bolos Documentation

Release 0.1

Alejandro Luque

June 12, 2014

`BOLOS` is a BOLtzmann equation solver Open Source library.

This package provides a pure Python library for the solution of the Boltzmann equation for electrons in a non-thermal plasma. It builds upon previous work, mostly by G. J. M. Hagelaar and L. C. Pitchford [HP2005], who developed BOLSIG+. `BOLOS` is a multiplatform, open source implementation of a similar algorithm compatible with the BOLSIG+ cross-section input format.

The code was developed by Alejandro Luque at the Instituto de Astrofísica de Andalucía (IAA), CSIC and is released under the LGPLv2 License. Packages can be downloaded from the project homepage on PyPI. The source code can be obtained from GitHub, which also hosts the bug tracker. The documentation can be read on ReadTheDocs.

Contents:

CHAPTER 1

Tutorial

This tutorial will guide you through all the step that you must follow in order to use BOLOS in your code to solve the Boltzmann equation.

1.1 Installation

BOLOS is a pure Python package and it sticks to the Python conventions for the distribution of libraries. Its only dependencies are NumPy and SciPy. See here for installation instructions of these packages for your operating system.

There are a few ways to have BOLOS installed in your system:

1. Download the full source repo from github:

   ```
   git clone https://github.com/aluque/bolos.git
   ```

   This will create a *bolos* folder with the full code, examples and documentation source. You can then install bolos by e.g. typing:

   ```
   python setup.py install
   ```

   Alternatively since BOLOS is pure python package, you can put the *bolos/* sub-folder to whatever place where it can be found by the Python interpreter (including your *PYTHONPATH*).

2. You can use the Python Package Index (PyPI). From there you can download a tarball or you can instruct *pip* to download the package for you and install it in your system:

   ```
   pip install bolos
   ```

1.2 First steps

To start using bolos from your Python, import the required modules:

```
from bolos import parser, grid, solver
```

Usually you only need to import these three packages:

- *parser* contains methods to parse a file with cross-sections in BOLSIG+ format,
- *grid* allows you to define different types of grids in energy space.
- *solver* contains the `solver.BoltzmannSolver`, which the class that you will use to solve the Boltzmann equation.

Now you can define an energy grid where you want to evaluate the electron energies. The **:module:'grid'** contains a few classes to do this. The simplest one defines a linear grid. Let's create a grid extending from 0 to 20 eV with 200 cells:

```
gr = grid.LinearGrid(0, 60., 200)
```

We want to use this grid in a `solver.BoltzmannSolver` instance that we initialize as:

```
boltzmann = solver.BoltzmannSolver(gr)
```

1.3 Loading cross-sections

The next step is to load a set of cross-sections for the processes that will affect the electrons. BOLOS does not come with any set of cross-sections. You can obtain them from the great database LxCat. BOLOS can read without changes files downloaded from LxCat.

Now let's tell *boltzmann* to load a set of cross-sections from a file named *lxcat.dat*:

```
with open('lxcat.dat') as fp:
      processes = parser.parse(fp)
boltzmann.load_collisions(processes)
```

Do not worry if there are processes for species that you do not want to include: they will be ignored by BOLOS without a performance penalty.

1.4 Setting the conditions

Now we have to set the conditions in our plasma. First, we set the molar fractions; for example for synthetic air we do:

```
boltzmann.target['N2'].density = 0.8
boltzmann.target['O2'].density = 0.2
```

Note that this process requires that you have already loaded cross-sections for the targets that you are setting. Also, BOLOS does not check if the molar fractions add to 1: it is the user's responsibility to select reasonable molar fractions.

Next we set the gas temperature and the reduced electric field. BOLOS expect a reduced electric field in Vm^2 and a temperature in eV. However, you can use some predefined constants if you prefer to think in terms of Kelvin and Townsend. Here we set a temperature of 300K and a reduced electric field of 120 Td:

```
boltzmann.kT = 300 * solver.KB / solver.ELECTRONVOLT
boltzmann.EN = 120 * solver.TOWNSEND
```

After you set these conditions, you must tell BOLOS to update its internal state to take them into account. You must do this whenever you change kT, EN or the underlying grid:

```
boltzmann.init()
```

1.5 Obtaining the EEDF

We have now everything in place to solve the Boltzmann equation. Since the solver is iterative, we must start with some guess; it does not make much difference which one as long as it is not too unreasonable. For example, we can start with Maxwell-Boltzmann distribution with a temperature of 2 eV:

```
fMaxwell = boltzmann.maxwell(2.0)
```

Now we ask *boltzmann* to iterate the solution until it is satisfied that it has converged:

```
f = boltzmann.converge(fMaxwell, maxn=100, rtol=1e-5)
```

Here *maxn* is the maximum number of iterations and *rtol* is the desired tolerance.

We now have a distribution function in *f* that is a reasonable approximation to the exact solution. However, we made some arbitrary choices in order to calculate it and perhaps we may still get a more accurate one. For example, why did we select a grid from 0 to 60 eV with 200 cells? Perhaps we should base our grid on the mean energy of electrons:

```
# Calculate the mean energy according to the first EEDF
mean_energy = boltzmann.mean_energy(f0)

# Set a new grid extending up to 15 times the mean energy.
# Now we use a quadritic grid instead of a linear one.
newgrid = grid.QuadraticGrid(0, 15 * mean_energy, 200)

# Set the new grid and update the internal
boltzmann.grid = newgrid
boltzmann.init()

# Calculate an EEDF in the new grid by interpolating the old one
finterp = boltzmann.grid.interpolate(f, gr)

# Iterate until we have a new solution
f1 = boltzmann.converge(finterp, maxn=200, rtol=1e-5)
```

1.6 Calculating transport coefficients and reaction rates

Often you are not interested in the EEDF itself but you are working with a fluid clode and you want to know the transport coefficients and reaction rates as functions of temperature or E/n.

It's quite easy to obtain the reduced mobility and diffusion rate once you have the EEDF:

```
mun = boltzmann.mobility(f1)
diffn = boltzmann.diffusion(f1)
```

This tells you the reduced mobility *mu*n* and diffusion *D*n*, both in SI units.

To calculate reaction rates, use `solver.BoltzmannSolver.rate()`. There are a couple of manners in which you can specify the process. You can use its signature:

```
# Obtain the reaction rate for impact ionization of molecular nitrogen.
k = boltzmann.rate(f1, "N2 -> N2^+")
```

This is equivalent to the following sequence:

```
proc = boltzmann.search("N2 -> N2^+")[0]
k = boltzmann.rate(f1, proc)
```

Here we have first looked in the set of reactions contained in the *boltzmann* instance for a process matching the signature *"N2 -> N2^+"*. `solver.BoltzmannSolver.search()` returns a `process.Process` instance that you can then pass to `solver.BoltzmannSolver.rate()`.

The methods `solver.BoltzmannSolver.iter_all()`, `solver.BoltzmannSolver.iter_elastic()` and `solver.BoltzmannSolver.iter_inelastic()` let you iterate over the targets and processes contained in a `solver.BoltzmannSolver` instance. (These are the processes that we loaded earlier with `soler.BoltzmannSolver.load_collisions()`)

```
for target, proc in boltzmann.iter_inelastic():
    print "The rate of %s is %g" % (str(proc), boltzmann.rate(f1, proc))
```

1.7 Learn more

BOLOS is an ongoing effort and some of its features are not yet properly implemented or documented. If you want to learn more you can go to the *samples/* folder in the github repository. The code contains lots of (hopefully useful) comments, so if you want to understand better how to use or extend BOLOS, you should also read that.

CHAPTER 2

Frequently Asked Questions

2.1 Why another Boltzmann solver?

The low-temperature plasma community already has BOLSIG+, a highly optimized, user-friendly solver for the Boltzmann equation [HP2005]. BOLSIG+ is freely distributed by its authors, Hagelaar and Pitchford. Why did I write BOLOS, another Boltzmann solver based on similar algorithms?

The simplest reply is that, as a BOLSIG+ user, I wanted to understand better what goes on beneath BOLSIG+ and the best way to understand something is to do it yourself.

However, I also felt that an Open Source implementation would benefit the community. There are a number of drawbacks to the way BOLSIG+ is packaged that sometimes limited or slowed down my own research. For example, we only have a Windows version, whereas many of us now use Linux or Mac OS X as their platforms of choice. Also, since BOLSIG+ is distributed only as binary package, it is difficult or impossible to integrate into other codes or to make it part of an automated pipeline.

Finally, there is the old *hacker ethic*, where we tinker with each other's code and tools and collaborate to improve them. This is particularly relevant for scientists, since we all build on the work of others. Having an open source, modern, Boltzmann solver may facilitate new improvements and its integration with other tools.

2.2 Why did you use Python?

Because my main purpose was to develop a simple, readable code in the hope that other people would take it and perhaps improve it.

The code relies on the Numpy and SciPy libraries that interface with highly optimized, C or FORTRAN code.

2.3 What version(s) of Python does BOLOS support?

Presently, only 2.7. In future release, Python 3+ will be supported. Since BOLOS is a pure Python package, the transition should be straightforward.

2.4 Can BOLOS read cross-sections in BOLSIG+ format?

Yes! You can use your cross-sections files from BOLSIG+ or from LxCat without changes. Any problem reading these files will be treated as a bug.

2.5 How fast is BOLOS?

I would say it's `reasobaly fast`. It takes a few tenths of a second to solve the Boltzmann equation. The code was heavily optimized to use numpy's and scipy's features, particularly regarding sparse matrices.

2.6 Are results the same as with BOLSIG+?

In most cases the difference in reaction rates or transport parameters is between 0.1% and 1%. My guess is that most of the difference comes from the use of different grids but probably the growth-renormalization term is implemented differently (Hagelaar and Pitchford are not very clear on this point).

Here is a comparison between nitrogen ionization rates in synthetic air as calculated by BOLOS and BOLSIG+:

You can find the complete set of comparisons for synthetic air here.

2.7 Feature X is not implemented: what can I do?

Yes, there are still many features that are not implemented in BOLOS. In particular, only the temporal growth model is implemented and many parameters obtained from the EEDF are not yet implemented. I hope to add these things gradually. If you are interested in a particular feature you can give it a shot: pull requests are welcome. Or you can write me and I promise that I will look into it... but you know how tight all our agendas are.

2.8 If I use BOLOS for my research, which paper should I cite?

BOLOS follows the algorithm described by Hagelaar and Pitchford so you should definitely cite their paper [HP2005].

There is not yet any publication associated directly with BOLOS, so if you use it please link to its source code at github.

coefficients and rate coefficients for fluid models*, G. J. M. Hagelaar and L. C. Pitchford, Plasma Sources Sci. Technol. **14** (2005) 722–733.

CHAPTER 3

Bolos API reference

This page contains the documentation of the `BOLOS` API.

3.1 The `solver` Module

This module contains the main routines to load processes, specify the physical conditions and solve the Boltzmann equation.

The data and calculations are encapsulated into the `BoltzmannSolver` class, which you have to instantiate with a `grid.Grid` instance. Use `BoltzmannSolver.load_collisions()` or `BoltzmannSolver.add_process()` to add processes with their cross-sections. Afterwards, set the density of each component with `BoltzmannSolver.set_density()` or `BoltzmannSolver.target`. The method `BoltzmannSolver.maxwell()` gives you a reasonable initial guess for the electron energy distribution function (EEDF) that you can then improve iteratively with `BoltzmannSolver.converge()`. Finally, methods such as `BoltzmannSolver.rate()` or `BoltzmannSolver.mobility()` allow you to obtain reaction rates and transport parameters for a given EEDF.

class `bolos.solver.`**`BoltzmannSolver`**(*grid*)

Bases: `object`

Class to solve the Boltzmann equation for electrons in a gas.

This class contains the required elements to specify the conditions for the solver and obtain the equilibrium electron energy distribution function.

Parameters grid : `grid.Grid`

The grid in energies where the distribution funcition will be evaluated.

Examples

```
>>> import numpy as np
>>> from bolos import solver, grid
>>> grid.LinearGrid(0, 60., 400)
>>> bsolver = solver.BoltzmannSolver(grid)
>>> # Parse the cross-section file in BOSIG+ format and load it into the
>>> # solver.
>>> with open(args.input) as fp:
>>>     processes = parser.parse(fp)
>>> bsolver.load_collisions(processes)
>>>
```

```
>>> # Set the conditions.  And initialize the solver
>>> bsolver.target['N2'].density = 0.8
>>> bsolver.target['O2'].density = 0.2
>>> bsolver.kT = 300 * co.k / co.eV
>>> bsolver.EN = 300.0 * solver.TOWNSEND
>>> bsolver.init()
>>>
>>> # Start with Maxwell EEDF as initial guess.  Here we are starting with
>>> # with an electron temperature of 2 eV
>>> f0 = bsolver.maxwell(2.0)
>>>
>>> # Solve the Boltzmann equation with a tolerance rtol and maxn
>>> # iterations.
>>> f1 = bsolver.converge(f0, maxn=50, rtol=1e-5)
```

Attributes

benergy	(array of floats) Cell boundaries of the energy grid (set automatically at initialization). Equivalent to *grid.b*.
benergy	(array of floats) Cell lengths of the energy grid (set automatically at initialization). Equivalent to *grid.d*.
cenergy	(array of floats) Cell centers of the energy grid (set automatically at initialization). Equivalent to *grid.c*.
n	(int) Number of cells in the energy grid (set automatically at initialization). Equivalent to *grid.n*.
kT	(float) Gas temperature in eV. Must be set by the user.
EN	(float) Reduced electric field in Townsend (1 Td is 1e-21 V m^2). Must be set by the user.
target	(dict) A dictionary with targets in the set of processes. The user needs to set the density (molar fraction) of the desired targets using this dictionary. E.g. synthetic air is represented by

Methods

`add_process`(**kwargs)	Adds a new process to the solver.
`converge`(f0[, maxn, rtol, delta0, m, full])	Iterates and attempted EEDF until convergence is reached.
`diffusion`(F0)	Calculates the diffusion coefficient from a distribution function.
`init`()	Initializes the solver with given conditions and densities of the target species.
`iter_all`()	Iterates over all processes.
`iter_elastic`()	Iterates over all elastic processes.
`iter_growth`()	Iterates over all processes that affect the growth of electron density, i.e.
`iter_inelastic`()	Iterates over all inelastic processes.
`iter_momentum`()	
`iterate`(f0[, delta])	Iterates once the EEDF.
`load_collisions`(dict_processes)	Loads the set of collisions from the list of processes.
`maxwell`(kT)	Calculates a Maxwell-Boltzmann distribution function.
`mean_energy`(F0)	Calculates the mean energy from a distribution function.
`mobility`(F0)	Calculates the reduced mobility (mobility * N) from the EEDF.
`rate`(F0, k[, weighted])	Calculates the rate of a process from a (usually converged) EEDF.
`search`(signature[, product, first])	Search for a process or a number of processes within the solver.
`set_density`(species, density)	Sets the molar fraction of a species.

add_process(***kwargs*)
Adds a new process to the solver.

Adds a new process to the solver. The process data is passed with keyword arguments.

Parameters **type** : string

one of "EFFECTIVE", "MOMENTUM", "EXCITATION", "IONIZATION" or "ATTACHMENT".

target : string

the target species of the process (e.g. "O", "O2"...).

ratio : float

the ratio of the electron mass to the mass of the target (for elastic/momentum reactions only).

threshold : float

the energy threshold of the process in eV (only for inelastic reactions).

data : array or array-like

cross-section of the process array with two columns: column 0 must contain energies in eV, column 1 contains the cross-section in square meters for each of these energies.

Returns **process** : `process.Process`

The process that has been added.

See also:

`load_collisions` Add a set of collisions.

Examples

```
>>> import numpy as np
>>> from bolos import solver, grid
>>> grid.LinearGrid(0, 60., 400)
>>> solver = BoltzmannSolver(grid)
>>> # This is an example cross-section that decays exponentially
>>> energy = np.linspace(0, 10)
>>> cross_section = 1e-20 * np.exp(-energy)
>>> solver.add_process(type="EXCITATION", target="Kriptonite",
>>>                    ratio=1e-5, threshold=10,
>>>                    data=np.c_[energy, cross_section])
```

`converge` (*f0*, *maxn=100*, *rtol=1e-05*, *delta0=100000000000000.0*, *m=4.0*, *full=False*, ***kwargs*)
Iterates and attempted EEDF until convergence is reached.

Parameters **f0** : array of floats

Initial EEDF.

maxn : int

Maximum number of iteration until the convergence is declared as failed (default: 100).

rtol : float

Target tolerance for the convergence. The iteration is stopped when the difference between EEDFs is smaller than rtol in L1 norm (default: 1e-5).

delta0 : float

Initial value of the iteration parameter. This parameter is adapted in succesive iterations to improve convergence. (default: 1e14)

m : float

Attempted reduction in the error for each iteration. The Richardson extrapolation attempts to reduce the error by a factor m in each iteration. Larger m means faster convergence but also possible instabilities and non-decreasing errors. (default: 4)

full : boolean

If true returns convergence information besides the EEDF.

Returns **f1** : array of floats

Final EEDF

iters : int (returned only if `full` is True)

Number of iterations required to reach convergence.

err : float (returned only if `full` is True)

Final error estimation of the EEDF (must me smaller than `rtol`).

Notes

If convergence is not achieved after `maxn` iterations, an exception of type `ConvergenceError` is raised.

`diffusion`(*F0*)

Calculates the diffusion coefficient from a distribution function.

Parameters **F0** : array of floats

The EEDF used to compute the diffusion coefficient.

Returns **diffn** : float

The reduced diffusion coefficient of electrons in SI units..

See also:

`mobility` Find the reduced mobility from the EEDF.

`grid`

`init`()

Initializes the solver with given conditions and densities of the target species.

This method does all the work previous to the actual iterations. It has to be called whenever the densities, the gas temperature or the electric field are changed.

Notes

The most expensive calculations in this method are cached so they are not repeated in each call. Therefore the execution time may vary wildly in different calls. It takes very long whenever you change the solver's grid; therefore is is strongly recommended not to change the grid if is not strictly neccesary.

`iter_all`()

Iterates over all processes.

Returns An iterator over (target, process) tuples.

iter_elastic()
Iterates over all elastic processes.

Returns An iterator over (target, process) tuples.

iter_growth()
Iterates over all processes that affect the growth of electron density, i.e. ionization and attachment.

Returns An iterator over (target, process) tuples.

iter_inelastic()
Iterates over all inelastic processes.

Returns An iterator over (target, process) tuples.

iter_momentum()

iterate(*f0*, *delta=100000000000000.0*)
Iterates once the EEDF.

Parameters **f0** : array of floats

The previous EEDF

delta : float

The convergence parameter. Generally a larger delta leads to faster convergence but a too large value may lead to instabilities or slower convergence.

Returns **f1** : array of floats

A new value of the distribution function.

Notes

This is a low-level routine not intended for normal uses. The standard entry point for the iterative solution of the EEDF is the `BoltzmannSolver.converge()` method.

load_collisions(*dict_processes*)
Loads the set of collisions from the list of processes.

Loads a list of dictionaries containing processes.

Parameters **dict_processes** : List of dictionary or dictionary-like elements.

The processes to add to this solver class. See **:method:‘solver.add_process‘** for the required fields of each of the dictionaries.

Returns **processes** : list

A list of all added processes, as `process.Process` instances.

See also:

add_process Add a single process, with its cross-sections, to this solver.

maxwell(*kT*)
Calculates a Maxwell-Boltzmann distribution function.

Parameters **kT** : float

The electron temperature in eV.

Returns **f** : array of floats

A normalized Boltzmann-Maxwell EEDF with the given temperature.

Notes

This is often useful to give a starting value for the EEDF.

`mean_energy`(*F0*)

Calculates the mean energy from a distribution function.

Parameters **F0** : array of floats

The EEDF used to compute the diffusion coefficient.

Returns **energy** : float

The mean energy of electrons in the EEDF.

`mobility`(*F0*)

Calculates the reduced mobility (mobility * N) from the EEDF.

Parameters **F0** : array of floats

The EEDF used to compute the mobility.

Returns **mun** : float

The reduced mobility (mu * n) of the electrons in SI units (V / m / s).

See also:

`diffusion` Find the reduced diffusion rate from the EEDF.

Examples

```
>>> mun = bsolver.mobility(F0)
```

`rate`(*F0*, *k*, *weighted=False*)

Calculates the rate of a process from a (usually converged) EEDF.

Parameters **F0** : array of floats

Distribution function.

k : `process.Process` or string

The process whose rate we want to calculate. If *k* is a string, it is passed to `search()` to obtain a process instance.

weighted : boolean, optional

If true, the rate is multiplied by the density of the target.

Returns **rate** : float

The rate of the given process according to *F0*.

See also:

`search` Find a process that matches a given signature.

Examples

```
>>> k_ionization = bsolver.rate(F0, "N2 -> N2^+")
```

search (*signature*, *product=None*, *first=True*)
Search for a process or a number of processes within the solver.

Parameters **signature** : string

Signature of the process to search for. It must be in the form "TARGET -> RESULT [+ RESULT2]...".

product : string

If present, the first parameter is interpreted as TARGET and the second parameter is the PRODUCT.

first : boolean

If true returns only the first process matching the search; if false returns a list of them, even if there is only one result.

Returns **processes** : list or `process.Process` instance.

If `first` was true, returns the first process matching the search. Otherwise returns a (possibly empty) list of matches.

Examples

```
>>> ionization = solver.search("N2 -> N2^+")[0]
>>> ionization = solver.search("N2", "N2^+", first=True)
```

set_density (*species*, *density*)
Sets the molar fraction of a species.

Parameters **species** : str

The species whose density you want to set.

density : float

New value of the density.

Examples

These are two equivalent ways to set densities for synthetic air:

Using `set_density()`:

```
bsolver.set_density('N2', 0.8)
bsolver.set_density('O2', 0.2)
```

Using *bsolver.target*:

```
bsolver.target['N2'].density = 0.8
bsolver.target['O2'].density = 0.2
```

exception `bolos.solver.`**`ConvergenceError`**
Bases: `exceptions.Exception`

3.2 The `grid` Module

Routines to handle different kinds of grids (linear, quadratic, logarithmic)

class `bolos.grid.`**`AutomaticGrid`**(*grid*, *f0*, *delta=0.0001*)

Bases: `bolos.grid.Grid`

A grid set automatically using a previous estimation of the EEDF to fix a peak energy.

Methods

`cell`(x)	Returns the cell index containing the value x.
`interpolate`(f, other)	Interpolates into this grid an eedf defined in another grid.

class `bolos.grid.`**`GeometricGrid`**(*x0*, *x1*, *n*, *r=1.1*)

Bases: `bolos.grid.Grid`

A grid with geometrically progressing spacing. To be more precise, here the length of cell i+1 is r times the length of cell i.

Methods

`cell`(x)	Returns the cell index containing the value x.
`f`(x)	
`finv`(w)	
`interpolate`(f, other)	Interpolates into this grid an eedf defined in another grid.

`f`(*x*)

`finv`(*w*)

class `bolos.grid.`**`Grid`**(*x0*, *x1*, *n*)

Bases: `object`

Class to define energy grids.

This class encapsulates the information about an energy grid.

Parameters **x0** : float

Lowest boundary energy.

x1 : float

Highest energy boundary.

n : float

Number of cells

See also:

`LinearGrid` A grid with linear spacings (constant cell length).

`QuadraticGrid` A grid with quadratic spacings (linearly increasing cell length).

`GeometricGrid` A grid with geometrically increasing cell lengths.

`LogGrid` A logarithmic grid.

Notes

This is a base class and you usually do not want to instantiate it directly. You can define new grid classes by subclassing this class and then defining an *f* method that maps energy to a new variable *y* that is divided uniformly.

Methods

`cell`(x)	Returns the cell index containing the value x.
`interpolate`(f, other)	Interpolates into this grid an eedf defined in another grid.

`cell` (*x*)
Returns the cell index containing the value x.

Parameters **x** : float

The value x which you want to localize.

Returns **index** : int

The index to the cell containing x

`interpolate` (*f*, *other*)
Interpolates into this grid an eedf defined in another grid.

Parameters **f** : array or array-like

The original EEDF

other : `Grid`

The old grid, where *f* is defined.

Returns **fnew** : array or array-like

An EEDF defined in our grid.

class `bolos.grid.`**`LinearGrid`** (*x0*, *x1*, *n*)
Bases: `bolos.grid.Grid`

A grid with linear spacing.

Methods

`cell`(x)	Returns the cell index containing the value x.
`f`(x)	
`finv`(w)	
`interpolate`(f, other)	Interpolates into this grid an eedf defined in another grid.

`f` (*x*)

`finv` (*w*)

class `bolos.grid.`**`LogGrid`** (*x0*, *x1*, *n*, *s=10.0*)

Bases: `bolos.grid.Grid`

A pseudo-logarithmic grid. We add a certain s to the variable to avoid log(0) = -inf. The grid is actually logarithmic only for x >> s.

Methods

`cell`(x)	Returns the cell index containing the value x.
`f`(x)	
`finv`(w)	
`interpolate`(f, other)	Interpolates into this grid an eedf defined in another grid.

`f`(*x*)

`finv`(*w*)

class `bolos.grid.`**`QuadraticGrid`**(*x0*, *x1*, *n*)

Bases: `bolos.grid.Grid`

A grid with quadratic spacing.

Methods

`cell`(x)	Returns the cell index containing the value x.
`f`(x)	
`finv`(w)	
`interpolate`(f, other)	Interpolates into this grid an eedf defined in another grid.

`f`(*x*)

`finv`(*w*)

`bolos.grid.`**`mkgrid`**(*kind*, **args*, ***kwargs*)

Builds and returns a grid of class kind. Possible values are ‘linear’, ‘lin’, ‘quadratic’, ‘quad’, ‘logarithmic’, ‘log’.

3.3 The `parser` Module

This module contains the code required to parse BOLSIG+-compatible files. To make the code re-usabe in other projects it is independent from the rest of the BOLOS code.

Most user would only use the method `parse()` in this module, which is documented below.

`bolos.parser.`**`parse`**(*fp*)

Parses a BOLSIG+ cross-sections file.

Parameters **fp** : file-like

A file object pointing to a Bolsig+-compatible cross-sections file.

Returns **processes** : list of dictionaries

A list with all processes, in dictionary form, included in the file.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

Bibliography

[HP2005] *Solving the Boltzmann equation to obtain electron transport

[HP2005] *Solving the Boltzmann equation to obtain electron transport coefficients and rate coefficients for fluid models*, G. J. M. Hagelaar and L. C. Pitchford, Plasma Sources Sci. Technol. **14** (2005) 722–733.