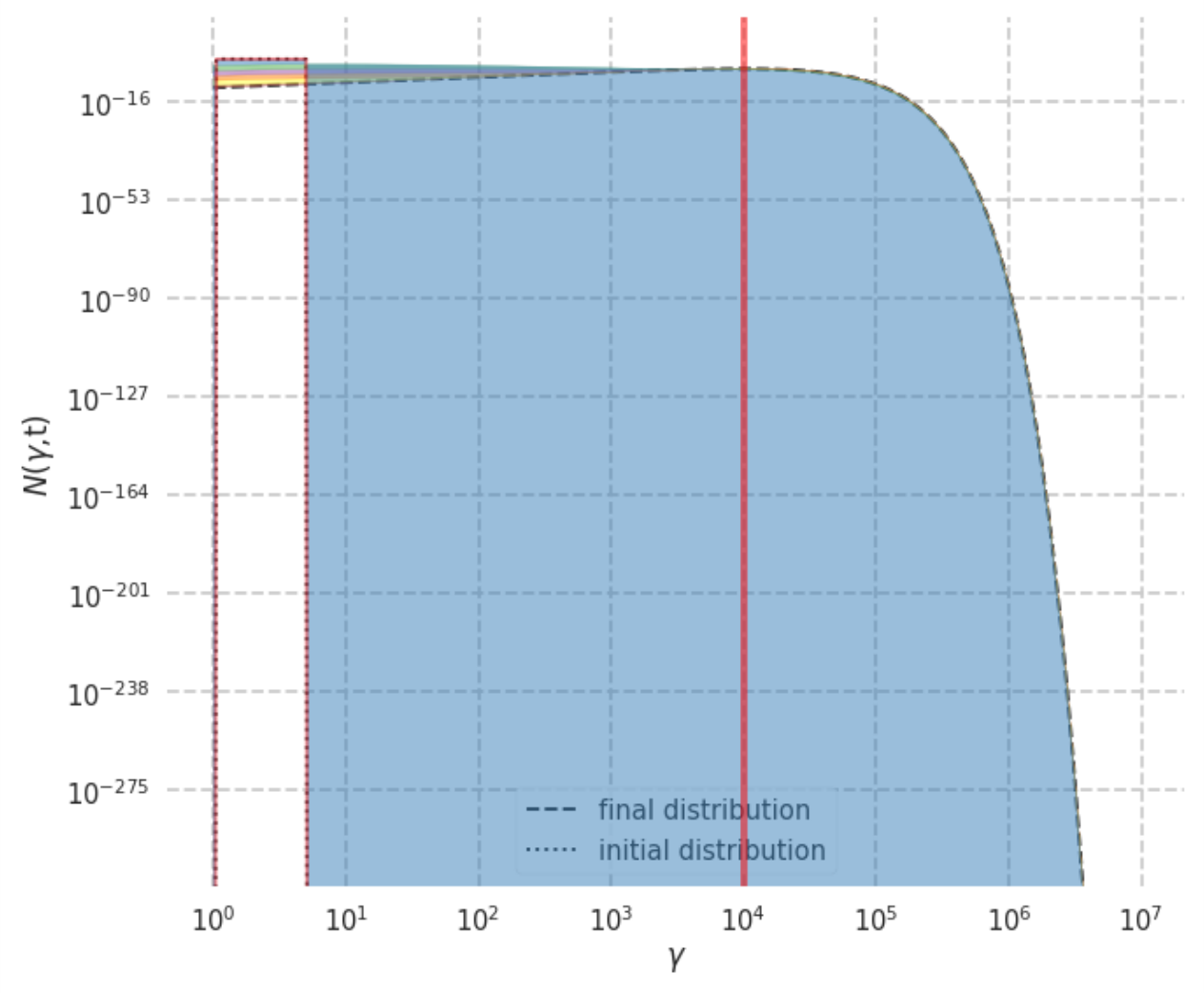
2.1.3 Plot the history of the solution

[5]:
```
fig = generic_ca.plot_evolution(
    skip=5, alpha=0.5, cmap="Set1", show_initial=True, show_final=True, show_
↪legend=True
)

ax = fig.get_axes()[0]

# plot the equilbrium solution
ax.axvline(1e4, color="red", lw=4, zorder=100, alpha=0.5)
```

[5]:
```
<matplotlib.lines.Line2D at 0x7f768b52d750>
```



[ ]:

[1]:
```
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt
from jupyterthemes import jtplot

jtplot.style(context="talk", fscale=1.4, spines=False, gridlines="--")

from pychangcooper import SynchrotronCooling_ContinuousPLInjection
```

CHAPTER 3

Synchrotron cooling

In the case of simple synchrotron cooling, the diffusion/dispersion term is zero. However, we will have a source term, $Q(\gamma, t)$. Thus we have the simple case of:

$$\frac{\partial N\left(\gamma, t\right)}{\partial t} = \frac{\partial}{\partial \gamma} B\left(\gamma, t\right) N\left(\gamma, t\right) + Q(\gamma, t)$$

The synchrotron cooling class computes the cooling time of the electrons and sets the time step of the solver to the cooling time of the highest energy electron which evolves the most.

3.1 Solution

3.1.1 Setup the solver

First we set the solver with the initial physical parameters.

[2]:
```
synch_cool = SynchrotronCooling_ContinuousPLInjection(
    B=1e10,
    index=-3.5,
    gamma_injection=1e3,
    gamma_cool=500,
    gamma_max=1e5,
    store_progress=True,
)
```

3.1.2 Solving

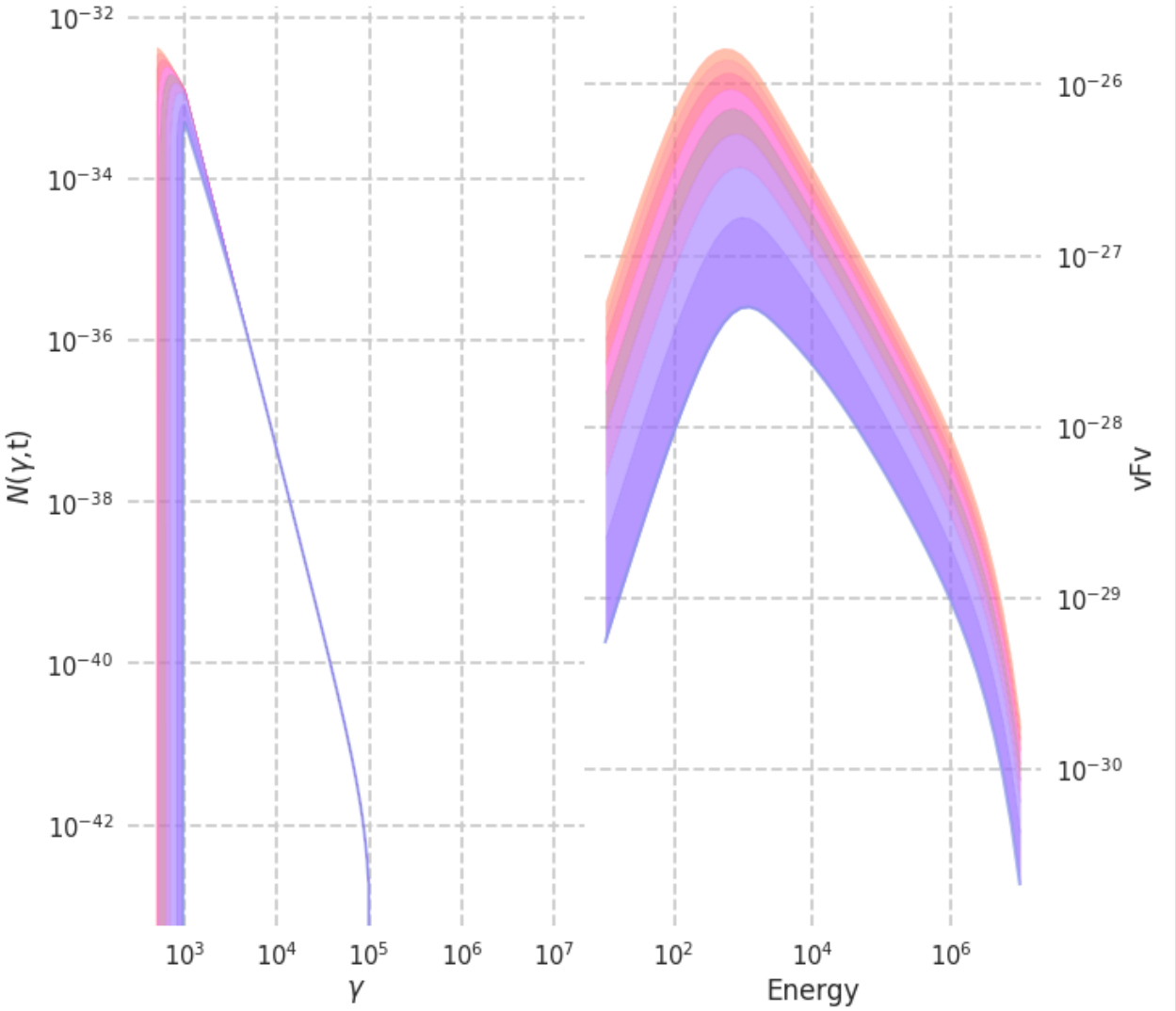
The solver automatically sets of the number of iterations such that the dynamical time equals the cooling time. If an array of photon energies are also passed, then the photon spectrum that is emitted by the electrons is also computed.

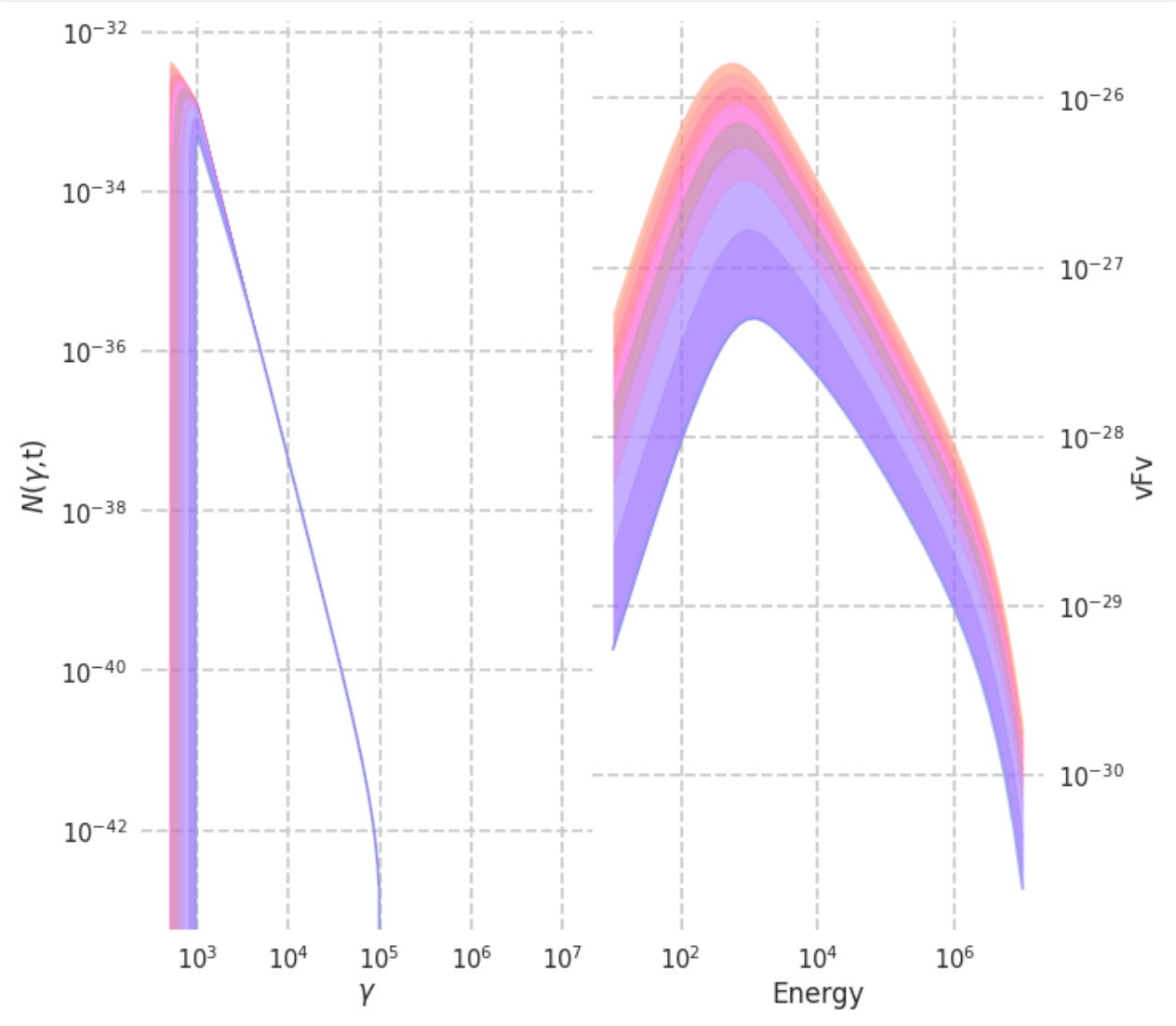
[3]:
```
synch_cool.run(photon_energies=np.logspace(1, 7, 50))
```

```
solving electrons electrons:   0%|          | 0/200 [00:00<?, ?it/s]
computing spectrum:   0%|          | 0/200 [00:00<?, ?it/s]
```

[4]:
```
synch_cool.plot_photons_and_electrons(skip=20, alpha=0.7, cmap="vaporwave")
```

[4]:




10^{-32}
10^{-34}
10^{-36}
10^{-38}
10^{-40}
10^{-42}
$N(\gamma,t)$
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
γ
10^{-26}
10^{-27}
10^{-28}
10^{-29}
10^{-30}
vFv
10^{2}
10^{4}
10^{6}
Energy

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

P