python Documentation

Release 86g

Knox Long, Christian Knigge, Stuart Sim, Nick Higginbottom, Jam

Jan 22, 2024

DOCUMENTATION

CHAPTER

ONE

RADIATIVE TRANSFER AND IONISATION CODE

Python is a Monte-Carlo radiative transfer code designed to simulate the spectrum of biconical (or spherical) winds in disk systems. It was origianally written by Long and Knigge (2002) and was intended for simulating the spectra of winds in cataclysmic variables. Since then, it has also been used to simulate the spectra of systems ranging from young stellar objects to AGN.

The name Python is today unfortunate, and changing the name is an ongoing debate within the development team. The program is written in C and can be compiled on systems runining various flavors of linux, including OSX on Macs.

The code is is available on github Issues regarding the code and suggestions for improvement the code regarding the should be reported there. We actively encourage other to make use of the code for their own science. If anyone has questions about whether the code might be useful for a project, we encourage you to contact one of the authors of the code.

CHAPTER

TWO

DOCUMENTATION

Various documentation exists:

- A *Quick Guide* describing how to install and run Python (in a fairly mechanistic fashion).

For more information on how this page was generated and how to create documentation for *python*, look at the page for *documentation on the documentation*.

CHAPTER

THREE

AUTHORS

The authors of the *python* code and their institutions are:

Knox Long
Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218, USA Eureka Scientific, Inc., 2452 Delmer St., Suite 100, Oakland, CA 94602-3017, USA

Christian Knigge
Department of Physics and Astronomy, University of Southampton, Southampton, SO17 1BJ, UK

Stuart Sim
School of Mathematics and Physics, Queen's University Belfast, University Road, Belfast, BT7 1NN, UK

Nick Higginbottom
Department of Physics and Astronomy, University of Southampton, Southampton, SO17 1BJ, UK

James Matthews
Institute of Astronomy, University of Cambridge, Cambridge, CB3 0HA, UK

Sam Mangham
Department of Physics and Astronomy, University of Southampton, Southampton, SO17 1BJ, UK

Edward Parkinson
Department of Physics and Astronomy, University of Southampton, Southampton, SO17 1BJ, UK

Mandy Hewitt
School of Mathematics and Physics, Queen's University Belfast, University Road, Belfast, BT7 1NN, UK

Nicolas Scepi
Univ. Grenoble Alpes, CNRS, IPAG, 38000 Grenoble, France

3.1 *Quick Guide to Python*

This quide is intended to allow users to install Python, to run Python as a computer program and then to check whether the run has completed as expected.

It does not describe (except in passing) any information about the physics of Python, the details of a particular wind model, or criteria for evaluating whether the inputs correspond to a plausible model of an astrophysical system.

- *Installation* – how to install Python from github and to run a model
- *Creating the input file for Python* – Simple instructions how to set up a model interactively
- *The files produced by Python* – A quick look at the output files
- *Evaluation of the results* – A discussion of whether a model has run as required, or not

3.2 Getting Started

What machines will python run on? We have run python various versions of linux and on Mac. It is compiled using mpicc, with an option to compile with gcc.

It uses the Gnu Scientific Libraries (gsl)

(Developers should also have cproto in their path in order to create new prototypes, and access to indent to insure that routines are formatted in a standard fashion. They will also want to make sure the py_progs routines are properly installed, as indicated below).

3.2.1 Installation

Python and the various routines associated are set up in a self-contained directory structure. The basic directory structure and the data files that one needs to run Python need to be retrieved and compiled.

If you want to obtain a stable (!) release, go to the Releases page.

If you want to download the latest dev version, you can zip up the git repository by clicking on the zip icon to the right of the GitHub page. Alternatively, clone it directly as

```
$ git clone https://github.com/agnwinds/python.git
```

If you anticipate contributing to development we suggest Forking the repository and submitting pull requests with any proposed changes.

Once you have the files, you need to cd to the new directory and set your environment variables

```
$ export PYTHON = /path/to/python/
$ cd $PYTHON
$ ./configure
$ make install
$ make clean
```

One can run a more rigorous clean of GSL with `make distclean`, or remove the compiled GSL libraries altogether with `make rm_lib`.

note that export syntax is for bash- for csh use

```
$ setenv PYTHON /path/to/python/
```

The atomic data needed to run Python is included in the distribution.

(Python is updated fairly ofen. Normally, one does not need to redo the entire installation proces. Insstead follow the instuctions in updating)

3.2.2 Running python

To run python you need to add the following to your $PATH variable:

```
$PYTHON/bin
```

You can then setup your symbolic links by running

```
$ Setup_Py_Dir
```

and run the code by typing, e.g.

```
$ py root.pf
```

Running in parallel mode

While Python can be run in single processor mode, it is generally more efficient to run on multiple processors. in multiprocessor mode, When multiprocessing is invoked, Python uses mulitple threads for photon transfer and in calcuation ionization equilibrium. As these comprise the bulk of the computational load the total time to run is roughly an inverse of the number of threads. Python uses MPI for parallel processing and so software libraries that implement this must be on the machine that is being used. For Macs, mpi can installed with HomeBrew or Fink. For linux machines, two common libraries are Open-MPI and MPICH If not already installed, one should install them.

With mpi installed (and after recompiling with mpicc, which is the default) one would simply run the above program with

```
$ mpirun -np 8  py root.pf
```

where -np followed by a number designates the number of threads assigned.

Auxiliary programs

There are two programs that are useful for extracting information about models

- windsave2table generates a series of astropy tables that can be used to inspect elements of the various models, including densities of specific ions
- py_wind is a mainly interactive routine that prints similar infomation to the screen.

The two files are run as follows

```
$ windsave2table root
$ py_wind root
```

Brief descriptions of command line options for running these routines can obtained using a -h switch

Python scripts

There are a number of python, the progamming language scripts, that can be used to plot results from a Python run. These are not particularly well documented and many have been developed for looking at various aspects of the code. A few may require python packages to be installed. However, a number are likely to be useful.

To make use of these scipts one should add

$PYTHON/py_progs both to the PATH and PYTHONPATH variables

One script that is particularly useful is run_check.py, which is run as follows

```
$run_check.py root
```

This should create an html file that contains a summary set of information about a run, with plots that indicate how much of the wind has converged as a function of cycle, which cells have converged at the end, what the electron and temperature structrue of the wind is, as well as quick plots of the spectra that were produced.

Directory structure

The python directory structure is fairly simple:

source
Location of source code

bin
Location of executables

docs
Location of documentation, including sphinx docs, doxygen, parameters and documentation for the python programs in py_progs.

data
Location for all datafiles. Files that are mainly for reference should be gzipped to save space. Such files are not recreated in

bin
The location of the executables. (It is a good idea to put this directory in your path)

software
This directory contains libraries which are used in in python that must be recompiled when creating an installation on a new machine, primarily Bill Pence's cfitsio package and the GNU scientific library gsl

py_progs
python programs for helping analyse the code. We recommend adding this directory to your PATH and PYTHON_PATH environment variables.

examples
A directory with a few examples of python runs. (Note that the input files will have changed and so one may not be able to run these examples without some changes in the input files.)

Please help by reporting bugs in installation

This can be done by submitting a bug under the Issues page

3.3 Running Python

The normal way to run Python is simply to enter

```
py xxx
```

where `xxx` is the root name of a parameter file. (The full name `xxx.pf` can also be entered).

However Python features a number of command line options which can be used to modify it's operation. These include the following:

-h Causes Python to print out a brief help message and quit. The help message principally describes the command line options

-i (or –dry-run)
Causes Python to read and verify the inputs, writing a clean version of the input file `xxx.pf` to the output file `xxx.out.pf`, and then stop. This option is useful for setting up a proper `.pf` file. (Often one will want to copy `xxx.out.pf` back to `xxx.pf` before proceeding.

-t time_max Limits a run of python to approximately time_max in sec. This switch is used in situations where one would like to check whether the routine is operating properly be continuing, or where one needs to checkpoint the program after a certain period of time (due for example to time limits placed on jobs in a Beowulf cluster). The time is checked at the end of ionization and spectral cycles, immediately after saving the binary files that describe a model, and so one needs to leave a cushion between time_max and the maximum time one wants the program to run

-r Restarts a run that has been interrupted or halted, by reading a the `xxx.windsave` and `xxx.specsave` file (if it exists). Note that very few values in the .pf file are read when this options is used, as most of the information there has already been utilized in setting up and executing the run. The main ones that can be changed are the numbers of cycles for either ionizaion or detailed spectral cycles. Parameters that will be ignored include those assoicated with the wavelength range and extraction angles of the detailed spectra. The way to make changes to the detailed spectra is usually to use the option of setting the System_type to previous, which will allow one to set all of the detailed spectral parameters anew.

-v n Changes the amount of information printed to the screen by Python during a run. The default is 4. Larger numbers increase this. Smaller numbers decrease it. The log files are not affected.

--rseed Causes Python to use a random number seed that is time-based, rather than fixed. In most cases, a fixed seed is preferred so that problems can be replicated, but if is repeating the same calculation multiple times, then one may want a random seed.

--rng Save or load the RNG state to file, to allow persistent RNG states between restarts

--version Causes Python to print out the version number and commit hash (and whether uncommitted files exist, and then stop.

-p n_steps Changes the number of photons generated during ionization cycles so that the number increases logarithmically to the maximum value. The number `n_steps` is optional, and specifies the number of decades over which the increase takes place.

3.3.1 Special switches

Python has a number of other switches that are not intended for the general user, but which may be useful in certain special cases. These include:

-d Enables a variety of specialized diagnostic inputs which have been implemented to help with solving various problems, and were regarded (by someone) as useful enough to maintain in the program. The user is then queried regarding which of these diagnostics to enable for a specific run. These diagnostic queries all start with @ (and can co-exist in the `.pf` file, with normal commands. These options accessible with this flag are described further in *Diag*.

-e n Where `n` is a number, changes the number of errors of a specific type that are allowed to occur before the program gives up. For a variety of reasons, errors are expected during Python runs. Most of these errors are harmless in the sense that they occur rarely. But if an error occurs too often, something is seriously and so Python halts at that point. The default is 10^5 (per thread).

-e_write n
Changes the number of times an error message of a specific type is written to a diagnostic file. When errors occur, a line describing the error is written to the diagnostic file the first `n` times the error occurs. After that

statistics are maintained as to the number of times the error occurred, but it is not printed to the diagnostic file. The default is 100 (per thread)

-classic Reverts to using v/c corrections for special relativity and eliminates work done to treat co-moving frames properly. This is for testing, and is likely to be removed in the not too distant future.

-srclassic Use Python with full special relativity for Doppler shits, etc., but do not include any co-moving frame effects.

-no-matrix-storage Do not store macro-atom transition matrices if using the macro-atom line transfer and the matrix matom_transition_mode.n

-ignore_partial_cells Ignore wind cells that are only partially filled by the wind (This is now the default)

-include_partial_cells Include wind cells that are only partially filled by the wind

3.4 Inputs

Todo: Fill in

3.4.1 Overview

Python uses a keyword based parameter file the specify a model. A portion of a parameter file (which must have the extension .pf) is as follows:

```
Wind.radiation(yes,no)                              yes
Wind.number_of_components                       1
Wind.type(SV,star,hydro,corona,kwd,homologous,yso,shell,imported)                    sv
Wind.coord_system(spherical,cylindrical,polar,cyl_var)          cylindrical
Wind.dim.in.x_or_r.direction                    30
Wind.dim.in.z_or_theta.direction                30
```

Each line begins with a keyword followed optionally by a comment in parentheses, and then a value, e.g

- **Keyword:** `Wind.type`
- **Comment:** `SV,star,hydro,corona,kwd,homologous,shell,imported`
- **Value:** `SV`

The comment generally specifies a set of valid choices or the units in which information is expected.

When a series of choices is presented, one does not need to enter the complete word, just enough to provide unique match to the choice.

One does not need to create a parameter file before running Python. Instead, assuming one is not working from a template parameter file, one simply invokes Python.

```
py my_new_model
```

or

```
py -i my_new_model
```

Python then queries the user for answers to a series of question, creating in the process a pf file, my_new_model.pf, that can be edited and used in future runs.

An example of a line presented to the user in interactive mode is:

```
Disk.mdot(msol/yr) (1e-08) :
```

There the number in the second set of parenthesis is a suggested value of the parameter. The user types in a new value and a carriage return, or, if the the suggested value seems appropriate, responds with a carriage return, in which case the suggested value will be used.

The `-i` switch above indicates that Python should accumulate all of the necessary inputs, write out the parameter file, and exit, which is useful if one is not completely sure what one wants.

Changes in the input files as the code evolves

Occassionally, new input variables will be introduced into Python. In this case, when one tries to run a parameter file created with a previous version of Python in single processor mode, the program will query the user for the parameters that are missing, and then attempt to run the program as normal.

If the original name of the parameter file was `test.pf`, the modified version of the parameter file will be written to `test.out.pf`, so one normally copies, in this case `test.out.pf` to `test.pf` to avoid having the reenter the variable by hand if one wishes to run the parameter file a second time.

A better approach, if one is aware a change to the inputs has been made, is to run the old parameter file with `-i` switch, copy the `test.out.pf` to `test.pf`, and then run the program normally.

Alternatively, if one heeds to modify a number of input files, once one knows what the change is, one can simply edit the .pf files directly.

(In multiprocessor mode, if the inputs have changed, the program will fail at the outset, requiring one to got through the process of runnning the program with the `-i` switch, copying the `test.out.pf` to `test.pf`, and then running normally.)

3.4.2 System Description

The first set of parameters which Python needs are information about the overall system

```
System_type(star,cv,bh,agn,previous)                    bh

### Parameters for the Central Object
Central_object.mass(msol)                              10
Central_object.radius(cm)                  8.85667e+06
Binary.mass_sec(msol)                                  15
Binary.period(hr)                                      72

### Parameters for the Disk (if there is one)
Disk.type(none,flat,vertically.extended)                  flat
Disk.radiation(yes,no)                                 yes
Disk.rad_type_to_make_wind(bb,models)                    bb
Disk.temperature.profile(standard,readin)             standard
Disk.mdot(msol/yr)                                  1e-6
Disk.radmax(cm)                                      1e13

### Parameters for Boundary Layer or the compact object in an X-ray Binary or AGN
```

(continues on next page)

(continued from previous page)

```
BH.radiation(yes,no)                                yes
BH.rad_type_to_make_wind(bb,models,power,cloudy,brems)                power
Boundary_layer.lum(ergs/s)                4.72063e+39
Boundary_layer.power_law_index                   -1.5
```

System_type is starting point, a basic classification of the type of object one is trying to model. This is used to guide further questions about the object and to set defaults.

Most of the other parameters are fairly self-explanatory, and are documented fully in the various Parameters entries.

3.4.3 Wind Model Parameters

Python allows for various types of models, which are defined by the following parameters. This page focuses on the actual parameters in the file, but further description of the wind models and instructions on how to import models can be found under *Wind Models*.

```
### Parameters describing the various winds or coronae in the system
Wind.radiation(yes,no)                              yes
Wind.number_of_components                     1
Wind.type(SV,star,hydro,corona,kwd,homologous,shell,imported)                    sv
Wind.coord_system(spherical,cylindrical,polar,cyl_var)          cylindrical
Wind.dim.in.x_or_r.direction                  30
Wind.dim.in.z_or_theta.direction              30
```

Wind.radiation (WHICH PROBABLY WILL BE MOVED) allows for wind not only to scatter and absorb photons, but also to emit them by various processes, bound-bound, free-free, and recombination. It is the default for simple radiative transfer.

Wind.number_of_components is usually 1, but can be greater if one wishes to construct a wind from a combination of several wind models, for example a fast flow near the poles of a system, and a slow for near the disk. If the number of components exceeds 1, then the remaining questions relating to the wind will be posed multiple times.

The wind models incorporated into Python currently are:

SV
: The Shlosman and Vitello parameterization of a bi-conical flow

Stellar_wind
: A fairly standard parameterization of a spherical outflow for a hot star

Hydro
: A special purpose mode used by us for importing models from Zeus and Pluto

Corona
: A simple model for a corona above the disk

KWD
: The Knigge Woods and Drew parameterization of a bi-conical flow

Homologous
: A homologous expansion law useful for simulating SNe

Shell
: A model of a thin shell useful for diagnostic studies

Imported
: A general purpose mode for importing a wind from an ascii file (see also *Python Script documentation*).

3.4.4 Parameters

Todo: Fill in

System_type

The parameter is provides the program with a broad overview of the type of system that will be simulated, and is used by Python to initialize certain variable, and to control what variables are asked for later.

Type
Enumerator

Values

star
System in which the central object is a star

cv
System with a secondary star, which can occult the central object and disk depending on phase

bh
System with a black hole binary

agn
AGN

previous
In this case, one is starting from a previous run with python, and one want to either continue the run or change some parameters associated with radiation sources

File
python.c

Child(ren)

- *Boundary_layer.radiation*
- *Wind.old_windfile*
- *Spectrum.orbit_phase*
- *Central_object.geometry_for_source*
- *Binary.mass_sec*
- *Central_object.temp*
- *Atomic_data*
- *Central_object.blackbody_temp*
- *Wind.number_of_components*
- *Central_object.luminosity*
- *Binary.period*

Central object

Todo: Fill in

Binary

Todo: Fill in

Binary.mass_sec

In binary systems the mass of the secondary. This is used along with the period to establish the Roche lobes, so that one can see the effects of eclipses on the system

Type
Double

Unit
M$odot$/year

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *System_type*: cv, bh

Binary.period

The perids of a binary system. Along with a mass, the binary period is used to define the Roche lobe of the system, which in turn can be used to see the effect of eclipses on the spectrum. Defining the system as a secondary also initializes the outer radius of the disk.

Type
Double

Unit
Hours

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *System_type*: cv, bh

Boundary_layer

Todo: Fill in

Boundary_layer.luminosity

The luminosity of the boundary layer.

Type
Double

Unit
ergs/s

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *Boundary_layer.rad_type_to_make_wind*: `models`, `power`
- *Boundary_layer.rad_type_in_final_spectrum*: `models`, `uniform`

Boundary_layer.power_law_cutoff

This is a low frequency cutoff for an AGN-style power law spectrum of a form $L_nu=Knu^alpha$, as applied to the boundary layer of a star. It prevents the power-law being applied to low frequencies and giving an odd SED.

Type
Double

Unit
Hz

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *Boundary_layer.rad_type_to_make_wind*: `power_law`

Boundary_layer.power_law_index

The exponent $alpha$ in a power law SED applied to an AGN-style power law source for a non-AGN system. central source of the form $L_nu=Knu^alpha$.

Type
Double

Values
Any - but sign is not assumed, so for negative index use a negative value

File
setup_star_bh.c

Parent(s)

- *Boundary_layer.rad_type_to_make_wind*: `power_law`

Boundary_layer.rad_type_in_final_spectrum

Determines the luminosity and SED of the boundary layer. The code can cause a source to radiate differently in the ionisation and spectral cycles. This variable allows a boundary layer source to radiate differently from *Boundary_layer.rad_type_to_make_wind* during the cycles used to calculate the output spectra. This can be

Type
Enumerator

Values

bb
Black-body radiation. The boundary layer radiates as a black-body source with surface luminosity set by its effective temperature (*Boundary_layer.temp*) and resulting in a total luminosity proportional to its surface area.

models
Radiate according to a model. Python can support tabulated models that output with a binned luminosity distribution depending on system properties like temperature and gravity. See *Input_spectra.model_file*. The total luminosity will be set by *Boundary_layer.luminosity*.

uniform
Available for *System_type* of `star` or `cv`. Photons are generated with a random, uniformly-distributed wavelength between *Spectrum.wavemin* and *Spectrum.wavemax*. Can in some cases substitute for a Kurcuz spectrum. This mode is only available when generating final spectra.

File
python.c

Parent(s)

- *Boundary_layer.radiation*: `True`

Child(ren)

- *Input_spectra.model_file*
- *Boundary_layer.luminosity*
- *Boundary_layer.temp*

Boundary_layer.rad_type_to_make_wind

Determines the luminosity and SED of the boundary layer. The code can cause a source to radiate differently in the ionisation and spectral cycles. This variable allows a boundary layer source to radiate differently from *Boundary_layer.rad_type_in_final_spectrum* during the cycles used to calculate the wind ionisation state and temperature.

Type
Enumerator

Values

bb
Black-body radiation. The boundary layer radiates as a black-body source with surface luminosity set by its effective temperature (*Boundary_layer.temp*) and resulting in a total luminosity proportional to its surface area.

models
Radiate according to a model. Python can support tabulated models that output with a binned luminosity distribution depending on system properties like temperature and gravity. See *Input_spectra.model_file*. The total luminosity will be set by *Boundary_layer.luminosity*.

power
Radiate following a power-law model as $L_nu=Knu^alpha$. The total luminosity will be set by *Boundary_layer.luminosity*.

File
setup_star_bh.c

Parent(s)

- *Boundary_layer.radiation*: `True`

Child(ren)

- *Boundary_layer.power_law_index*
- *Input_spectra.model_file*
- *Boundary_layer.luminosity*
- *Boundary_layer.power_law_cutoff*
- *Boundary_layer.temp*

Boundary_layer.radiation

Says whether the boundary layer will radiate.

Type
Boolean (yes/no)

File
setup_star_bh.c

Parent(s)

- *System_type*: `star`, `cv`

Child(ren)

- *Boundary_layer.rad_type_to_make_wind*
- *Boundary_layer.rad_type_in_final_spectrum*

Boundary_layer.temp

The temperature of the boundary layer when radiating as a black body.

Type
Double

Unit
Kelvin

Values
Greater than 0

File
setup.c

Parent(s)

- *Boundary_layer.rad_type_to_make_wind*: bb
- *Boundary_layer.rad_type_in_final_spectrum*: bb

Central_object

Todo: Fill in

Central_object.blackbody_temp

If the AGN/BH is radiating as a black body, what temperature should it radiate at?

Type
Double

Unit
Kelvin

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *System_type*: agn, bh
- *Central_object.rad_type_to_make_wind*: bb

Central_object.bremsstrahlung_alpha

The frequency exponent $alpha$ in bremstrahlung SED of the form $L_nu=nu^{alpha}e^{-hnu/kT}$

Type
Double

Values
Any - sign is not assumed so use negative if you want negative

File
setup_star_bh.c

Parent(s)

- *Central_object.rad_type_to_make_wind*: `brems`

Central_object.bremsstrahlung_temp

The temperature T in bremstrahlung SED of the form $L_nu=nu^{alpha}e^{-hnu/kT}$

Type
Double

Unit
K

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *Central_object.rad_type_to_make_wind*: `brems`

Central_object.cloudy.high_energy_break

This is a command to define a cloudy type broken power law SED - mainly used for testing the code against cloudy. This SED has hardwired frequency exponents of 2.5 below the low energy break and -2.0 above the high energy break. This parameter defines the energy of the high energy break.

Type
Double

Unit
eV

Values
Greater than *Central_object.cloudy.low_energy_break*

File
setup_star_bh.c

Parent(s)

- *Central_object.rad_type_to_make_wind*: `cloudy`

Central_object.cloudy.low_energy_break

This is a command to define a cloudy type broken power law SED - mainly used for testing the code against cloudy. This SED has hardwired frequency exponents of 2.5 below the low energy break and -2.0 above the high energy break. This parameter defines the energy of the low energy break.

Type
Double

Unit
eV

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *Central_object.rad_type_to_make_wind*: `cloudy`

Central_object.geometry_for_source

If the central source in an AGN/BH system is radiating, what geometry should it radiate from?
This is applicable even for black-body sources, where the *luminosity* depends on *Central_object.radius*.

Type
Enumerator

Values

lamp_post
The central source radiates from two point sources located on the system axis above and below the disk plane. Emission is completely isotropic.

sphere
The central source radiates from a spherical surface with radius *Central_object.radius*. Emission is cosine-weighted in the direction of the sphere normal at the point of emission. Photons that would be spawned in an extended disk (if *Disk.type* is *vertically.extended*) are re-generated.

File
setup_star_bh.c

Parent(s)

- *System_type*: `agn`, `bh`
- *Central_object.radiation*: `True`

Child(ren)

- *Central_object.lamp_post_height*

Central_object.lamp_post_height

The distance above and below the disk plane of the two point sources used in the lamp-post model.

Type
Double

Unit
Central_object.radius

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *Central_object.geometry_for_source*: `lamp_post`

Central_object.luminosity

The luminosity of a non-blackbody AGN central source. This is defined as the luminosity from 2-10keV.

Type
Double

Unit
ergs/s

Values
Greater than 0.

File
setup_star_bh.c

Parent(s)

- *System_type*: `agn`, `bh`
- *Central_object.rad_type_to_make_wind*: `brems`, `cloudy`, `model`, `power`

Central_object.mass

Mass of the central object. This is very important, affecting wind speeds, gravitational heating and such.

Type
Double

Unit
M$odot$

Values
Greater than 0

File
setup_star_bh.c

Central_object.power_law_cutoff

Adds a low-frequency cutoff to the power law spectrum. Whilst this is required for power-law emission modes, it's set globally and also used in *cloudy* broken-power-law emission modes!

Type
Double

Unit
Hz

Values
Greater than or equal to 0

File
setup_star_bh.c

Parent(s)

- *Central_object.rad_type_to_make_wind*: `power`

Central_object.power_law_index

The exponent $alpha$ in a power law SED applied to a power law source of the form $L_nu=Knu^alpha$.

Type
Double

Values
Greater than 0

File
setup_star_bh.c

Parent(s)

- *Central_object.rad_type_to_make_wind*: `cloudy`, `power`

Central_object.rad_type_in_final_spectrum

Determines the SED of the central object in the spectral cycles. The luminosity is set by the options for the ionisation cycles, however.

Type
Enumerator

Values

bb
Available for *System_type* of `star` or `cv`. Black-body radiation. The boundary layer radiates as a black-body source with surface luminosity set by its effective temperature (*Central_object.temp*) and resulting in a total luminosity proportional to its surface area.

models
Available for *System_type* of `star` or `cv`. Radiate according to a model. Python can support tabulated models that output with a binned luminosity distribution depending on system properties like temperature and gravity. See *Input_spectra.model_file*. The total luminosity will be set by *Central_object.luminosity*.

uniform
Available for *System_type* of `star` or `cv`. Photons are generated with a random, uniformly-distributed wavelength between *Spectrum.wavemin* and *Spectrum.wavemax*. Can in some cases substitute for a Kurcuz spectrum. This mode is only available when generating final spectra.

brems
Available for *System_type* of `agn` or `bh`. Central object radiates with SED of a brehmsstralung spectrum as $L_nu=nu^{alpha}e^{-hnu/kT}$. This was originally developed to allow comparison to spectra generated according to Blondin heating and cooling rates.

cloudy
Available for *System_type* of `agn` or `bh`. Central object radiates with a 'broken' power law, intended largely for testing purposes against Cloudy. The SED form is $L_nu=Knu^alpha$, but beyond the provided high and low energy breakpoints the luminosity falls off sharply.

power
Available for *System_type* of `agn` or `bh`. Radiate following a power-law model as $L_nu=Knu^alpha$. The total luminosity will be set by *Boundary_layer.luminosity*.

File
python.c

Parent(s)

- *Central_object.radiation*: `True`

Child(ren)

- *Input_spectra.model_file*

Central_object.rad_type_to_make_wind

Multi-line description, must keep indentation.

Type
Enumerator

Values

bb
Black-body radiation. The boundary layer radiates as a black-body source with surface luminosity set by its effective temperature (*Central_object.temp*) and resulting in a total luminosity proportional to its surface area.

models
Radiate according to a model. Python can support tabulated models that output with a binned luminosity distribution depending on system properties like temperature and gravity. See *Input_spectra.model_file*. The total luminosity will be set by *Central_object.luminosity*.

brems
Available for *System_type* of `agn` or `bh`. Central object radiates with SED of a brehmsstralung spectrum as $L_nu=nu^{alpha}e^{-hnu/kT}$. This was originally developed to allow comparison to spectra generated according to Blondin heating and cooling rates.

cloudy
Available for *System_type* of `agn` or `bh`. Central object radiates with a 'broken' power law, intended largely for testing purposes against Cloudy. The SED form is $L_nu=Knu^alpha$, but beyond the provided high and low energy breakpoints the luminosity falls off sharply.

power
Available for *System_type* of `agn` or `bh`. Radiate following a power-law model as $L_nu=Knu^alpha$. The total luminosity will be set by *Boundary_layer.luminosity*.

File
setup_star_bh.c

Parent(s)

- *Central_object.radiation*: `True`

Child(ren)

- *Central_object.power_law_cutoff*
- *Central_object.bremsstrahlung_alpha*
- *Central_object.cloudy.low_energy_break*
- *Central_object.bremsstrahlung_temp*
- *Central_object.blackbody_temp*
- *Input_spectra.model_file*
- *Central_object.cloudy.high_energy_break*
- *Central_object.luminosity*
- *Central_object.power_law_index*

Central_object.radiation

A booliean variable stating whether of not the central object should radiate. This will enable different follow-up questions depending on the system type.

Type
Boolean (yes/no)

File
setup_star_bh.c

Child(ren)

- *Central_object.geometry_for_source*
- *Central_object.rad_type_to_make_wind*
- *Central_object.rad_type_in_final_spectrum*

Central_object.radius

Radius of the central object in the system, e.g the white dwarf or black hole

For systems containing a WD the default radius is set by the mass-radius relation. For a BH, the default is set R(ISCO) for a non-rotating BH, that is to say $6R_g$.

Type
Double

Unit
cm

Values
Greater than 0

File
setup_star_bh.c

Central_object.temp

Temperature of the central star. Physically, this is used in blackbody radiation, shock heating and disk heating in YSO models. It is also used to help determine the frequency bands in which photons are emitted.

Type
Double

Unit
Kelvin

Values
Greater than zero

File
setup_star_bh.c

Parent(s)

- *System_type*: `star`, `cv`

Disk

Todo: Fill in

Disk.T_profile_file

When the user chooses to read in the temperature profile as a function of radius, the user is asked the name of the file that contains the desired profile.

Type
String

File
setup_disk.c

Parent(s)

- *Disk.temperature.profile*: readin

Disk.colour_correction

Type of colour correction to use

Type
Enumerator

Values

Done12
Temperature dependent form of colour correction from Done 2012 (see Disk)

File
setup_disk.c

Parent(s)

- *Disk.radiation*: `True`
- *Disk.type*: `flat`, `vertically.extended`
- *Disk.rad_type_to_make_wind*: `bb`, `models`, `mod_bb`

Disk.mdot

The mass transfer rate in the disk when considering a standard Shakura-disk.

Type
Double

Unit
M$odot$/year

File
setup_disk.c

Parent(s)

- *Disk.temperature.profile*: standard

Disk.rad_type_in_final_spectrum

Multi-line description, must keep indentation.

Type
Enumerator

Values

bb
Blackbody from each annulus

models
Use model files such as stellar atmosphers

mod_bb
modified blackbody (colour correction)

File
python.c

Parent(s)

- *Disk.radiation*: `True`

Child(ren)

- *Input_spectra.model_file*

Disk.rad_type_to_make_wind

Multi-line description, must keep indentation.

Type
Enumerator

Values

bb
Blackbody from each annulus

models
Use model files such as stellar atmosphers

mod_bb
modified blackbody (colour correction)

File
setup_disk.c

Parent(s)

- *Disk.radiation*: `True`
- *Disk.type*: `flat`, `vertically.extended`

Child(ren)

- *Input_spectra.model_file*
- *Disk.colour_correction*

Disk.radiation

Multi-line description, must keep indentation.

Type
Boolean(yes/no)

File
setup_disk.c

Parent(s)

- *Disk.type*: `flat`, `vertically.extended`

Child(ren)

- *Disk.rad_type_to_make_wind*
- *Disk.temperature.profile*
- *Disk.rad_type_in_final_spectrum*

Disk.radmax

The outer edge of the disk. Photons inside this radius are absorbed or re-radiated. Photons which are outside this radius pass through the disk plane.

Type
Double

Unit
cm

Values
Greater than 0

File
setup_disk.c

Parent(s)

- *Disk.type*: `flat`, `vertically.extended`

Disk.temperature.profile

The choice of disk temperature profile

Type
Enumerator

Values

standard
A Shakura - Sunyaev disk, with a hard inner boundary

readin
Read the profile in from a file; the user will be queried for the name of the file

File
setup_disk.c

Parent(s)

- *Disk.radiation*: `True`

Child(ren)

- *Disk.mdot*
- *Disk.T_profile_file*

Disk.type

Parameter defining whether there is a disk in the system

Type
Enumerator

Values

none
No disk

flat
Standard flat disk

vertically.extended
Vertically extended disk

File
setup_disk.c

Child(ren)

- *Disk.rad_type_to_make_wind*
- *Disk.radiation*
- *Disk.z1*
- *Disk.z0*

Disk.z0

Fractional height at maximum radius. The physical height at the outer disk will be this * *Disk.radmax*.

Type
Double

Values
Greater than 0

File
setup_disk.c

Parent(s)

- *Disk.type*: vertically.extended

Disk.z1

For a vertically extended the disk, the height of the disk is set to be *Disk.z0* * *Disk.radmax* * (r/*Disk.radmax*)**Disk.z1 where Disk.z1 is the power law index

Type
Double

Values
Greater than 0

File
setup_disk.c

Parent(s)

- *Disk.type*: vertically.extended

Wind

Todo: Fill in

Corona

Todo: Fill in

Corona.base_den

The coronal model is defined in terms of a base density and a scale height

Type
Double

Unit
number/cm**3

Values
Greater than 0

File
corona.c

Parent(s)

- *Wind.type*: corona

Corona.radmax

The corona is a box-shaped region which sits immediately above the disk. radmax defines the outer edge of the box.

Type
Double

Unit
cm

Values
Greater than *Central_object.radius*

File
corona.c

Parent(s)

- *Wind.type*: corona

Corona.radmin

The corona is a box-shaped region which sits immediately above the disk. radmin defines the inner edge of the box.

Type
Double

Unit
cm

Values
Greater than *Central_object.radius*

File
corona.c

Parent(s)

- *Wind.type*: corona

Corona.scale_height

The coronal model is defined in terms of a base density and a scale height

Type
Double

Unit
cm

Values
Greater than 0

File
corona.c

Parent(s)

- *Wind.type*: corona

Corona.vel_frac

For the coronal model, the azimuthal velocity is given by the velocity of the underlying disk. One can also give the corona a radial velocity, which is a fraction of the disk velocity. (As coded, if this number is positive, the velicty is the r direction is toward the central object).

Type
Double

Unit
Disk velocity

Values
Any, 0 implies no radial velocity.

File
corona.c

Parent(s)

- *Wind.type*: corona

Corona.zmax

The corona is a box-shaped region which sits immediately above the disk. zmax defines the height of the box.

Type
Double

Unit
cm

Values
Greater than that the radius of the central object

File
corona.c

Parent(s)

- *Wind.type*: corona

Homologous

Todo: Fill in

Homologous.boundary_mdot

The mass loss rate at the base of the wind in a homlogous flow model, a flow in which the velocity is proporional to the radius. In general, mdot will decline with radius, depending on the exponent of the power law that describes the trend in density.

Type
Double

Unit
M$odot$/yr

Values
Greater than 0

File
homologous.c

Parent(s)

- *Wind.type*: homologous

Homologous.density_exponent

The power law exponent which defines the decline in density of a homologous flow as a function of radious.

Type
Double

Values
Greater than 0 for a density that declines with radius

File
homologous.c

Parent(s)

- *Wind.type*: homologous

Homologous.radmax

Maximum extent of the homologous wind.

Type
Double

Unit
cm

Values
Greater than *Homologous.radmin*

File
homologous.c

Parent(s)

- *Wind.type*: homologous

Homologous.radmin

The starting point of for madel of a homologous flow, a model in which the velocity at any radius is proportional to the radius

Type
Double

Unit
cm

Values
Greater than or equal to *Central_object.radius*

File
homologous.c

Parent(s)

- *Wind.type*: homologous

Homologous.vbase

Velocity at the base of the wind

Type
Double

Unit
cm

Values
Greater than 0

File
homologous.c

Parent(s)

- *Wind.type*: homologous

Hydro

Todo: Fill in

Hydro.file

Relative path to a hydrodynamic snapshot file to be imported.

Type
String

File
hydro_import.c

Parent(s)

- *Wind.type*: hydro

Hydro.thetamax

The maximum theta value to be read in from a hydrodynamic snapshot. This is typically used to excise a dense disk from the midplane of such a snapshot. Use a negative value to tell the code to use all the data.

Type
Double

Unit
Degrees

Values

-1
use all data

X
use up to that angle (typically less than 90)

File
hydro_import.c

Parent(s)

- *Wind.type*: hydro

KWD

Todo: Fill in

KWD.acceleration_exponent

Sets the length scale over which the accleration to v_inf is accomplished. It is the value of the exponent beta for the Caster & Lamers equation of a stellar wind, v(r) = v_0 + (v_inf - v_0) * (1 - R_s/r) ** beta.

Type
Double

Values
Greater than 0

File
knigge.c

Parent(s)

- *Wind.type*: kwd

KWD.acceleration_length

The size of the acceleration length scale for a disk wind described by the KWD model.

Type
Double

Unit
cm

Values
Greater than 0

File
knigge.c

Parent(s)

- *Wind.type*: kwd

KWD.d

The ratio d/d_min is used to describe the degree of geometric collimation of the disk wind in the KWD model. However, d (the distance to the focal point in central object radii) is used as this provides a more natural parameter.

Type
Double

Unit
Central_object.radius

Values
Greater than 0

File
knigge.c

Parent(s)

- *Wind.type*: kwd

KWD.mdot_r_exponent

The exponent for the mass loss rate as defined in the KWD model, m_dot(r) = F(r) ** alpha = T(r) ** (4 * alpha). F is the local luminous flux and T is the local temperature at a radius R. A value of 0 sets a uniform mass loss rate.

Type
Double

Values
Greater than or equal to 0

File
knigge.c

Parent(s)

- *Wind.type*: kwd

KWD.rmax

The radius at which the disk wind terminates, in units of central object radii. This has to be greater than rmin.

Type
Double

Unit
Central_object.radius

Values
Greater than *KWD.rmin*

File
knigge.c

Parent(s)

- *Wind.type*: kwd

KWD.rmin

The radius at which the disk wind begins, in units of central object radii.

Type
Double

Unit
Central_object.radius

Values
Greater than 1

File
knigge.c

Parent(s)

- *Wind.type*: kwd

KWD.v_infinity

The velocity at large distances of a steller wind described by the KWD model, in units of escape velocity. Described in terms of Castor & Lamers equation, v(r) = v_0 + (v_inf - v_0) * (1 - R_s/r) ** beta.

Type
Double

Unit
Escape velocity

Values
Greater than 0

File
knigge.c

Parent(s)

- *Wind.type*: kwd

KWD.v_zero

Multiple of the local sound speed at the base of the wind, this results in the initial velocity of the wind being able to be greater or less than the local sound speed.

Type
Double

Unit
Sound speed at wind base

Values
Greater than 0

File
knigge.c

Parent(s)

- *Wind.type*: kwd

SV

Todo: Fill in

SV.acceleration_exponent

Power-law acceleration exponent (i.e. alpha) of a line driven wind in a Shlosman & Vitello (SV) CV disk wind model. Sets the length scale over which the accleration to v_inf is accomplished. This value is a constant; when equal to 1 the results resemble those of a linear velocity law. Typically for an SV type wind this power law exponent is 1.5. See equation (2) Shlosman & Vitello 1993, ApJ 409, 372.

Type
Double

Values
Greater than 0

File
sv.c

Parent(s)

- *Wind.type*: SV

SV.acceleration_length

The size of the acceleration length scale for a disk wind described by the Shlosman Vitelo model. See equation (2) Shlosman & Vitelo ApJ (1993),409,372

Type
Double

Unit
cm

Values
Greater than 0

File
sv.c

Parent(s)

- *Wind.type*: SV

SV.diskmax

The outermost radius from which the wind rises in a Shlossman-Vitello type disk wind. This radius is measured along the radial disk (r) direction i.e. zero describes the centre of the central object (white dwarf) See figure 1 of Shlosman & Vitello 1993, ApJ 409,372.

Type
Double

Unit
cm

Values
Greater than or equal to *SV.diskmin* (inner radius disk wind)

File
sv.c

Parent(s)

- *Wind.type*: SV

SV.diskmin

The innermost radius from which the wind rises in a Shlossman-Vitello type disk wind. This radius is measured along the radial disk (r) direction i.e. zero describes the centre of the central object (white dwarf) See figure 1 of Shlosman & Vitello 1993, ApJ 409,372.

Type
Double

Unit
cm

Values
Greater than or equal to *Central_object.radius*

File
sv.c

Parent(s)

- *Wind.type*: SV

SV.mdot_r_exponent

The exponent for the mass loss rate as defined in the Shlosman Vitelo model, See lambda in equation (4) Shlosman & Vitelo,ApJ,1993,409,372.

Type
Double

Values
Greater than or equal to 0. 0 sets a uniform mass loss rate.

File
sv.c

Parent(s)

- *Wind.type*: SV

SV.thetamax

The angle at which the wind rises from the outermost launching radius in a Shlossman-Vitello type disk wind. This angle is measured with respect to the vertical (z) direction i.e. zero describes a vertical wind. See figure 1 of Shlossman & Vitello 1993, ApJ 409,372.

Type
Double

Unit
Degrees

Values
Greater than sv.thetamin

File
sv.c

Parent(s)

- *Wind.type*: SV

SV.thetamin

The angle at which the wind rises from the innermost launching radius in a Shlossman-Vitello type disk wind. This angle is measured with respect to the vertical (z) direction. I.e. zero descirbes a vertical wind. See figure 1 of Shlossman & Vitello 1993, ApJ, 409, 372.

Type
Double

Unit
Degrees

Values
Greater than 0

File
sv.c

Parent(s)

- *Wind.type*: SV

SV.v_infinity

Asymptotic (i.e. final) velocity of a line driven wind in a Shlosman & Vitello CV disk wind model. Assumed to scale with the local velocity at the base of the streamline. See equation (2) Shlosman & Vitello 1993, ApJ 409, 372.

Type
Double

Unit
Escape velocity

Values
Greater than 0

File
sv.c

Parent(s)

- *Wind.type*: SV

SV.v_zero

The velocity at the wind base.

Type
Double

Unit
['Speed of sound in the wind', 'cm/s']

Values
Greater than 0

File
sv.c

Parent(s)

- *SV.v_zero_mode*: `sound_speed`, `fixed`

SV.v_zero_mode

Multi-line description, must keep indentation.

Type
Enumerator

Values

fixed
Multi-line description, must keep indentation.

sound_speed
Multi-line description, must keep indentation.

File
sv.c

Parent(s)

- *Wind.type*: SV

Child(ren)

- *SV.v_zero*

Shell

Todo: Fill in

Shell.wind.acceleration_exponent

Exponent beta for the Caster and Lamers description of a stellar wind v(r)=v_o + (v_inf - v_o) (1+R_s/r)**beta for a shell wind.

Type
Double

Values
Greater than or equal to 0

File
shell_wind.c

Parent(s)

- *Wind.type*: shell

Shell.wind.radmax

Multi-line description, must keep indentation.

Type
Double

Unit
cm

Values
Greater than *Shell.wind.radmin*

File
shell_wind.c

Parent(s)

- *Wind.type*: shell

Shell.wind.radmin

The innermost edge of a diagnostic type of wind made up of a single (ideally thin) shell.

Type
Double

Unit
cm

Values
Greater than 0

File
shell_wind.c

Parent(s)

- *Wind.type*: shell

Shell.wind.v_at_rmax

The velocity of a shell wind at the outer edge of the shell - the variation of the velocity in the shell is set by the velocity law exponent. It allows a gradient to be enforced.

Type
Double

Unit
cm/s

Values
Greater than or equal to 0

File
shell_wind.c

Parent(s)

- *Wind.type*: shell

Shell.wind_mdot

The mass loss through a diagnostic shell type wind. One normally sets this experimentally in order to get a required hydrogen density in the shell

Type
Double

Unit
M$odot$/year

Values
Greater than 0

File
shell_wind.c

Parent(s)

- *Wind.type*: shell

Shell.wind_v_at_rmin

The velocity of a shell wind at the inner edge of the shell - the variation of the velocity in the shell is set by the velocity law exponent. It allows a gradient to be enforced.

Type
Double

Unit
cm/s

Values
Greater than or equal to 0

File
shell_wind.c

Parent(s)

- *Wind.type*: shell

Stellar_wind

Todo: Fill in

Stellar_wind.acceleration_exponent

Exponent beta for the Caster and Lamers description of a stellar wind v(r)=v_o + (v_inf - v_o) (1+R_s/r)**beta

Type
Double

Values
Greater than or equal to 0

File
stellar_wind.c

Parent(s)

- *Wind.type*: star

Stellar_wind.mdot

Mass loss rate for a wind modelled in terms of the Caster and Lamemers prescription for a stellar wind.

Type
Double

Unit
M$odot$/year

Values
Greater than 0

File
stellar_wind.c

Parent(s)

- *Wind.type*: star

Stellar_wind.radmax

Multi-line description, must keep indentation.

Type
Double

Unit
cm

Values
Greater than or equal to *Stellar_wind.radmin*

File
stellar_wind.c

Parent(s)

- *Wind.type*: star

Stellar_wind.radmin

Inner edge in cm for a stellar wind, normally the radius of the star.

Type
Double

Unit
cm

Values
Greater than or equal to *Central_object.radius*

File
stellar_wind.c

Parent(s)

- *Wind.type*: star

Stellar_wind.v_infinity

The velocity at large distance of a stellar wind described in terms of the Casters and Larmers equation v(r)=v_o + (v_inf - v_o) (1+R_s/r)**beta

Type
Double

Unit
cm/s

Values
Greater than 0

File
stellar_wind.c

Parent(s)

- *Wind.type*: star

Stellar_wind.vbase

Multi-line description, must keep indentation.

Type
Double

Unit
cm/s

Values
Condition e.g. greater than 0 or list e.g. [1, 2, 5]

File
stellar_wind.c

Parent(s)

- *Wind.type*: star

Wind

Todo: Fill in

Wind.coord_system

The coordinate system used for a describing a component of the wind.

Type
Enumerator

Values

spherical
Spherical

cylindrical
Cylindrical

polar
Spherical polar

cyl_var
Cylindrical varying z

File
setup_domains.c

Parent(s)

- *Wind.number_of_components*: Greater than 0. Once per wind.

Wind.dim.in.x_or_r.direction

Winds are calulated on spherical, cylindrical, or polar grids. This input variable gives the size of the grid in the x or r direction. Because some grid cells are used as a buffer, the actual wind cells are contained in a slightly smaller grid than the number given.

Note that in some situations there may be more than one wind component, known technically as a domain. In that case the user will be queried for this value mulitple times, one for each domain

Type
Integer

Values
Greater than or equal to 4, to allow for boundaries.

File
setup_domains.c

Parent(s)

- *Wind.number_of_components*: Greater than or equal to 0. Once per wind.
- *Wind.type*: Not imported

Wind.dim.in.z_or_theta.direction

Winds are calulated on spherical, cylindrical, or polar grids. This input variable gives the size of the grid in the z or theta direction. Because some grid cells are used as a buffer, the actual wind cells are contained in a slightly smaller grid than the number given.

Note that in some situations there may be more than one wind component, known technically as a domain. In that case the user will be queried for this value mulitple times, one for each domain

Type
Integer

Values
Greater than 0

File
setup_domains.c

Parent(s)

- *Wind.number_of_components*: Greater than 0. Once per wind.
- *Wind.type*: Not imported

Wind.filling_factor

The volume filling factor of the outflow. The implementation of clumping (microclumping) is described in Matthews et al. (2016), 2016MNRAS.458..293M. Asked once per domain.

Type
Double

Values
0 < f <= 1, where 1 is a fully smooth wind.

File
setup_domains.c

Parent(s)

- *Wind.number_of_components*: Greater than 0. Once per domain.

Wind.fixed_concentrations_file

The filename for the fixed ion concentrations if you have set Wind_ionization to 2 (fixed). This file has format [atomic_number ionizationstage ion fraction].

And example of a fixed concentrations file is below:

```
1   1  0
1   2  1
```

In the example, the only element is H, and H is completely ionized.

Note that if one wants electron densities to agree with that expected from the ions in the in the fixed concentrations file, then for imported models, one make sure the the elements_ions data file only activates these elements.

Type
String

File
setup.c

Parent(s)

- *Wind.ionization*: fixed

Wind.ionization

The approach used by Python to calculate the ionization of the wind during ionization cycles. A number of these modes are historical or included for diagnostic purposes.

Type
Enumerator

Values

on.the.spot
Use a simple on the spot approximation to calculated the ionization.

LTE_te
Calculate ionization based on the Saha equation using the electron temperature. (This is intended as a diagnostic mode.)

LTE_tr
Calculate ionization based on the Saha equation using the radiation temperature. (This is intended as a diagnstic mode)

ML93
Use the modified on the spot approimation described by Mazzli & Lucy 1993

fixed
Read the ion aboundances in from a file. All cells will have the same abundances. (This is intended as a diagnostic mode, mainly to investigate the details of raditive transrfer. It should be used with caution. In particular, if the elements for which abundances are provided differ from the elements to be used as described in the elements/ions portion of the atomic data, then one should not expect the calculated electron density to be that that comes simply from the fixd concetnrations file.)

matrix_bb
Estimate photoionization rates by approximating the spectrum in each cell based on the radiation temperature and an effective weight. Invert the rate matrix equations to calculate the ionization

matrix_pow
Estimate photionization rates by approximating the spectrum in a cell by a piecewise approximation, usually a power law. Invert the rate matrix equation to calculate the ionization. (This is the preferred ionization mode for most calculations)

matrix_est
Estimate photionization rates by calculating rates directly from the photons that pass through a cell. There is no attempt to model the spectrum. Invert the rate matrix equation to calculate the ionization.

File
setup.c

Child(ren)

- *Wind.fixed_concentrations_file*

Wind.mdot

Multi-line description, must keep indentation.

Type
Double

Unit
M$odot$/year

Values
Greater than 0

File
['knigge.c', 'sv.c']

Parent(s)

- *Wind.type*: `knigge`, `SV`

Wind.model2import

The name of a file to containing a generic model to read in to python from an ascii file. (Note that this is not the same as reading in a model generated by python, but is intended to allow one to read in a generic model in a variety of formats with only a limited amount of information required).

Type
String

File
import.c

Parent(s)

- *Wind.type*: imported

Wind.number_of_components

While most simple description of a wind consist of a single region of space, Python can calculate radiative transfer through more complicated structres, where one region of space is described with one prescription and another region of space with a second prescription. For example, one might want to place a disk atmoosphere between the disk and a wind. This parameter describes the number of components (aka domains) of the wind.

Type
Integer

Values
Greater than 0

File
python.c

Parent(s)

- *System_type*: `star`, `binary`, `agn`

Child(ren)

- *Wind.t.init*
- *Wind.coord_system*
- *Diag.adjust_grid*
- *Wind.radmax*
- *Wind.filling_factor*
- *Wind.dim.in.z_or_theta.direction*
- *Wind.type*
- *Wind.dim.in.x_or_r.direction*

Wind.old_windfile

The rootname of a previously saved model in a calculation one wishes to continue (with the possiblity of making changes to some of the details of the radiation sources, or to extract spectra from different inclinations)

Type
String

File
python.c

Parent(s)

- *System_type*: previous

Wind.radiation

Whether or not the wind should radiate.

Type
Boolean (yes/no)

File
python.c

Wind.radmax

Multi-line description, must keep indentation.

Type
Double

Unit
cm

Values
Greater than *Central_object.radius* and any minimum wind radii in the system.

File
setup_domains.c

Parent(s)

- *Wind.number_of_components*: Greater than 0. Once per domain.

Wind.t.init

Starting temperature of the wind.

Type
Double

Unit
Kelvin

Values
Greater than 0

File
setup_domains.c

Parent(s)

- *Wind.number_of_components*: Greater than 0. Once per domain.

Wind.type

Multi-line description, must keep indentation.

Type
Enumerator

Values

SV
Multi-line description, must keep indentation.

corona
Multi-line description, must keep indentation.

homologous
Multi-line description, must keep indentation.

hydro
Multi-line description, must keep indentation.

imported
Multi-line description, must keep indentation.

kwd
Multi-line description, must keep indentation.

shell
Multi-line description, must keep indentation.

star
Multi-line description, must keep indentation.

yso
Multi-line description, must keep indentation.

File
setup_domains.c

Parent(s)

- *Wind.number_of_components*: Greater than 0. Once per domain.

Child(ren)

- *Shell.wind_v_at_rmin*
- *Corona.radmax*
- *Wind.mdot*
- *KWD.mdot_r_exponent*
- *Corona.base_den*
- *KWD.v_zero*
- *Stellar_wind.mdot*

- *Homologous.radmin*
- *KWD.acceleration_length*
- *Corona.radmin*
- *KWD.rmax*
- *Homologous.radmax*
- *SV.thetamax*
- *SV.acceleration_exponent*
- *Corona.zmax*
- *Corona.scale_height*
- *Homologous.density_exponent*
- *Hydro.thetamax*
- *Wind.dim.in.z_or_theta.direction*
- *SV.diskmin*
- *SV.diskmax*
- *SV.acceleration_length*
- *Hydro.file*
- *KWD.acceleration_exponent*
- *Corona.vel_frac*
- *Stellar_wind.radmin*
- *Shell.wind.radmax*
- *Stellar_wind.radmax*
- *Wind.model2import*
- *Homologous.vbase*
- *Homologous.boundary_mdot*
- *KWD.rmin*
- *Shell.wind_mdot*
- *SV.mdot_r_exponent*
- *KWD.d*
- *Shell.wind.v_at_rmax*
- *Stellar_wind.acceleration_exponent*
- *Stellar_wind.v_infinity*
- *Shell.wind.radmin*
- *Shell.wind.acceleration_exponent*
- *SV.v_zero_mode*
- *SV.v_infinity*
- *Stellar_wind.vbase*

- *Wind.dim.in.x_or_r.direction*
- *KWD.v_infinity*
- *SV.thetamin*

Radiative Transfer & Ionisation

Todo: Fill in

Atomic_data

Python uses an atomic data file, as found in the *agnwinds/data* repository. This is the relative path to the Atomic Data header file on disk. See *Atomic Data*

Type
String

File
setup_line_transfer.c

Parent(s)

- *System_type*: `AGN`, `binary`, `star`

Ionization_cycles

The number of ionization cycles to execute - these are cycles to determine the ionization and thermal state of the wind

Type
Integer

Values
Greater than 0

File
setup.c

Line_transfer

The way in which line transfer and scattering is dealt with in the code. Governs whether we adopt any approximations for radiative transfer, whether to use the indivisible packet and macro-atom machinery, and whether to use isotropic or anisotropic scattering.

Thermal trapping mode is recommended for non-macro atom runs, while thermal trapping macro-atom mode is recommended for macro-atom runs.

Type
Enumerator

Values

pure_abs
Pure absorption

The pure absortion approximation.

pure_scat
Pure scattering

The pure scattering approximation.

sing_scat
Single scattering

The single scattering approximation.

escape_prob
Escape probability

Resonance scattering and electron scattering is dealt with isotropically. free-free, compton and bound-free opacities attenuate the weight of the photon wind emission produces additional photons, which have their directions chosen isotropically. The amount of radiation produced is attenuated by the escape probability.

thermal_trapping
Escape probability + anisotropic scattering

We use the 'thermal trapping method' to choose an anistropic direction when an r-packet deactivation or scatter occurs.

macro_atoms
Macro-atoms

use macro-atom line transfer. Packets are indivisible and thus all opacities are dealt with by activate a macro-atom, scattering, or creating a k-packet. the new direction following electron scattering or deactivation of a macro atom is chosen isotropically.

macro_atoms_thermal_trapping
Macro-atoms + anisotropic scattering

as macro_atoms, but we use the 'thermal trapping method' to choose an anistropic direction when an r-packet deactivation or scatter occurs.

File
setup_line_transfer.c

Child(ren)

- *Reverb.matom_lines*
- *Wind_heating.kpacket_frac*

Photons_per_cycle

Multi-line description, must keep indentation.

Type
Double

Values
Greater than 0

File
setup.c

Spectrum_cycles

Multi-line description, must keep indentation.

Type
Integer

File
setup.c

Child(ren)

- *Spectrum.orbit_phase*
- *Spectrum.no_observers*
- *Spectrum.wavemin*
- *Spectrum.select_photons_by_position*
- *Spectrum.type*
- *Spectrum.live_or_die*
- *Spectrum.select_specific_no_of_scatters_in_spectra*
- *Spectrum.wavemax*

Surface.reflection.or.absorption

When photons hit the disk, there are several options

Type
Enumerator

Values

reflect
The photons are scattered back into the wind

absorb
The photons are simply lost from the calculation

thermalized.rerad
The photons are absorbed, in the next ionization cycle energy lost is treated as extra heat, and the effective temperature of the ring in the disk will be increased accordingly

File
setup.c

Wind_heating

Todo: Fill in

Wind_heating.extra_luminosity

This is a very special option put in place for modelling FU Ori stars, and should be used with extreme caution. Determines the shock factor.

Type
Double

Values
Condition e.g. greater than 0 or list e.g. [1, 2, 5]

File
setup.c

Parent(s)

- *Wind_heating.extra_processes*: `nonthermal`, `both`

Wind_heating.extra_processes

Multi-line description, must keep indentation.

Type
Enumerator

Values

adiabatic
Multi-line description, must keep indentation.

both
Multi-line description, must keep indentation.

none
Multi-line description, must keep indentation.

nonthermal
Multi-line description, must keep indentation.

File
setup.c

Child(ren)

- *Wind_heating.kpacket_frac*
- *Wind_heating.extra_luminosity*

Wind_heating.kpacket_frac

Multi-line description, must keep indentation.

Type
Double

Unit
None

Values
Condition e.g. greater than 0 or list e.g. [1, 2, 5]

File
setup.c

Parent(s)

- *Wind_heating.extra_processes*: `nonthermal`, `both`
- *Line_transfer*: `macro_atoms`, `macro_atoms_thermal_trapping`

Spectrum

Todo: Fill in

Spectrum.angle

The inclination angle with respect to the polar axis for obtaining a spectrum. This question will be repeated once for each desired incliniation

Type
Double

Unit
Degrees

Values
0 to 90 degrees, where 0 is normal to the disk and 90 is on the disk plane

File
setup.c

Parent(s)

- *Spectrum.no_observers*: Greater than 0. Once per observer.

Spectrum.live_or_die

Normally in creating detailed spectrum Python "extracts" photons in a certain direction reweighting them to account for the fact that they have been extracted in a certain direction. It is possible to just count the photons that are emitted in a single angle range. The two methods should yield the same or very similar results but the extraction method is much more efficient and live or die is basically a diagnostic mode.

Type
Enumerator

Values

live.or.die
Count only those photons that escape within a small angle range towards the observer

extract
Extract a component of all photons that scatter towards the observer

File
setup.c

Parent(s)

- *Spectrum_cycles*: Greater than or equal to 0

Spectrum.no_observers

The number of different inclination angles for which spectra will be extracted.

Type
Integer

Values
Greater than 0

File
setup.c

Parent(s)

- *Spectrum_cycles*: Greater than or equal to 0

Child(ren)

- *Spectrum.angle*

Spectrum.orbit_phase

For binary systems, the orbital phase at which the spectrum is to be extracted (so the effects of an eclipse can be taken into account in creating the spectrum). Phase 0 corresponds to inferior conjunciton, that is with the secondary in front (or depending on inclination angle, partially in front of) the primary

Type
Double

Values
Between 0 and 1

File
setup.c

Parent(s)

- *Spectrum_cycles*: Greater than or equal to 0
- *System_type*: binary

Spectrum.select_azimuth

Advance command which along with several other parameters specifies a spherical region of space in cylindrical co-ordinates. This parameter desribes the azimuth of the region. When this general option is used, a detailed spectrum is constructed just from photons that originate or scatter int he region

Type
Double

Unit
Degrees

Values
Between 0, and 360 or -180 to 180

File
setup.c

Parent(s)

- *Spectrum.select_location*: spherical_region

Spectrum.select_location

One of several related parameters that permit one to apply additional conditions on the location of photons extracted in the detailed spectrum. The location refers here to the either where the photons was created or where it last scattered

Type
Enumerator

Values

all
Select photons regardless of where they are generated

below_disk
Select only photons generated from below (-z) the disk

above_disk
Select only photons orginating above the disk

spherical_region
Select photons by defining a spherical region

File
setup.c

Parent(s)

- *Spectrum.select_photons_by_position*: `True`

Child(ren)

- *Spectrum.select_r*
- *Spectrum.select_rho*
- *Spectrum.select_azimuth*
- *Spectrum.select_z*

Spectrum.select_photons_by_position

Advanced command associated with adding conditions for the detailed spectra that are extracted. This command simply asks whether one would like to construct spectra from photons that originate or last scattered from a certain regions of space.

If yes, then one will be asked to specify the regions for all extraction angles.

This option is useful for diagnostic purposes, such as differentiating between photons that read the observer from the near or far side of the disk.

Type
Boolean (yes/no)

File
setup.c

Parent(s)

- *Spectrum_cycles*: Greater than or equal to 0

Child(ren)

- *Spectrum.select_location*

Spectrum.select_r

Part of a set of parameters which define a spherical region of space from which photons are to be extracted. select_r defines the radius of the spherical region

Type
Double

Unit
cm

Values
Greater than 0

File
setup.c

Parent(s)

- *Spectrum.select_location*: spherical_region

Spectrum.select_rho

Advanced command which defines a spherical region of space from which photons are to be extracted in constructing a detailed spectrum. The region is defined by a cylindrical distance, and z height and azimuth, and a rho coordainate. This parameter defines the rho coordinate of the region.

Type
Double

Unit
cm

Values
Condition e.g. greater than 0 or list e.g. [1, 2, 5]

File
setup.c

Parent(s)

- *Spectrum.select_location*: spherical_region

Spectrum.select_scatters

Advaned command that allows one to construct spectra from photons that have undergone a certain number of scatters.

- If n >= MAXSCAT,that is to say a very large number, then all photons that could contribute to a spectrum are selected.
- If n lies between 0 and MAXSCAT then only contributions arising from photons that have scattered exactly n times will contribute to the spectrum.
- If n is < 0 then contributions from photons with n or greater scattters will be extracted.

This command is quite useful for gaining a better understanding of the nature of a line profile, including the relative importance of absorption and multiple scattering.

To explain the posiblities little more clearly, consider a photon which undergoes a total on n scatters. In extract mode, this photon will have made n+1 contributions to the total spectrum, one when it was first emitted, one when it scattered the first time, one when it scattered the second time, etc. If one chooses to construct a spectrum from photons that have one scatter, the contribution of this photon to the total spectra, at it's first scatter will be reported.

In live_or_die mode, a simlar process occurs, but in this case, one only counts spectra that escape with the desired number of scatters.

Type
Integer

Values
Greater than 0

File
setup.c

Parent(s)

- *Spectrum.select_specific_no_of_scatters_in_spectra*: `True`

Spectrum.select_specific_no_of_scatters_in_spectra

Advanced command which allows one to place additional constraints on the detailed spectra that are extracted.

If yes, then one will be asked to supply details for each extraction angle.

The command is useful for diagnositic purposes when one would like to separate contributions to the spectra from, for example, unscattered, singly scatterred, and multiply scattered photons.

Type
Boolean (yes/no)

File
setup.c

Parent(s)

- *Spectrum_cycles*: Greater than or equal to 0

Child(ren)

- *Spectrum.select_scatters*

Spectrum.select_z

Advanced command which defines a spherical region of space from which photons are to be extracted in constructing a detailed spectrum. The region is defined by a cylindrical distance, and z height and an aximuth, and a rho coordinate. This parameter defines the z coordinate of the region.

Type
Double

Unit
cm

Values
Within the z range of the model

File
setup.c

Parent(s)

- *Spectrum.select_location*: spherical_region

Spectrum.type

The type of spectra that are produced in the final spectra. The current choices are flambda, fnu, or basic, where basic implies simply summing up the energy packets that escape within a particularly wavelength/ frequency bin.

Type
Enumerator

Values

flambda
F()

fnu
F()

basic
F()

File
setup.c

Parent(s)

- *Spectrum_cycles*: Greater than or equal to 0

Spectrum.wavemax

The maximum wavelength of the detailed spectra that are to be produced

Type
Double

Unit
Angstroms

Values

Spectrum.wavemin
Greater than

File
setup.c

Parent(s)

- *Spectrum_cycles*: Greater than or equal to 0

Spectrum.wavemin

The minimum wavelength of the final spectra in Angstroms

Type
Double

Unit
Angstroms

Values
Greater than 0

File
setup.c

Parent(s)

- *Spectrum_cycles*: Greater than or equal to 0

Other

Todo: Fill in

Diag

A series of advanced/diagnostic commands (Accessed using the `-d` flag, see *Running Python*). The commands generally allow access to additional information from the simulation, or allow more precise control. Advanced commands have an @ symbol in front of them in the parameter file.

Note that some of these commands will also change the radiative transfer treatment in the code. A number of them are also only accessed when the `Diag.extra` command is answered with yes.

Todo: Fill in

Diag.adjust_grid

Choose whether or not you would like to adjust the scale length for the logarithmic grid. Advanced command.

Type
Boolean (yes/no)

File
setup_domains.c

Parent(s)

- *Wind.number_of_components*: Greater than 0. Once per domain.

Child(ren)

- *geo.xlog_scale*
- *geo.zlog_scale*

Diag.extra

Decide whether or not to use extra diagnostics in advanced mode. If set, this triggers a many extra questions that allow one to investigate things such as photon cell statistics, the velocity gradients in cells and the resonant scatters in the wind

Type
Boolean (yes/no)

File
python.c

Child(ren)

- *Diag.save_cell_statistics*
- *Diag.make_ioncycle_tables*
- *Diag.print_dvds_info*
- *Diag.save_photons*
- *Diag.save_extract_photons*
- *Diag.keep_ioncycle_windsaves*
- *Diag.track_resonant_scatters*

Diag.fractional_distance_photon_may_travel

The distance photon may travel in a cell is limited to prevent a photon from moving such a long path that the velocity may change non-linearly. This problem arises primarily when the photon is travelling azimuthally in the grid. This changes the default for the fraction of the maximum distance in a cell.

Type
Double

Values
0 to 1

File
diag.c

Parent(s)

- *Diag.use_standard_care_factors*: `False`

Diag.invoke_searchlight_option

Todo: Fill in

Type
Boolean (yes/no)

Diag.keep_ioncycle_windsaves

Decide whether or not to keep a copy of the windsave file after each ionization cycle in order to track the changes as the code converges. Produces files of format python01.wind_save and so on (02,03...) for subsequent cycles.

Type
Boolean(yes/no)

File
diag.c

Parent(s)

- *Diag.extra*: `True`

Diag.keep_photoabs_in_final_spectra

This advanced options allows you to include or exclude photoabsorpiotn in calculating the final spectra. (but ksl does not know what the default is)

Type
Boolean (yes/no)

File
diag.c

Parent(s)

- *Diag.use_standard_care_factors*: `False`

Diag.lowest_ion_density_for_photoabs

For efficiency reasons, Python does not try to calculate photoabsorption for an ion with an extremly low density. This advance parameter changes this density limit

Type
Double

Unit
n/cm**3

Values
Greater than 0

File
diag.c

Parent(s)

- *Diag.use_standard_care_factors*: `False`

Diag.make_ioncycle_tables

Multi-line description, must keep indentation.

Type
Boolean (yes/no)

File
diag.c

Parent(s)

- *Diag.extra*: `True`

Diag.partial_cells

Additional options for how to deal with cells that are partially filled by wind. Somewhat degenerate with the `-include_partial_cells` flag under *Running Python*.

Type
Enumerator

Values

include
Include wind cells that are only partially filled by the wind

zero_densities
Ignore wind cells that are only partially filled by the wind by zeroing their density

extend_full_cells
Experimental model that extends full cells

Diag.print_dvds_info

Print out information about the velocity gradients in the cells to a file root.dvds.diag.

Type
Boolean (yes/no)

File
diag.c

Parent(s)

- *Diag.extra*: `True`

Diag.save_cell_statistics

Choose whether to save the statistics for a selection of save_cell_statistics. If yes, it looks for a file called "diag_cells.dat" which contains the cells to track, and saves the photon details (weights, frequencies) for those that interact in the cell. Useful for checking the detailed MC radiation field in a cell.

Type
Boolean (yes/no)

File
diag.c

Parent(s)

- *Diag.extra*: `True`

Diag.save_extract_photons

Multi-line description, must keep indentation.

Type
Boolean (yes/no)

File
diag.c

Parent(s)

- *Diag.extra*: `True`

Diag.save_photons

Multi-line description, must keep indentation.

Type
Boolean (yes/no)

File
diag.c

Parent(s)

- *Diag.extra*: `True`

Diag.track_resonant_scatters

Multi-line description, must keep indentation.

Type
Boolean (yes/no)

File
diag.c

Parent(s)

- *Diag.extra*: `True`

Diag.turn_off_upweighting_of_simple_macro_atoms

Advanced command, allows one to turn off the "upweighting" scheme for simple ions in indivisible packet mode, as described under the *Bound-free Continua of Simple Atoms* section.

Type
Boolean (yes/no)

Diag.use_jumps_for_emissivities_in_detailed_spectra

Advanced command, allows one to go back to using the MonteCarlo jumps method, rather than the much faster matrix scheme for computing macro-atom emissivities in the spectral cycles (see *Macro-atom Emissivity Calculation*).

Type
Boolean (yes/no)

Diag.use_standard_care_factors

Advanced command which allows one to change various other defaults associated with radiative transfer, inclusing the fractional distance in a cell that a photon can travel

Type
Boolean (yes/no)

File
diag.c

Child(ren)

- *Diag.lowest_ion_density_for_photoabs*
- *Diag.keep_photoabs_in_final_spectra*
- *Diag.fractional_distance_photon_may_travel*

Diag.write_atomicdata

Choose whether to write the atomic data that is being used to an output file.

Type
Boolean (yes/no)

File
setup_domains.c

Photon_sampling

Todo: Fill in

Photon_sampling.approach

Choice of whether and how to use stratified sampling in creating photons during the ionization stage of the calculation.

Type
Enumerator

Values

T_star
Sets a single band based on the temperature given

cv
Traditional cv setup

yso
YSO setup

AGN
Test for balance matching the bands we have been using for AGN runs

min_max_freq
Mode 1 sets a single wide band defined by f1 and f2

user_bands
User-defined bands

cloudy_test
Set up to compare with cloudy power law table command note that this also sets up the weight and photon index for the PL, to ensure a continuous distribution

wide
Test for balance to have a really wide frequency range

logarithmic
Generalized method to set up logarithmic bands

File
bands.c

Child(ren)

- *Photon_sampling.nbands*
- *Photon_sampling.high_energy_limit*
- *Photon_sampling.low_energy_limit*

Photon_sampling.band_boundary

When the user specifies what bands are used for stratfied sampling, this parameter specifies the boundaries between energy bands in which a minimum fraction of photons will be generated. The number of times this parameter is request depends upon the number of energies bands being used.

Type
Double

Unit
eV

Values
Greater than 0, monotonically increasing

File
bands.c

Parent(s)

- *Photon_sampling.nbands*: Greater than 0, once per band

Photon_sampling.band_min_frac

When specifying manually the bands used for generating photons during the ionization phase, this parameter specifies the The minimum fraction of photons to be generated in this energy band. The number of times this parameter will be reqested depends upon the number of bands. The summ of the fractions need not sum to 1, in which case the remaining photons will be distributed according to the luminosity in the energy bands

Type
Double

Values
Greater than 0 and should sum to less than 1 over all bands

File
bands.c

Parent(s)

- *Photon_sampling.nbands*: Greater than 0, once per band

Photon_sampling.high_energy_limit

Stratified sampling is used during ionization cycles to generate photons. This parameter specifies the high energy limit for the frequencies of photons to be generated.

Type
Double

Unit
eV

Values
Greater than *Photon_sampling.low_energy_limit*

File
bands.c

Parent(s)

- *Photon_sampling.approach*: user_bands

Photon_sampling.low_energy_limit

During the ionization phase, stratified sampling is used to provide good coverage of the full ionizing spectrum. This parameter sets the lowest envergy (frequency) of for phtoons to be generated whne the user wants to customize the bands.

Type
Double

Unit
eV

Values
Greater than 0

File
bands.c

Parent(s)

- *Photon_sampling.approach*: user_bands

Photon_sampling.nbands

Python uses stratified samplign to generate photons during the ionization phase. This parameter allows the user to define the number of bands for stratified sampling, if s/he wants to customize the bands used for the generation of photons

Type
Integer

Values
Greater than 0

File
bands.c

Parent(s)

- *Photon_sampling.approach*: user_bands

Child(ren)

- *Photon_sampling.band_min_frac*
- *Photon_sampling.band_boundary*

Reverb

Todo: Fill in

Reverb.angle_bins

Used when generating 3d .vtk output files for visualisation. Sets the number of angle bins used in the output. Aesthetic only; bigger makes prettier meshes with larger filesizes.

Type
Integer

Values
Greater than 0

File
setup_reverb.c

Parent(s)

- *Reverb.visualisation*: `vtk`, `both`

Reverb.disk_type

Setting for how photons generated in the disk are treated when generating path distributions for wind cells.

Type
Enumerator

Values

correlated
This mode assumes that disk emission is correlated with the central source. Photons generated in the disk start with a delay equal to the direct distance to the central source. This assumes that the ionisation state and luminosity of the disk surface layer is mostly determined by unscattered photons from the central source.

uncorrelated
This mode generates photons with a delay of 0 wherever in the disk they come from. This mode is of slightly questionable use and should be ignored in preference to 0 or 2. It will, in practise, generally work out similar to type 0 as most of the UV photons are generated close-in to the CO.

ignore
This mode assumes that disk photons do *not* correlate with the central source (i.e. disk surface ionisation state and emissivity is driven not by irradiation from the CO but by the mass inflow). This means that whilst they contribute to heating the wind, they do not strongly contribute to the lags for a given line. Photons generated by the disk do not contribute to the path distributions in the wind in this mode.

By removing the (generally) short-delay disk photons from the wind path distributions, this will slightly bias them towards the longer delays associated with wind self-heating/excitation.

File
setup_reverb.c

Parent(s)

- *Reverb.type*: `wind`, `matom`

Reverb.dump_cell

Position for a cell, listed as a pair of R:Z coordinates. Will accept any position that falls within a grid, will error out on ones that don't. This can be slightly awkward and you may want to run a quick test then use py_wind to idenfity where wind locations are.

Type
Float:Float

Unit
cm:cm

Values
>0:>0

File
setup_reverb.c

Parent(s)

- *Reverb.dump_cells*: Greater than 0

Reverb.dump_cells

Number of cells to dump. When dumping the path distribution info for a range of cells, this specifies the number of lines of *Reverb.dump_cell* that will be provided.

Type
Integer

Values
Greater than or equal to 0

File
setup_reverb.c

Parent(s)

- *Reverb.visualisation*: `wind`, `matom`

Child(ren)

- *Reverb.dump_cell*

Reverb.filter_line

Line number of one line to include in the output `.delay_dump` file. This is the python internal line number. It can be found using either the macro-atom mode (which prints out the line number once it's found one) or by doing an exploratory run with *Reverb.filter_lines* = -1, then looking through the delay dump file for photons of the right wavelength to see what their line is. This should almost certainly be changed to be specified using a species and wavelength!

Type
Integer

Values
Any valid line index

File
setup_reverb.c

Parent(s)

- *Reverb.filter_lines*: Greater than 0, once per filer line.

Reverb.filter_lines

Whether or not to filter any lines out of the output file. This is used to keep output file sizes down, and avoid them overwhelming the user.

Type
Int

Values

0
No filtering

Include *all* photons that contribute to the spectra in the output file. Not recommended as it leads to gargantuan file sizes.

-1 *Filter continuum*

Include all photons whose last interaction was scatter or emission in a line. Recommended setting for exploratory runs where you'd like to identify which lines are the easiest to process.

N
Filter lines

Include N *Reverb.filter_line* entries, each specifying one line to keep in the output file. If *Reverb.matom_lines* is >0, all macro-atom lines of interest are automatically included in the filter list.

File
setup_reverb.c

Parent(s)

- *Reverb.type*: `wind`, `matom`

Child(ren)

- *Reverb.filter_line*

Reverb.matom_line

Specifies a line associated with a given macro-atom transition. The species and transition involved are specified. The internal line associated with this transition will be printed to standard-out for use when processing outputs. A line is specified as Element:Ion:Upper level:Lower level.

Type
Int:Int:Int:Int

Values
>0:>0:>1:>0

File
setup_reverb.c

Parent(s)

- *Reverb.matom_lines*: Greater than 0, once per matom line.

Reverb.matom_lines

Number of macro-atom lines to track paths for individually. This many reverb.matom_line entries are required, and the line associated with each has the path of photons deexciting into it recorded in its own array. Note: This doesn't give rise to any noticable differences to the pure wind mode in most simulations.

Type
Integer

Values
Greater than or equal to 0

File
setup_reverb.c

Parent(s)

- *Reverb.type*: matom
- *Line_transfer*: `macro_atoms`, `macro_atoms_thermal_trapping`

Child(ren)

- *Reverb.matom_line*

Reverb.path_bins

Number of bins for photon paths. Reverb modes that record the distribution of path lengths in every wind cell bin them in this number of bins. Bins are logarithmically spaced between the minimum scale in the system (the smallest 'minimum radius' in any domain) and the 10 * the maximum scale in the system (10 * the 'maximum radius' in any domain). Default value is 1000, going much higher does not lead to qualitative differences in TF, going lower makes the bin boundaries show up in the TF.

Type
Integer

Values
Greater than 0

File
setup_reverb.c

Parent(s)

- *Reverb.type*: `wind`, `matom`

Reverb.type

Whether to perform reverberation mapping. Reverberation mapping tracks the path of photons emitted in the simulation as they travel through the geometry, assuming that any delays from recombination etc. are negligible and all delays are due to light travel time. For each final spectrum, all contributing photons are output to a '.delay_dump' file that can then be processed using our 'tfpy' Python (no relation) library.

Type
Enumerator

Values

none
Off

photon
Each photon is assigned an initial path based on its distance from the central source (assuming emission in the disk and wind is correlated with emission from the CO).

wind
CO photons are assigned paths as in Photon mode, disk photons are assigned paths as set by the reverb.disk_type parameter. Photons generated in the wind are assigned a path based on the *distribution* of paths of photons that have contributed to continuum absorption in that cell.

matom
This works as wind mode, but for a number of specified macro-atom lines paths are tracked for those photons who cause a deexcitation into a given line. When a photon is emitted in one of those lines, the path is drawn from that specific distribution. This distribution is build up not just from the last cycle of the simulation, but from all cycles after the wind achieves >90% convergence. This is necessary as some lines are poorly-sampled.

This mode gives pretty much identical results to wind mode, but at least we made it to check rather than just assuming it would be fine.

This requires that *Line_transfer* is either `macro_atoms` or `macro_atoms_thermal_trapping`

File
setup_reverb.c

Child(ren)

- *Reverb.matom_lines*
- *Reverb.filter_lines*
- *Reverb.path_bins*
- *Reverb.visualisation*
- *Reverb.disk_type*

Reverb.visualisation

Which type of visualisation to output, if any. Reverb modes that keep arrays of photon paths per cell can output them either as averages in a 3d model, or as a selection of flat text files with full bin-by-bin breakdowns. Useful for diagnostics.

Type
Enumerator

Values

none
No visualisation.

vtk
Mesh visualisation. Outputs mean incident path per cell, photon count per cell, and mean observed delay to '.vtk' format, readable using a range of programs including (my preferred option) VisIt, available at https://visit.llnl.gov/.

dump
Outputs distributions of paths for continuum heating and each line to a range of 'dump cells' specified by X & Z position.

both
Outputs both vtk and dump.

File
setup_reverb.c

Parent(s)

- *Reverb.type*: `wind`, `matom`

Child(ren)

- *Reverb.dump_cells*
- *Reverb.angle_bins*

geo

Todo: Fill in

geo.xlog_scale

Choose the logarithmic scale length for the grid in the x-direction.

Type
Double

Unit
cm

File
setup_domains.c

Parent(s)

- *Diag.adjust_grid*: `True`

geo.zlog_scale

Choose the logarithmic scale length for the grid in the z-direction.

Type
Double

Unit
cm

File
setup_domains.c

Parent(s)

- *Diag.adjust_grid*: `True`

Input_spectra.model_file

In addition to being able to generate several types of spectra, such as blackbodies and power laws, Python can read in a series of spectra which are tabulated and are describable in terms of (usually) temperature and gravity). This parameter defines the name of the file which gives the location of the individual spectra and the temperate and gravity associated with each spectrum. (One may wish to use the same files for several radiation sources, viz the disk and the star) Python actually only reads in the data the first time.

Type
String

File
setup.c

Parent(s)

- *Central_object.rad_type_to_make_wind*: models
- *Central_object.rad_type_in_final_spectrum*: models
- *Disk.rad_type_to_make_wind*: models
- *Disk.rad_type_in_final_spectrum*: models
- *Boundary_layer.rad_type_to_make_wind*: models
- *Boundary_layer.rad_type_in_final_spectrum*: models

Top level parameters

Todo:

Fill in

- *Photon_sampling.approach*
- *Disk.type*
- *Central_object.radiation*
- *Ionization_cycles*
- *Diag.write_atomicdata*
- *Wind.ionization*
- *Diag.extra*
- *Wind.radiation*
- *Reverb.type*
- *Central_object.mass*
- *Photons_per_cycle*
- *Central_object.radius*
- *Wind_heating.extra_processes*
- *Diag.use_standard_care_factors*
- *Line_transfer*
- *Spectrum_cycles*

- *System_type*
- *Surface.reflection.or.absorption*

Top-level parameters

- *System_type*
- *Central_object.mass*
- *Central_object.radius*
- *Central_object.radiation*
- *Disk.type*
- *Wind.ionization*
- *Wind.radiation*
- *Photons_per_cycle*
- *Spectrum_cycles*
- *Ionization_cycles*
- *Wind_heating.extra_processes*
- *Line_transfer*
- *Surface.reflection.or.absorption*
- *Photon_sampling.approach*
- *Reverb.type*
- *Diag.extra*
- *Diag.use_standard_care_factors*
- *Diag.write_atomicdata*

3.5 Outputs & Evaluation

Python produces a large number of files in both binary and ascii format. Tools exist to examine the binary files.

3.5.1 Diagnostic files

Python logs a considerable amount of information as it runs. Some of this information is printed to the screen but a much more voluminous version of progress of the program is placed in a sub-directory, named diag_whatever, where whatever is the root name of the model being run.

In this directory one will find log files, e.g. **whatever_0.diag**, **whatever_1.diag**, and so on, where the in a multiprocessor run, the number refers to the log of a specific thread.

Inspecting these logs is important for understanding whether a Python run is successful, and for understanding why if failed if that is the case.

3.5.2 Evaluation

Determining whether Python has run successfully from a a scientific point of view depends very specifically on one's goals. Did the spectra turn out to be what one expected? Here by evaluation we mean, did my run complete without significant errors and did the ionization structure converge to a "steady state" given the number of ionization cycles, the number of photons, and the frequency distributions of the photons we chose.

Convergence

Ionization cycles in Python are intended to establish the correct ion densities and temperature of the various cells in the wind. The degree to which this happens for a given number of ionization cycles depends on how far the initial guess of electrons temperatures in various portions of the wind and the number of photons generated during each photoionization cycles. Furthermore, the accuracy of the final model depends on the number of photons that pass through each cell. As a result, the accuracy with which ion abundances and temperature are determined will differ on a cell by cell basis. In a typical model with a biconical outflow, a small cells at the outer edge of the accretion disk will record fewer photon passages than one in the middle of the grid that is exposed to large numbers of photons from the disk.

A very basic question about a particular run is, has it reached a "steady state" and if it is in a steady state are cells stable in the sense that fluctuations are small. Hopefully, each ionization cycle brings one closer and closer to to a solution after which the ionization structure no longer evolves. Of course, since Python is a Monte Carlo code, the degree to which the solution stays constant from cycle to cycle is limited by counting statistics. To check convergence in Python, we monitor the the radiation temperature T_r, the electron temperature T_e, and the total heating $\mathrm{Heat_{tot}}$ and cooling $\mathrm{Cooling_{tot}}$ in each cell as a function of the ionization cycle n.

To estimate whether a a model calculation has reached a steady state, Python carries out three tests, one comparing the difference in the radiation temperature between the current and the preceding ionization cycle, one comparing the electron temperature in the same manner and once comparing heating and cooling rates in the current cycle. If a cell satisfies the following 3 tests,

$$\left|\frac{T_e^n - T_e^{n-1}}{T_e^n + T_e^{n-1}}\right| < \epsilon$$

$$\left|\frac{T_r^n - T_r^{n-1}}{T_r^n + T_r^{n-1}}\right| < \epsilon$$

$$\left|\frac{\mathrm{Heat_{tot}}^n - \mathrm{Cooling_{tot}}^n}{\mathrm{Heat_{tot}}^n + \mathrm{Cooling_{tot}}^n}\right| < \epsilon$$

where $\epsilon = 0.05$, it is flagged as converged.

It is rare that all of the cells in a model will satisfy all of these criteria. That is is because the number photons passing that pass through a cell vary considerably and the statistical noise depends on the the number of photons. It is important to note that the photons that contribute most to the spectra of an object will be those which have the most photons passing through them. Typically, we consider a model to be well converged if 90% of the cells are converged by this criterion.

The routine `run_check.py` (see *Python Script documentation*) produces two plots related to convergence, one showing the fraction of cells that have passed each of the tests above as a function of cycle, and the other showing the number of failed checks for each cell in the wind.

Note that it is not always important that all cells be converged. The Monte Carlo process preferentially picks out the cells which affect the emergent radiation field. Portions of the grid which do not get many photons are typically the ones that are "not converged", but since they don't contribute much to the emergent radiation, one does not need them to be converged (except if one wants to make nice plots of the temperature as a function of position in the wind or of the density of a particular ion species). On the other hand, if one is using Python in conjunction with a hydrodynamical code one wants all the cells to be converged.

Errors

Python is designed to continue to run unless something catastrophic happens. As it runs, it logs error messages that can be found in the .diag files. These messages are a combinations of warnings, and/or unusual occurrences, that if they start occurring often suggest a real problem.

These error messages are all of the form:

```
Error: wind2d: Cell   0 ( 0, 0) in domain 0 has 1 corners in wind, but zero volume
```

that is they begin with the word `Error`. followed by the subroutine in the code where the error occurred followed by what is hopefully a helpful. If one is concerned about a particular message, one can hopefully determine what is happening by looking for that message in the log files.

Python keeps a count of the number of times a particular message has occurred and at the end of the program, and the very end of the diag files contain a listing of how many times a particular error has occurred.

```
Error summary: End of program, Thread 2 only
Recurrences --  Description
     7 -- getatomic_data: line input incomplete: %s
   128 -- get_atomicdata: Could not interpret line %d in file %s: %s
     1 -- error_count: This error will no longer be logged: %s
     1 -- get_wind_params: zdom[ndom].rmax 0 for wind type %d
     1 -- wind2d: Cell %3d (%2d,%2d) in domain %d has %d corners in wind, but zero volume
     1 -- check_grid: velocity changes by >1,000 km/s in %i cells
     1 -- check_grid: some cells have large changes. Consider modifying zlog_scale or␣
↪grid dims
```

As indicated here, these are the errors for only thread 2 of a program. In order to get a summary of all the threads, there is a script py_error.py that be run as `py_error.py rootname` from the main run directory. Note that in many cases, the summary will be the number times an error occurred in one thread times the number of threads, but not always.

One should be aware of these errors, and watch out for situations where the number of errors of a particular type is much larger than usual.

3.5.3 Model

As Python is run, it repeatedly writes out two binary files that contain essentially all information about the wind as calculated in the ionization phase of the program, along with status of the program at the last point where the file was written. These files along with the parameter file are sufficient to restart the program, if for example, one wants to check point the program after a certain time, and restart where one left off, or to add spectral cycles to get better spectra.

.wind_save
: A binary file that contains essentially all information about the wind including ion densities, temperatures, and velocities in each cell, along with status of the program at the last point where the file was written.

.spec_save
: A binary file that contains all of the information about the spectra that have created. This file is not of interest to users directly. It is used when restarting

Two routines exist as part of the Python distribution allow the user to gain insight into the actual model

windsave2table
: Executed from the command line with `windsave2table rootname`.

  Produces a set of standard set ascii tables that that show for each grid cell quantities such as wind velocity, n_e, temperatures, and densities of prominent ions.

py_wind
: Executed from the command line with `py_wind rootname`

  Allows the user to query for information about the model interactively. The results can be written to ascii files for future reference

3.5.4 Spectra Files

Python is intended to produce simulated spectra. These spectra are all ascii tables intended to be accessible with software packages such as astropy.

All of the ascii begin with commented headers that contain all of the parameters of associated with a run, along with the date of the run and the specific version of Python used to make the run. In principle, if one still has access to any of the spectra, one can reproduce the entire run again.

Broad band spectra are created from the last ionization cycle. (More accurately the broad band spectra are written out at the end of each ionization cycle, so one the program is finished one has the broad band spectrum of the last cycle)

Detailed are calculated from all of the spectral cycles. (Properly normalized spectra are written out at the end of each spectral cycle, and with each cycle the photon statistics improves.)

The units in which the spectra are written out is also indicated in the header to the file.

For a model with root name *cv*, the following broadband spectra will be created:

- **cv.spec_tot**
- **cv.log_spec_tot**
- **cv.spec_tot_wind**
- **cv.log_spec_tot_wind**

File types

.spec_tot
: An ascii file that contains various spectra from the ionization-calculation phase of the program on a linear frequency scale. The first few lines of the file (omitting the header) are as follows:

```
Freq.        Lambda     Created    WCreated   Emitted    CenSrc     Disk       Wind␣
↪      HitSurf
2.524334e+14 11876.102  3.5244e+18          0 3.5244e+18          0 3.5244e+18     ␣
↪    0 1.1547e+16
2.550397e+14 11754.737  3.4721e+18          0 3.4721e+18          0 3.4721e+18     ␣
↪    0 2.8761e+15
2.576461e+14 11635.827  3.4433e+18          0 3.4433e+18          0 3.4433e+18     ␣
↪    0 2.8835e+15
2.602524e+14 11519.299  3.6858e+18          0 3.6858e+18          0 3.6858e+18     ␣
↪    0 2.8706e+15
2.628587e+14 11405.082  3.6711e+18          0 3.6711e+18          0 3.6711e+18     ␣
↪    0 1.1528e+16
```

The first two columns are fairly obvious. Lambda is in Angstroms.

The remainder indicate the luminosity, that is L_ν of the system for specific types of photons. The units are $\mathrm{erg\ s^{-1} Hz^{-1}}$.

The remaining columns are:

- Created is the total spectrum of all of the photons paakets as created, that is before having been translated through the wind
- WCreated is the spectrum of the photons that are created in the wind before translation
- Emitted is the emergent spectrum after the photons have been translated through the wind
- CenSrc is the emergent spectrum from photon bundles originating from the Star or BL,
- Disk is the emergent spectrum from photon bundles originating from the disk,
- Wind is the emergent spectrum from photon bundles that have been reprocessed by the wind,
- HitSurf represents photons that did not escape the system but ran into a boundary

.log_spec_tot
An ascii file which contains the same information as *.spec_tot*, but with a logarithmically space frequency intervals. This gives better sampling of the SED in a lot of cases and is much better for plotting things such as the input spectrum.

.spec_tot_wind
Identical to *.spec_tot* but just including photons that were generated in the wind or scattered by the wind

.log_spec_tot_wind
A logarithmic version of *.spec_tot_wind*

.spec
an ascii file that contains the final detailed spectra for the wavelengths of interest at a distance of **100 pc**. The units for the detailed spectra are determined by the input parameter Spectrum.type.

Photons bundles are generated in cycles in Python and the *.spec* file is actually written out at the end of each cycle as the program is running in the spectrum-generation phase of the program. So one can inspect the spectrum as it is building up.

The beginning of the file (omitting the header) is as follows:

```
Freq.        Lambda     Created    WCreated   Emitted    CenSrc     Disk       Wind␣
↪      HitSurf    Scattered  A01P0.50   A30P0.50   A60P0.50   A80P0.50
5.998778e+14  4997.560  5.5824e-13          0 5.5824e-13          0 5.5824e-13     ␣
↪    0 1.0097e-15          0 1.9797e-12  1.141e-12 4.0282e-13  1.068e-13
6.001705e+14  4995.122  6.4224e-13          0 6.4224e-13          0 6.4224e-13     ␣
↪    0 1.3472e-15          0 2.0123e-12 1.2369e-12 5.1482e-13 1.0398e-13
6.004632e+14  4992.687  7.2239e-13          0 7.2239e-13          0 7.2239e-13     ␣
↪    0          0          0 1.8656e-12 1.2165e-12 4.9179e-13 1.3359e-13
6.007560e+14  4990.254  7.4183e-13          0 7.4183e-13          0 7.4183e-13     ␣
↪    0 6.7702e-16          0 1.7185e-12 1.4226e-12 5.9175e-13 1.6808e-13
6.010487e+14  4987.824  7.9709e-13          0 7.9709e-13          0 7.9709e-13     ␣
↪    0 3.3825e-16          0  2.262e-12 1.6291e-12 7.2959e-13 1.4697e-13
```

Where the first set columns are as follows:

- Frequency in Hz
- Wavelength in Angstorms
- The spectrum of photons which are created (before passing through the wind)
- The spectrum of all photons which are created in the wind (before processing by the wind)
- The spectrum of all photons which escape the wind (after passing through the wind)
- The spectrum of all photons created by the star or BH (after passing through the wind)
- The spectrum of all photons created by the wind (after passing though the wind)

- The spectrum of all photons that are scattered by the wind (after passing through the wind)

These data in the first set of columns do not reflect the angular dependence of the emission. They are effectively an angle averaged spectrum. Except for the fact that the units are different and the wavelength range is limited these should resemble the spectra in the output files (such as .spec_tot) that record the spectra constructed in the ionization cycles.

The remaining columns are the spectra at various inclination angles and binary phases. The label A30P0.50 means the spectrum is viewed at an inclination angle of 30 degrees and at a phase of 0.5 – for a binary system this is when the secondary was located behind the primary.

.log_spec
: Identical to the spectrum .spec file except with logarithmic intervals.

3.5.5 Issues with Generating Spectra

With the current machinery to create spectra, it is possible to come across the situation where models with large optical depth or wind velocities will generate spectra with different flux normalisation depending on the wavelength range.

This problem was originally encountered whilst modelling Tidal Disruption Events. Two spectra for the same model were generated over two wavelength ranges; a restricted (1100 - 2600 A) and a broader (500 - 5000 A) range. The problem encountered was that the broad range spectrum had *more* flux than the spectrum with the restricted range. The figure below shows the same model, but over two wavelength ranges - as well as two spectra where the maximum number of scatters a photon can undergo is changed,

- **tde_flux_small_range:** The restricted wavelength range
- **tde_flux_large_range:** The broad wavelength range
- **tde_flux_small_range_maxscat:** The restricted wavelength range with a value `MAXSCAT = 50`
- **tde_flux_no_maxscat:** The restricted wavelength range with no `MAXSCAT` limit

The problem here is not caused by a bug with the code, but is a consequence of the large wind velocities and optical depths of the model. We currently believe that there are two reasons why the flux differs between these two wavelength ranges.

Doppler Shifting out of the Spectrum Wavelength Range

At the edges of the restricted spectrum above, the flux is reduced. This is due to photon frequencies being shifted outside of the wavelength range of the spectrum. If a significant number of photons are removed from the spectrum in this way, then the following Error is printed,

```
spectrum_create: Fraction of photons lost: 0.10 wi/ freq. low, 0.19 w/freq hi
```

This tells one the fraction of the photon sample which does not contribute towards the spectrum due to to the photon frequencies being larger or smaller than the defined spectrum range, due to Doppler shifting. In models with large wind velocities (0.2 - 0.5 c) and a small spectral range, the fraction of photons lost is large and the flux at the edge of generated spectra is reduced - as can be seen above in the above figure. However, when the wind has a more moderate velocity, the number of photons lost due to being shifted out of the range is much lower and does not produce a noticeable effect on the flux normalisation of the spectra.

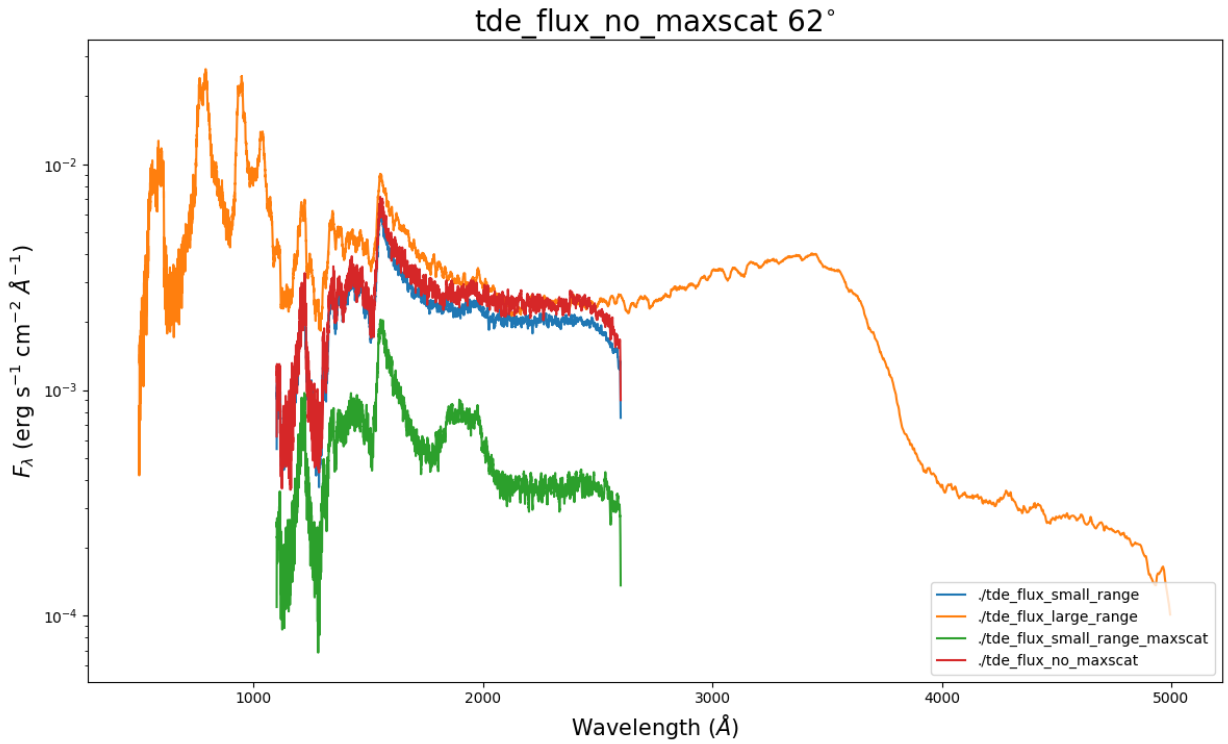


Fig. 1: Example spectra showing differing flux totals

Removing Photons due to Too Many Scatters

As well as edge effects, flux can be lost due to photons being removed from the photon sample due to scattering too many times. In Python, when a photon has undergone *MAXSCAT = 500* scatters, a photon is assumed to have become stuck in the wind and hence it is terminated and no longer tracked.

In models with large optical depths, the number of photons terminated in this way can become large. During spectrum generation, these photons will never fully escape the system but will only contribute partially to the spectrum due to extract - they will never contribute if Live or Die is used instead.

At current, there is no logic to detect this and hence no error is given. However, it is often insightful to read the output from the *Photons contribution to the various spectra* table, as shown below,

```
Photons contributing to the various spectra
Inwind   Scat    Esc     Star    >nscat    err    Absorb   Disk    sec    Adiab(matom)
     0       0    3455       0       0       0       0       0       0       0
     0       0    3455       0       0       0       0       0       0       0
     0       0     427       0       0       0       0       0       0       0
     0       0    1598       0       0       0       0       0       0       0
     0       0    1430       0       0       0       0       0       0       0
     0       0       0       0       0       0       0       0       0       0
     0       0    3313       0   17209       0     169       0       0       0
     0  282914  487831    1474       0       0     129  223605       0       0
     0  441756  336514    2223       0       0     135  215325       0       0
     0  609395  180082    5677       0       0      94  200705       0       0
     0  750672   61308   12010       0       0      59  171904       0       0
     0  838923   26143   29057       0       0      55  101775       0       0
```

In the above table, one can see that 17,209 photons which scattered more than `MAXSCAT` times contributed to the the scattered spectrum, suggesting that a large number of photons were terminated due to too many scatters.

Note: The photon numbers presented in this table are only for the master MPI process. Hence, if running in multi-processor mode, the number here will never equal the total number of photons in the simulation, but only the number of photons in the current process.

GitHub Issue

The original **GitHub** issue discussing this problem can be found here; #471.

3.6 Plotting & Processing Outputs

Python produces a large number of files in both binary and ascii format. Tools exist to examine the binary files.

3.6.1 Plotting a Spectrum

This notebook explains how to read and plot a spectrum for the `cv_standard` file found in the examples. Before running the python commands, you need to run the model from the command line. I suggest running the following commands, after you have compiled python:

```
mkdir cv_test
cd cv_test
cp $PYTHON/examples/basic/cv_standard.pf .
py cv_standard </code>
```

The model will take about 5 minutes to run on a single core. It will not converge and the spectrum will be a bit noisy, but will give us a model to use as an example.

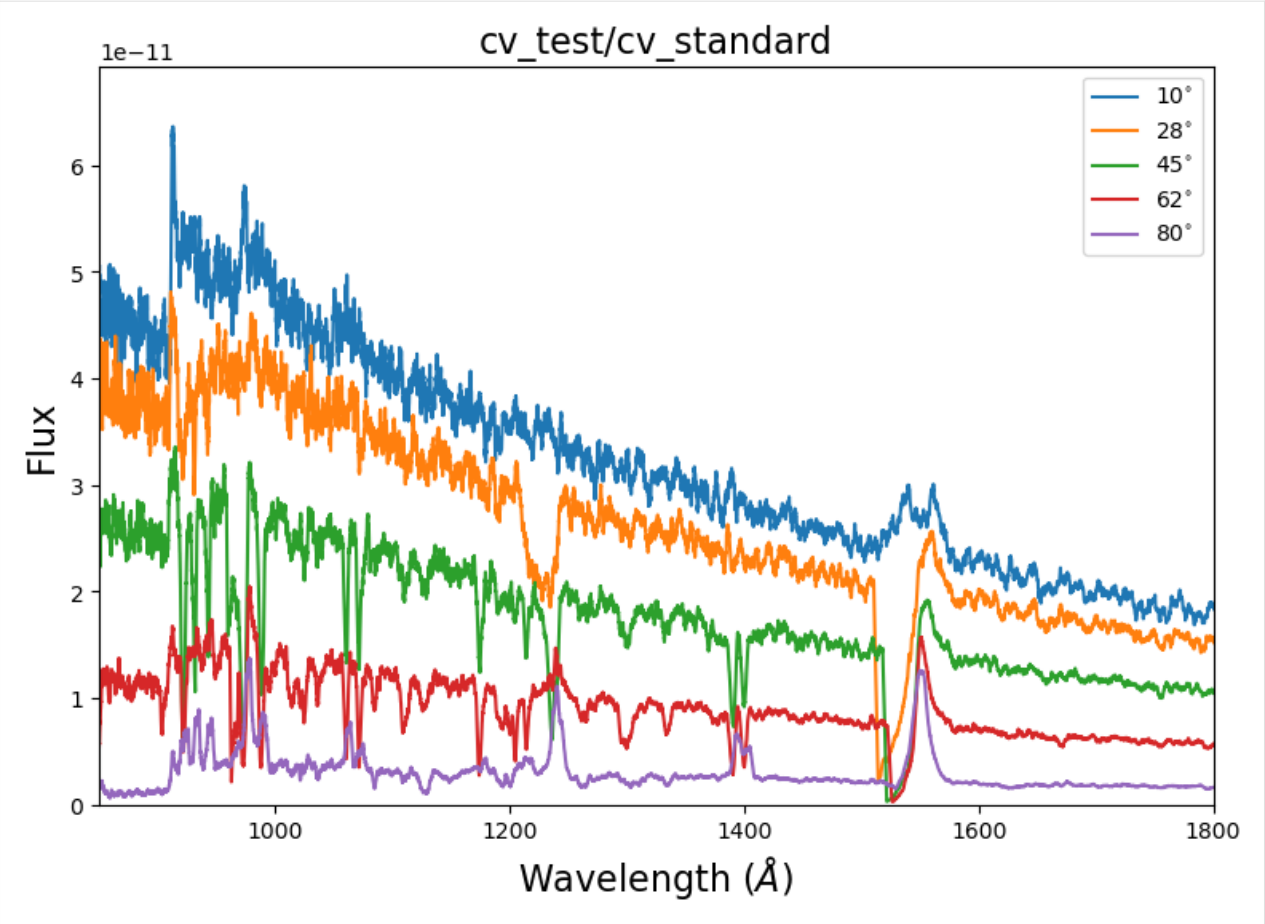
The simplest way to make a quick look spectrum plot is using the `plot_spec.py` routine in `$PYTHON/py_progs`. In this example, I will assume py_progs has been added to `$PATH` and to `$PYTHONPATH`. `plot_spec.py` can be run from the command line using

```
plot_spec.py [-wmin 850 -wmax 1850 -smooth 21] cv_standard
```

where the flags control the minimum and maximum wavelengths. Alternatively, it can be run from within python by doing:

[2]:
```
%matplotlib inline
import plot_spec
wmin, wmax = 850,1800
smooth = 21
fname = "cv_test/cv_standard"
plot_spec.do_all_angles(fname, smooth, wmin, wmax)
```

[2]:
```
'cv_test/cv_standard.png'
```



You may, however, wish to get more direct access to the data, which can be done easily by reading in the `cv_standard.spec` file, for example using `astropy`. In the next code block, we read in the spectrum file and print out the columns.

[3]:
```
import matplotlib.pyplot as plt
import astropy.io.ascii as io

s = io.read("{}.spec".format(fname))

print (s.colnames)
```

```
['Freq.', 'Lambda', 'Created', 'WCreated', 'Emitted', 'CenSrc', 'Disk', 'Wind', 'HitSurf
↪', 'Scattered', 'A10P0.50', 'A28P0.50', 'A45P0.50', 'A62P0.50', 'A80P0.50']
```

The first two columns are: * **Freq.:** frequency in Hz * **Lambda:** wavelength in Angstroms

The next set of columns correspond to: * **Created:** total spectrum of all of the photons paakets as created, that is before having been translated through the wind * **WCreated:** spectrum of the photons that are created in the wind before translation * **Emitted:** is the emergent spectrun after the photons have been translated through the wind * **CenSrc:** is the emergent spectrum from photons bundles originating on the Star or BL, * **Disk:** spectrum due to photons starting in the disk * **Wind:** spectrum due to photons starting in the wind * **HitSurf:** photons that did not escape the system but ran into a boundary

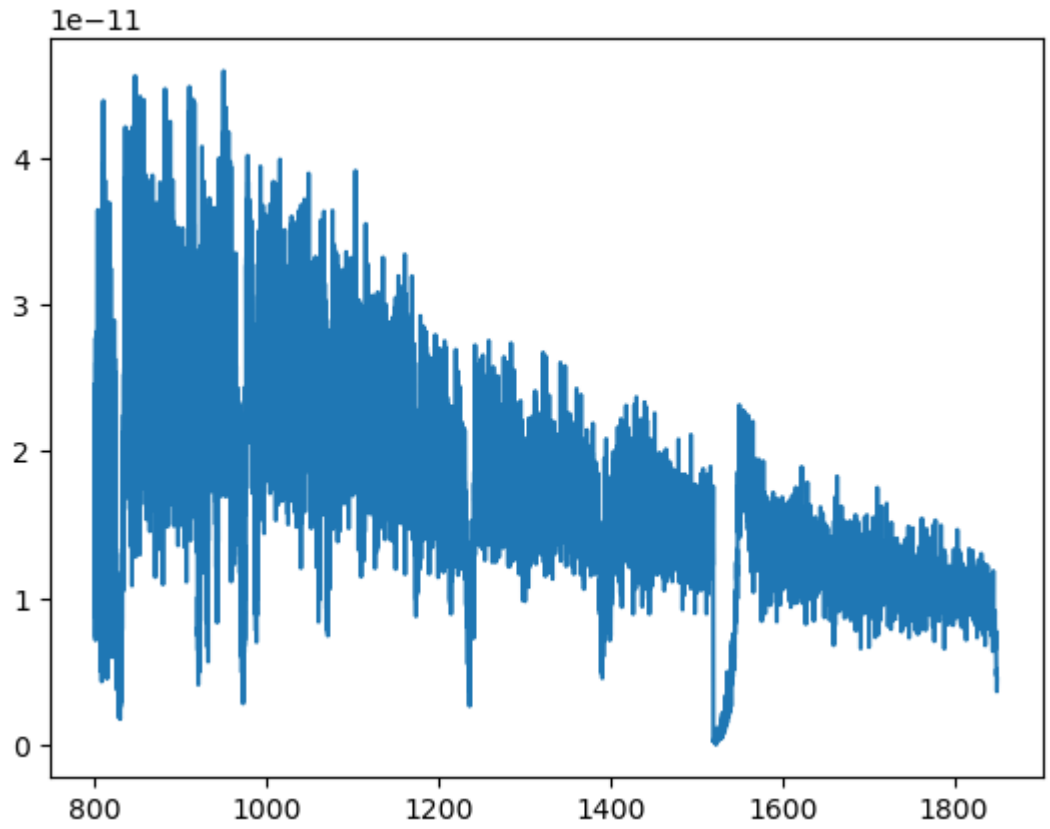
The remaining columns show the spectrum extracted at various angles, where A45P0.50 denotes an inclination of 45 degrees with respect to the polar axis, and a phase of 0.50 relative to inferior conjunction. Phase only matters if a companion is present.

Units: The units depend on whether flambda or fnu has been requested by the user, but correspond to CGS units either in per Angstrom or per Hz.

We can now plot one of the spectra.

[4]:
```
angle = 45
field = "A{:.0f}P0.50".format(angle)
plt.plot(s["Lambda"], s[field])
```

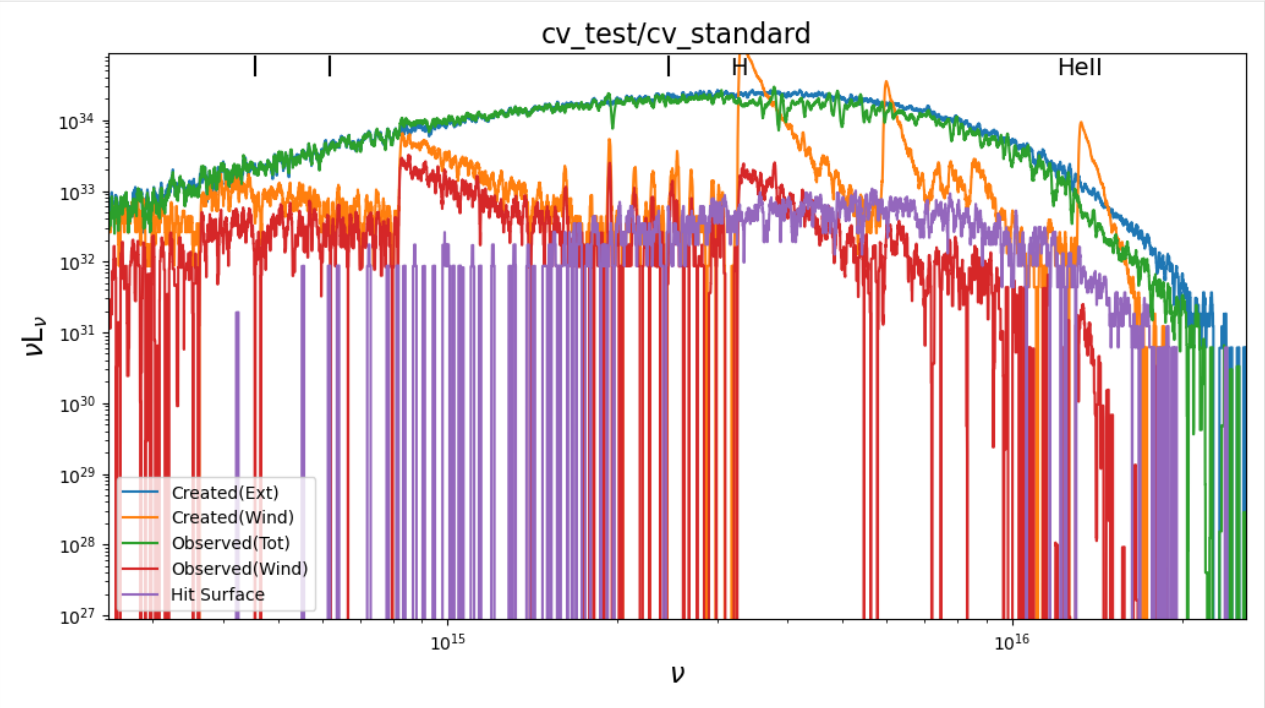
[4]:
```
[<matplotlib.lines.Line2D at 0x7fa5e132e550>]
```



We can also plot the components contributing to the total escaping spectrum in the requested wavelength range using the `plot_tot.py` script. Note that this script reads the `cv_standard.log_spec_tot` file and plots the flobal SED in νL_ν units as a function of ν. This file can also be read using astropy but excludes the angle columns.

[5]:
```
import plot_tot
plot_tot.doit(fname, smooth)
```

```
The Created luminosity was  4.478638412507001e+34
The emitted luminosity was  3.90951228511005e+34
```



3.6.2 Plotting Wind Properties

As described under *Models*, Python saves wind properties in binary wind_save files. This notebook explains how to read and plot wind variables for the `cv_standard` file found in the examples. Before running the python commands, you need to run the model from the command line. I suggest running the following commands, after you have compiled python:

```
mkdir cv_test
cd cv_test
cp $PYTHON/examples/basic/cv_standard.pf .
py cv_standard </code>
```

The model will take about 5 minutes to run on a single core. It will not converge, but will give us a model to use as an example. You should then run `windsave2table` on the output

```
windsave2table cv_standard
```

which will create a series of ascii files containing key variables in the wind cells. We will use these ascii files for our plots.

Make A Basic Quick Look Wind Plot

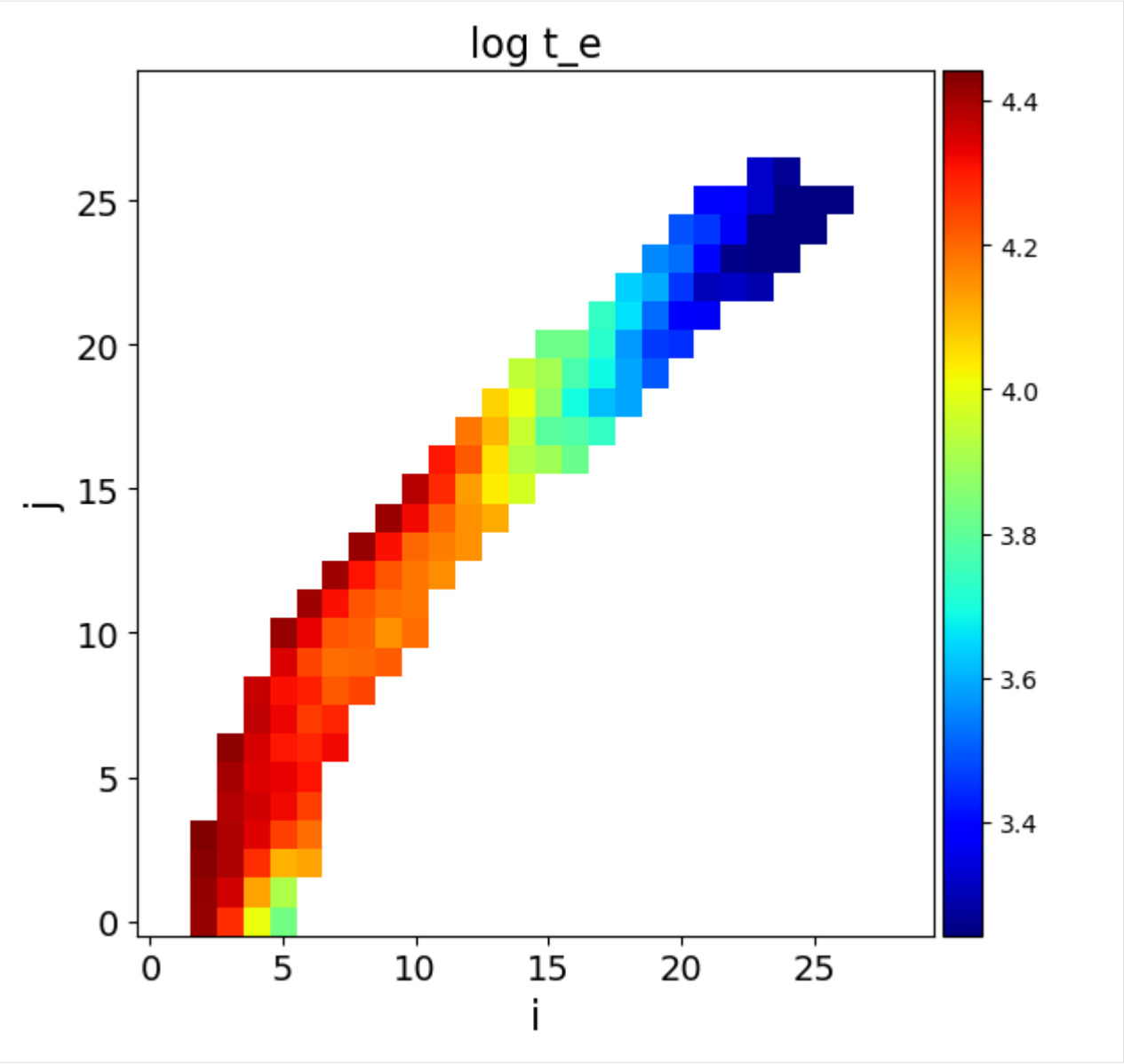
The simplest way to make a quick look plot of the electron temperature is using the `plot_wind.py` routine in `$PYTHON/py_progs`. In this example, I will assume py_progs has been added to `$PATH` and to `$PYTHONPATH`. `plot_wind.py` can be run from the command line using

```
plot_wind.py cv_standard t_e
```

where the second argument is the variable to plot. Alternatively, it can be run from within a python script by doing (where we are now assuming you are running this code from one directory above cv_test):

[5]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import plot_wind
fname = "cv_test/cv_standard.master.txt"
plot_wind.doit(fname, var="t_e")
```

[5]:
```
'cv_test/cv_standard_log_t_e.png'
```



More detailed/customisable plots

You may, however, wish to get more direct access to the data, which can be done easily by reading in the `cv_standard.master.txt` file, for example using `astropy`. In the next code block, we read in the data file and print out the columns.

[6]:
```
import matplotlib.pyplot as plt
import astropy.io.ascii as io

data = io.read(fname)

print (data.colnames)
```

```
['x', 'z', 'xcen', 'zcen', 'i', 'j', 'inwind', 'converge', 'v_x', 'v_y', 'v_z', 'vol',
↪'rho', 'ne', 't_e', 't_r', 'h1', 'he2', 'c4', 'n5', 'o6', 'dmo_dt_x', 'dmo_dt_y', 'dmo_
```

(continues on next page)

(continued from previous page)

```
↪dt_z', 'ip', 'xi', 'ntot', 'nrad', 'nioniz']
```

The py_plot_util script in py_progs comes with a handy guide to the main columns in the .master.txt file and returns a dictionary containing the description for all variables.

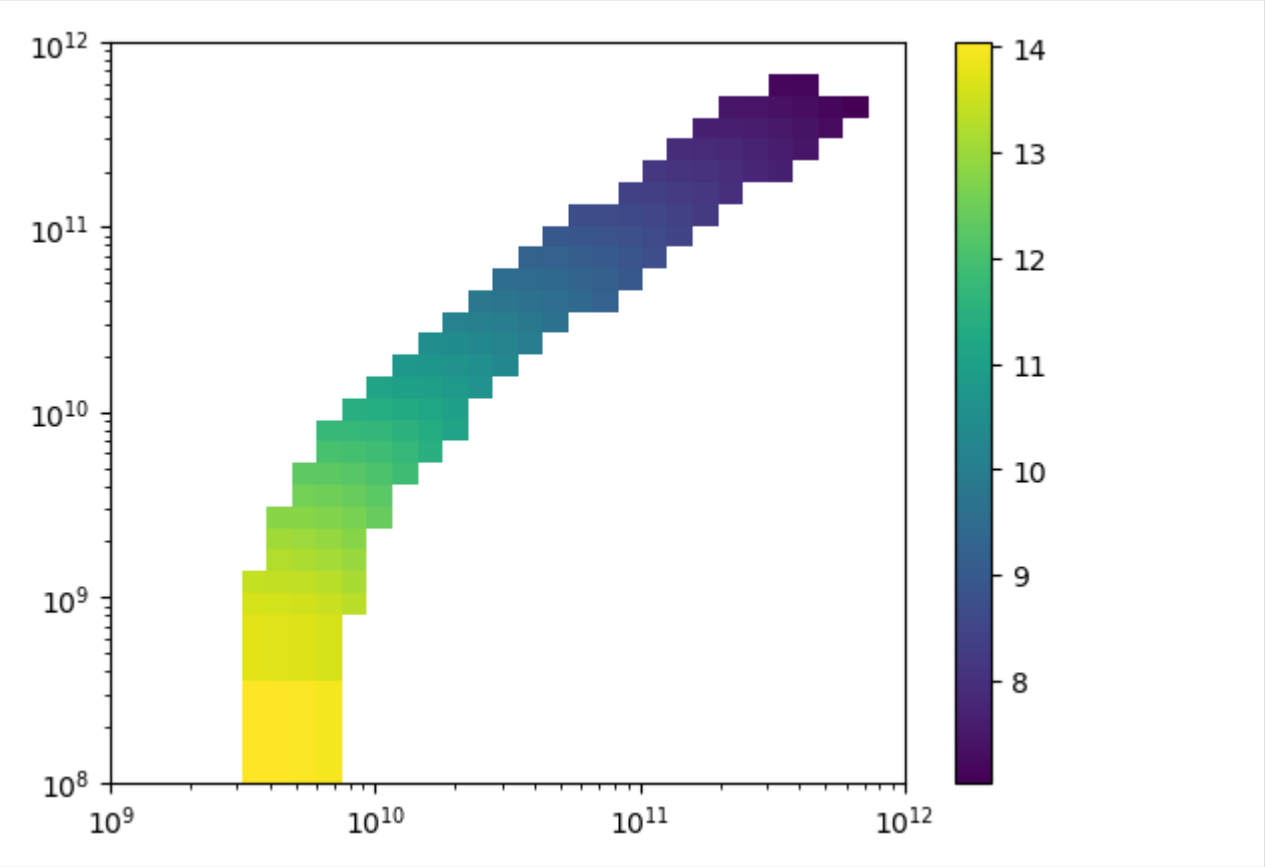
[7]:
```
import py_plot_util as util
descr = util.get_windsave_descriptions(data)
```

```
no description for column x
z           --  left-hand lower cell corner z-coordinate, cm
xcen        --  cell centre x-coordinate, cm
zcen        --  cell centre z-coordinate, cm
i           --  cell index (column)
j           --  cell index (row)
inwind      --  is the cell in wind (0), partially in wind (1) or out of wind (<0)
converge    --  how many convergence criteria is the cell failing?
v_x         --  x-velocity, cm/s
v_y         --  y-velocity, cm/s
v_z         --  z-velocity, cm/s
vol         --  volume in cm^3
rho         --  density in g/cm^3
ne          --  electron density in cm^-3
t_e         --  electron temperature in K
t_r         --  radiation temperature in K
h1          --  H1 ion fraction
he2         --  He2 ion fraction
c4          --  C4 ion fraction
n5          --  N5 ion fraction
o6          --  O6 ion fraction
dmo_dt_x    --  momentum rate, x-direction
dmo_dt_y    --  momentum rate, y-direction
dmo_dt_z    --  momentum rate, z-direction
ip          --  U ionization parameter
xi          --  xi ionization parameter
ntot        --  total photons passing through cell
nrad        --  total wind photons produced in cell
nioniz      --  total ionizing photons passing through cell
```

py_plot_util also contains some routines for reshaping and masking arrays and so on. One of the most useful for plotting is the wind_to_masked function which turns the raw 1D flattened data into a masked 2D array with the right shape which can be easily used with pcolormesh and so on. Here's an example plot of the electron density in the model.

[13]:
```
x, z, ne, inwind = util.wind_to_masked(data, value_string="ne", return_inwind=True)
plt.pcolormesh(x,z, np.log10(ne))
plt.loglog()
plt.xlim(1e9,1e12)
plt.ylim(1e8,1e12)
cbar = plt.colorbar()
```

```
/Users/matthewsj/.mpi_temp/ipykernel_33261/816639112.py:2: RuntimeWarning: divide by␣
↪zero encountered in log10
  plt.pcolormesh(x,z, np.log10(ne))
```



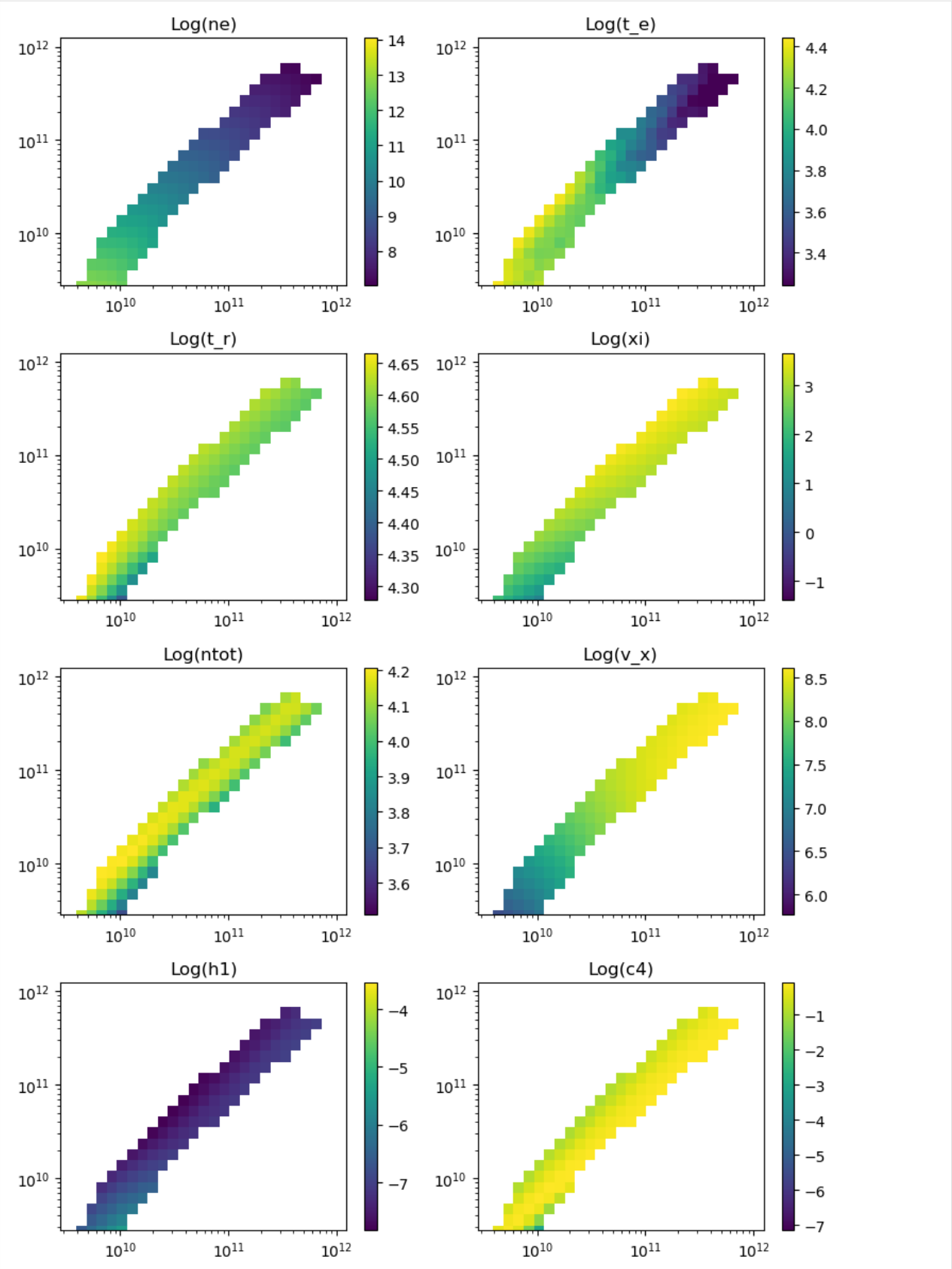
This procedure can be used to plot any of the variables in the masterfile and is a good starting point for delving into the properties of the wind.

To make a simple multi-panel plot of wind properties, you can use some of the routines in `py_plot_output`. The example below plots all the variables passed in an array and saves the file as `cv_standard_wind.png`

[18]:
```
import py_plot_output as plot
plot.make_wind_plot(data, "cv_standard_wind", var = ["ne", "t_e", "t_r", "xi", "ntot",
↪"v_x", "h1", "c4"], shape=(4,2) )
```

```
7363719978102.189
12772.992700729927
37305.83941605839
1214.8115635036497
11854.087591240876
145555532.84671533
5.267621167883212e-06
0.41242660718248175
```

[18]:
```
0
```

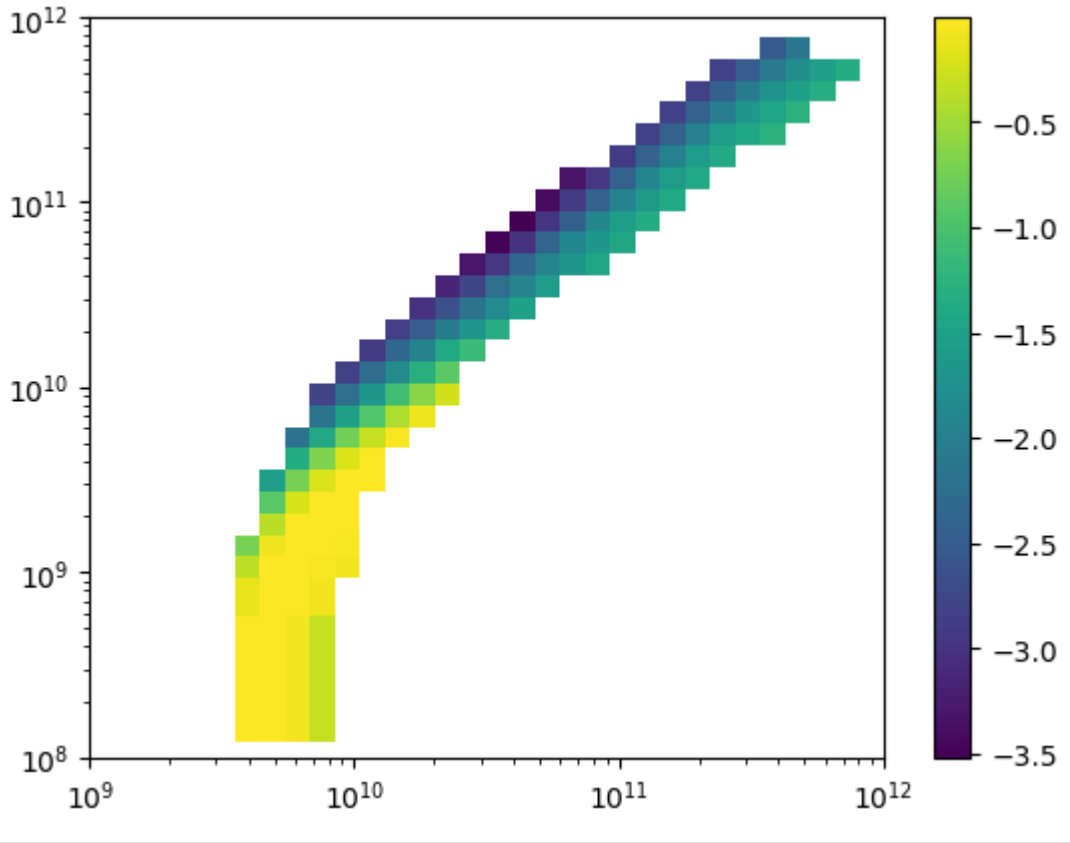

Log(ne)
Log(t_e)
Log(t_r)
Log(xi)
Log(ntot)
Log(v_x)
Log(h1)
Log(c4)

Plotting Ion Populations

Ion populations outputted from `windsave2table` are stored in files like `cv_standard.C.frac.txt`, where the letter before frac denotes the element. Plots of the C III ion fraction can thus be made through commands like the following, where strings like `i05` index the ion for each file.

[12]:
```
carbon_ion = io.read("cv_test/cv_standard.C.frac.txt")
x, z, c3_frac, inwind = util.wind_to_masked(carbon_ion, value_string="i03", return_
↪inwind=True)
plt.pcolormesh(x,z, np.log10(c3_frac))
plt.loglog()
plt.xlim(1e9,1e12)
plt.ylim(1e8,1e12)
cbar = plt.colorbar()
```

```
/Users/matthewsj/.mpi_temp/ipykernel_33261/4163463376.py:3: RuntimeWarning: divide by␣
↪zero encountered in log10
  plt.pcolormesh(x,z, np.log10(c3_frac))
```



[ ]:

3.7 Code Operation

The basic code operation of Python is split into different cycles; First, the ionization state is calculated (*Ionization Cycles*). As these photons pass through the simulation grid, their heating and ionizing effect on the plasma is recorded through the use of Monte Carlo estimators. This process continues until the code converges on a solution in which the heating and cooling processes are balanced and the temperature stops changing significantly (see *Convergence & Errors*). Once the ionization and temperature structure of the outflow has been calculated, the spectrum is synthesized by tracking photons through the plasma until sufficient signal-to-noise is achieved in the output spectrum for lines to be easily identified (*Spectral Cycles*).

3.7.1 Ionization Cycles

In order simulate a spectrum from a parameterized model of an outflow, one must first determine the ionization state of the wind. In order to accomplish this, one begins with a guess at the ionization structure, usually by setting the temperature of the wind at a specific value and assuming that the ionization equilibrium is simple given by the Saha equation for that particular temperature.

In Python, one then generates a set of photon bundles over a wide frequency range, and then causes these photons to pass through and interact via various processes with the wind. As the photons transit the wind, estimatores for various processes are accumulated, which characterize the intensity and spectrum of the radiation field in various parts of the wind, the amount of heating and rate at which ions are photoionized, etc.

Once all of these photon bundles have passed through the wind, one uses the various estimators to modify the ionization state and electron temperature in each cell, and then one repeats the process in order to try to find the actual state of the wind, given the assumed density and velocity field of the wind. There are a variety of approaches to carrying out this calculation and various limitations placed on the rate at which the plasma is is allowed to change between cycles. As the accuracy of any Monte Carlo simulation depends on numbers of photons bundles one uses to approximate the spectrum there are various options within Python to choose the number of photons with various energy/wavelength bins, and other options to begin with a smaller number of photons and increase this number in later cycles.

3.7.2 Spectral Cycles

The purpose of the ionization cycles is to establish the ionization state of the plasma. The purpose of the spectral cycles is to created simulated spectra given a defined ionization state in a wavelength range that is (usually) less wide than required to establish the ionization state. For a cataclysminc variable, One might, for example, want to create a simulated spectrum to compare with an observed spectrum of that object in the ultraviolet. which is observed with a specific inclination angle with respect to the disk.

For the simple atom case, the process is relatively straightforward. One begins by generating photon packets that cover a range that slightly boarder than the spectral range of interest, slightly broader because one needs to allow for Doppler effects associated with scatters that can occur. One then follows these photons through the wind, where the number of photons carred by the packet diminishes as it moves through the wind.

Then one could simply create a spectrum from photon packets that exit the simulation volume at a particular incination angle (plus or minus some delta) to construct the spectrum. This so-called live-or-die option is implemented in Python, but only as a diagnostic option, because it is inefficient since most photon packets exit the system at inclination angles that are not of interest.

Instead, the standard way of constructing detailed spectra is to use the method, termed the viewpoint technique, as described by Knigge, Woods, & Drew 1995, also known as the peel-off method (Yusef-Zedeh, Morris & White 1984. In this method, one follows a photon through the grid as before, but at point where the photon changes direction (including the inital point of photon generation), one creates a dummy photon headed in the desired direction. One adjust the weight of the dummy photon accoring to the relative probability that a photon will escape in the desired

direction compared to the angle averaged probability, and adjusts the number of photons by that fraction, that is

$$w_{\rm out} = \frac{P(\theta)}{\langle P \rangle} w_{\rm in}.$$

For isotropic scattering the $w_{\rm out} == w_{\rm in}$ but for resonant scattering the weight will increase if the desired photon direction is in the direction of maximum velocity gradient and decrease if it is along the direction of minumum velocity gradient (see *Anisotropic Scattering*). For photons generated at the surface of a star of disk, the weight of the dummy photon is determined by the limb darkening law assumed. One then extracts the dummy photon along the line of sight reducing the weight of the photon by the total optical depth along that lien of sight. Evidently, one can repeat this process at every interaction when one wishes to construct a spectrum along multiple lines of sight.

Detailed Spectral Calculation when Macro-atoms are used

When a macro-atom is excited, photon packets can emerge at very different frequencies than the frequency of the photon packet before an interaction. This requires a modification of the methods used during ionization cycles, where, in the macro-atom case, no photons or r-packets originate in the wind and a strict radiative equilibrium constraint is enforced (with a few exceptions, e.g. adiabatic cooling).

During the ionization cycles, the amounts of energy flowing into each macro-atom level, and into the thermal k-packet pool, are recorded in the matom_abs and kpkt_abs quantities. In the spectral cycles, one needs to know where this energy comes out - if energy flows into a given state, what proportion of that energy comes out via the various possible transitions? This issue is dealt with in the "macro-atom emissivity calculation", which is carried out at the start of the spectral cycles. The current procedure is to do a Monte Carlo sampling of the macro-atom machinery – a large number of packets are generated with initial macro-atom states in proportion to the estimators matom_abs and kpkt_abs. The fraction of times these packets de-activate from given states is then recorded, and the corresponding r-packet frequency is calculated. If the frequency falls within the requested range, the relevant macro-atom or k-packet emissivity is incremented by the appropriate fraction of matom_abs. If the frequency falls outside the range, the contribution is ignored. This procedure can be speeded up by using an implicit/matrix scheme where the matrix contains the mapping between the absorbed and emergent radiation; This method is currently in the development stage in the code.

In the actual photon transport stage, r-packets are generated in the wind in proportion with these frequency-limited emissivities, in a broadly similar to wind photon generation in the non-macro atoms scheme. In the process, we also ensure that the photons are only generated over the correct frequency range. The photon transport is then carried out as normal, except that whenever a macro-atom is activated, or a k-packet is created, that photon / energy packet is immediately thrown away to avoid double-counting – the emission resulting from the interaction has already been (statistically speaking) accounted for in the emissivity calculation. The main difference to the approach in the ionization cycles is that now the radiative equilibrium condition is enforced by the fact that the emissivities should be consistent with the "absorbed" radiation, instead of being explicitly enforced by never destroying packets.

When using indivisible packet line transfer (commonly referred to as macro-atom mode), the code is often used in a hybrid mode where some elements are treated as macro-atoms and some as simple-atoms. Simple atoms are treated differently to macro-atoms; there is no detailed model atom and internal transitions to other states are not possible. Instead, for both bound-free and bound-bound interactions, a fake two-level atom is excited, and the excited state either radiatively or collisionally decays. If it collisionally decays, this would normally excite a k-packet so the packet is destroyed for the same reasons as above (the emissivity from this is already accounted for). The k-packets generated from the emissivity are allowed to create simple-atom emission for consistency. If it radiatively decays, we treat the interaction like a resonant scatter and proceed, still tracking the packet. This is a reasonable approximation for resonant lines, but less so for bound-free continua, since the possible recombination cascade and potential reddening of the photons is not dealt with. Partially addressing this was the aim of the bound-free simple emissivity approach.

Other Issues

Spectral cycles are executed after the ionization and temperature state of the wind is computed by the ionization cycles. It is possible to modify the requested spectra (both wavelength range and observer angles) by making the changes to the spectral cycle parameters, setting the number if ionization cycles to zero, and then restarting the simulation.

3.8 Radiation Sources

Todo: Fill in. Add description of how to use your own input spectrum. Finish links to keywords.

3.8.1 External Radiation Sources

In generic terms, there are two main external radiation sources for any Python calculation: a **Central Source** which can be a normal star, a WD, or a BH, and a **disk**. Even though Python supports the existence of a secondary star for the purposes of calculating when light from a disk system is occulted, the secondary star does not radiate.

Photons for radiation from the central object emerge uniformly over its surface, except when a lamp-post geometry is specified for the `bh` or `agn` system types. In this lamp-post case, radiation originates from a point source above and below the central object, with a specified height.

Emission from a boundary layer can also be defined when this is relevant, from which radiation also emerges uniformly over the surface of the central object.

3.8.2 The Wind as a radiation source

In macro-atom calculations the wind is NOT a radition source. All of the photons in a macro-atom calculation are generated externally, and with minor exceptions photons preserve their weight throughout their passage through the wind. (The minor exceptions have to do with processes like adiabiatic cooling, which result in the loss of photons).

In the simple-atom approach, various processes cause photons passing through the wind to lose energy as they pass through the wind. This energy heats the plasma. To account for this, photons are generated from the wind at the beginning of each cycle. Processes include, free-free emission, free-bound emission and line emission.

In non-macro-atom calculations wind radiation can be turned on and off using the *Wind.radiation* keyword.

(In various files that contain the spectra there is a column WCreated that in the simple atom mode gives the spectrum of photons that were created in the wind. This column, also exists in the macro-atom case, where it records the spectrum of pbotons that have interacted with the wind and been re-emitted.)

3.8.3 Spectra of the external radiation sources

For the most part, the various radiation sources can radiate by any of the following process, as appropriate)

1. Blackbody radiation, specified in terms of a temperature. Depending on the nature of the source, the luminosity specified either by the size of the object, or directly as the total luminosity.
2. Bremsstrahlung radiation, specified in terms of a temperature and a luminosity between 2 and 10 keV
3. Power law radiation, specified in terms of a spectral index, and a luminosity between 2 and 10 keV

4. One or more spectral models read from a series of files. The models must specified in terms of two parameters, usually T and log g, each model consists of an ascii file containing the spectra. An example of the ascii files that can be read in is contained in the xdata folder that is part of the distribution (See below).

In the ionization cycles, the spectra of the central source, boundary layer (if present) and disk are determined by these three keywords:

- *Central_object.rad_type_to_make_wind*
- *Boundary_layer.rad_type_to_make_wind*
- *Disk.rad_type_to_make_wind*

It is possible to choose different input spectra for the ionization and spectral cycles, so a corresponding keyword of the form *Disk.rad_type_in_final_spectrum* is also needed.

3.8.4 Spectra from a model grid (details)

Python was initially written to model the winds of cataclysmic variables (CVs). Although the spectra of the disks of cataclymic variables are often modelled in terms of blackbodies, the spectra of CVs show clear evidence of features that arise from the i disk (as well as the wind). The features that arise from the disk resemble in many respects those that arise from an appropriately weighted set of stellar atmospheres. To allow for this possibility, Python can be configured to read a set of models characterized by a temperature and log g, and produce spectra of either the central object or the disk by interpolating on t and log g. The data that must read in consists of a file that associates a temperature and log g with the indvidual spectra.

For example, as part of the standard distruction there is a file kurucz91.ls, which starts as follows

```
data/kurucz91/fp00t3500g00k2c125.txt          3500          0.0
data/kurucz91/fp00t3500g05k2c125.txt          3500          0.5
data/kurucz91/fp00t3500g10k2c125.txt          3500          1.0
data/kurucz91/fp00t3500g15k2c125.txt          3500          1.5
data/kurucz91/fp00t3500g20k2c125.txt          3500          2.0
data/kurucz91/fp00t3500g25k2c125.txt          3500          2.5
data/kurucz91/fp00t3500g30k2c125.txt          3500          3.0
data/kurucz91/fp00t3500g35k2c125.txt          3500          3.5
data/kurucz91/fp00t3500g40k2c125.txt          3500          4.0
data/kurucz91/fp00t3500g45k2c125.txt          3500          4.5
data/kurucz91/fp00t3500g50k2c125.txt          3500          5.0
data/kurucz91/fp00t3750g00k2c125.txt          3750          0.0
...
```

In this case we have spectra at a temperature of 3500, for 11 different values of log g, before going on to temperature of 3750 K. Each spectrum is one of the Kurucz models, and these contain entries which contain a set of wavelengths and a quantity that is understood to be proportional to F_λ.

The 3 column format above is required. If one wants to use a set of models that have only a T parameter one should simply choose a value for the second column. The use case here is fairly specific, especially with regard to the first parameter T. If the disk or central object temperature outside the temperatures in the grid, then Python will "adjust" the spectrum assuming that the overall spectrum changes as a BB would, but the features in the spectrum are unchanged. If the gravity goes outside the range of the grid, the closest value is chosen.

One need not use Kurucz models, of course. Any set of models can be used, as long as the files contain two columns, a wavelength in Angstroms and something that is proportional to F_λ. The normalization of the fluxes does not matter, because the models are only used to establish the shape of the spectrum. The normalization is determined by the total luminosity of the component.

Photon Banding Strategies

Photon packets are emitted from a number of different radiation sources, such as the accretion disk or from the wind itself. When a photon is created, it is defined by its frequency ν and weight w. Photons are generated at the beginning of each cycle and can either be generated uniformly over the entire frequency range, or can be generated in pre-defined frequency bands, where certain frequency bands are biased to have more photons.

Uniform Sampling

In the most simple case, the frequency of a photon is sampled uniformly over the entire frequency range. The total weight of all photons is equal to the luminosity of the system and each photon has weight has equal weight given by,

$$w_i = \frac{\sum_{i=\text{sources}} \int_{\nu_{\text{min}}}^{\nu_{\text{max}}} L_{\nu,i} d\nu}{N},$$

where N is the total number of photons, ν_{min} and ν_{max} define the frequency range and $L_{\nu,i}$ is the luminosity for a radiation source. Note that a summation is used to find the luminosity for each radiation source and that w has units of ergs s^{-1}.

Banded Sampling

In practice, uniform sampling is generally an inefficient approach to generating photon packets. For example, it is often desirable to produce a sufficient number of photons within a specific frequency range, i.e. around a photoionisation edge. However, if these frequencies lie on the Wien tail of a blackbody distribution, it is unlikely that a sufficient number of photons will be generated as most of the luminosity is generated at lower frequencies. It is possible to get around this problem by generating an increasingly large number of photons. But, this is computationally expensive and inefficient.

In order to cope with cases this, Python implements *importance sampling* which effectively increases the number of photons which are sampled from specific frequency bands considered important. Photons are now generated with the weight,

$$w_j = \frac{\sum_{j=\text{sources}} \int_{\nu_i}^{\nu_{i+1}} L_{\nu,j}\ d\nu}{f_i N},$$

where, again, this is a summation over all radiation sources. N is the total number of photons, f_i is the fraction of photons emerging from frequency band i, ν_i and ν_{i+1} are the lower and upper frequency boundaries for each frequency band and $L_{\nu,j}$ is the luminosity of the radiation source. Hence, more photons from frequency bands with a larger fraction f_i will be generated. However, photons from *important* bands (where more photons are sampled from) will have reduced weight, whilst photons from the *less important* frequency bands will have increased weight.

This scheme has the benefit of allowing the generation of a lower number of photons whilst still sufficiently sampling important frequency ranges, decreasing the computational expense of a simulation. As such, this is the preferred sampling method.

Available Sampling Schemes

Python currently implements seven pre-defined frequency bands and and two flexible *run time* banding schemes. The parameter used to define the photon sampling scheme is,

`Photon_sampling.approach(T_star,cv,yso,AGN,min_max_freq,user_bands,cloudy_test,wide, logarithmic)`

Minimum and Maximum Wavelengths

At present, the largest wavelength a photon can be is hardwired to 20,000 Angstroms. The smallest wavelength a photon can take is defined by the temperature of hottest radiation source, but is at least 115 Angstroms - twice that of the Helium edge.

T_star

Create a single frequency band given a temperature T, which is the temperature of the hottest radiation source in the model. All photons will then be generated from this single frequency band.

CV

Pre-defined bands which have been tuned for use with CV systems, where a hot accretion disk (~100,000 K) is assumed to exist. In this scheme, there are four bands where the majority of photons are generated with a wavelength of 912 Angstroms or less.

YSO

Pre-defined bands which have been tuned for use with YSO systems. In this scheme, there are four bands.

AGN

Pre-defined which have been tuned for use with AGN system. In this scheme, there are ten bands, with a minimum frequency of 1×10^{14} Hz and a maximum frequency of 1×10^{20} Hz.

min_max_freq

Create a single band using the minimum and maximum wavelength as described by the minimum and maximum wavelengths calculated for the current model.

user_bands

This allows a user to create their own frequency bands, defined by photon energies measured in electron volts. The first band has the lowest photon energy and each subsequent band must have a larger energy than the previous band. Each band also requires a minimum fraction of photons to be sampled from this band, where the sum of the fractions for each band must be equal to or less than one.

Maximum Number of Bands

Currently, a maximum of 20 frequency bands can be defined. If a user attemps to specify more than than 20 bands, Python will create an error message and fallback to using 20 bands.

cloudy_test

This set of bands were created for use in testing against the photoionisation and spectral synthesis code Cloudy.

wide

Pre-defined bands which have very wide frequency range. The purpose of this band is for testing, hence is best to avoid using this band for a working model.

logarithmic

This is the same as `user_bands`, however the frequency bands are now defined in log space. This allows one to better sample a frequency range which spans many orders of magnitude.

Maximum Number of Bands

Currently, a maximum of 20 frequency bands can be defined. If a user attemps to specify more than than 20 bands, Python will create an error message and fallback to using 20 bands.

Minimum Fraction

For logarithmic user defined bands, the fraction of each band is set to 1 / nbands.

The Disk

The disk is normally treated as infinitely thin and defined by an inner boundary and an outer boundary. It assumed to be in Keplerian rotation about the central object in the system. The temperature distribution of the disk is normally assumed to be that of a standard Shakura-Sunyaev disk, with a hard boundary at its inner edge. Options are provided for reading in a non-standard temperature distribution.

An option is provide for a vertically extended disk, whose thickness increases as with distance from the central object object.

The parameters involved in describing a flat disk are:

```
Disk.type(none,flat,vertically.extended)                    flat
Disk.radiation(yes,no)                              yes
Disk.rad_type_to_make_wind(bb,models,mod_bb)                        bb
Disk.temperature.profile(standard,readin)                 standard
Disk.mdot(msol/yr)                            5
Disk.radmax(cm)                               1e17
```

Colour Correction (mod_bb)

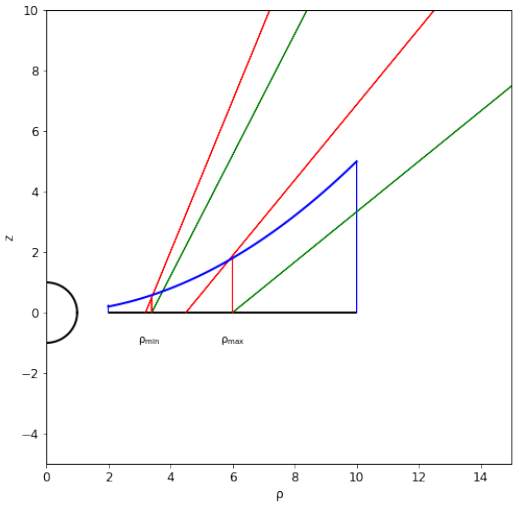
A simple form of the disc colour correction is available in the code, accessible via the Disk.rad_type_to_make_wind(bb,models,mod_bb) keyword. The colour correction factor, $f_{\rm col}$, is defined such that

$$B_\nu(\nu, T) \rightarrow f_{\rm col}^{-4} B_\nu(\nu, f_{\rm col} T).$$

This correction is designed to approximate the effect of radiative transfer in the disc atmosphere. We adopt the form given by Done et al. 2012, in which $f_{\rm col} = 1$ for $T < 3 \times 10^4$ K, and for $T > 3 \times 10^4$ K

$$f_{\rm col}(T) = \left(\frac{T}{3 \times 10^4 \ {\rm K}} \right)^{0.82}.$$

Vertically Extended disk (Details)



The figure above explains the basics issues associated with a vertically extended disk. The wind emerges from the actual disk between ρ_{min} and ρ_{max}.

In defining a vertically extended disk in the context of parameterized models, such as KWD of SV, one needs to decide how to tranlated values from a parameterized wind on a flat disk to a parameterized wind on verticallye extended disk. The choices we have made are (intended to be) as follows:

- The temperature and luminosity of a vertically extended disk are given by the distance from the central object in the disk plane.

- The density at the base of the wind is defined as the same as the flat disk that underlies it.
- The poloidal (and rotational) velocity at the footpoint is the poloidal velocity along the streamline, starting with v at the actual surface of the disk.
- For the SV model, the streamline direction and velocity are determined by the distance from the central object along the disk plane. This is not the same as one would obtain by projecting the streamline back to the disk plane.
- For the KWD disk, stream line directions that reflect the focus position and the poloidal velocity ate taken from that expected by projecting the stream line back to the disk plane.

(Note that the in the KWD case, there is a slight inconsistency/inaccuracy in calculating desired mass loss rates, because the mass loss rate is calculate as if the disk were flat, but the stream line directions are not exactly the same as due to the vertical extension of the disk. There are also issues more generally because we do not take into account the fact that the disk area of a vertically extended disk is not exactly the same as that of a flat disk.)

Non-Standard Temperature Profile

If desired the user can read the temperature profile for the disk from a file. Each line in the file should consist of a radius (in cm) and a temperature (in K), and optionally a value of $\log g$. The values separated by whitespace (in the first two columns). The values are assumed to be entered in a logical order, that is in ascending values of radius. Lines such as comments or header names of an astropy table, will be ignored.

The $\log g$ value is not required to generate BB spectra, but is required if the spectrum from the disk is to be generated from a two-dimensional grid of models, usually a set of spectra generated to represent the spectra from a set of stellar atmospheres calculations.

With this option, the radius of the disk will be set to the maximum radius (the last value of r) in the file.

3.9 Wind Models

python has a series of different wind models available, including parameterised wind models and the ability to import models. The links below detail each of the models available.

The actual Model parameters in the input file are also described under *Wind Models*.

3.9.1 SV93 biconical wind prescription

In the SV93 prescription, the wind emerges between r_{min} and r_{rmax} along streamlines whose orientation with respect to the system are described an angle

$$\theta = \theta_{min} + (\theta_{max} - \theta_{min})x^{\gamma}$$

where

$$x = \frac{r_o - r_{min}}{r_{max} - r_{min}}$$

and r_o refers to the footpoint of a streamline.

The poloidal velocity along the streamlines is defined to be

$$v_l - v_o + (v_\infty(r_o) - v_o)\frac{(l/R_v)^\alpha}{(l/R_v)^\alpha + 1}$$

The scale length R_v and the exponent α control the acceleration of the wind between a velocity v_o, at the base of the wind and the terminal velocity $v_\infty(r_o)$. The initial velocity v_o can be set to either a constant, normally 6 km/s, or a

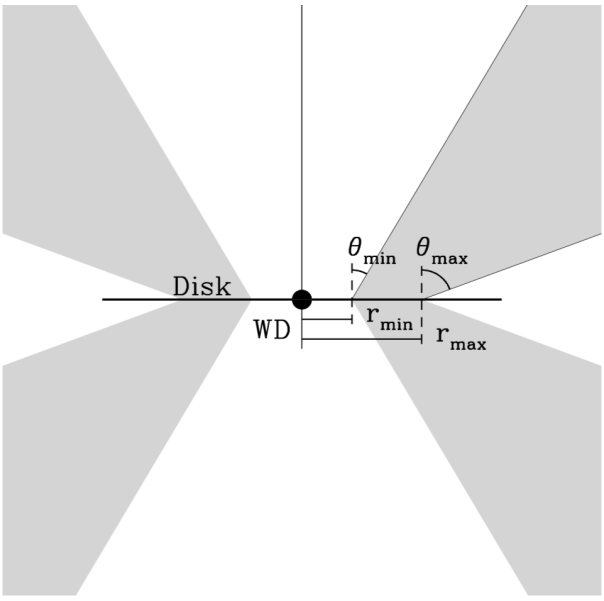


Fig. 2: The geometry of a Shlosman & Vitello wind

multiple of the sound-speed at the streamline base. The terminal velocity of each streamline varies depending on the location of the streamline in the inner and outer disk, being characterized as a fixed multiple of the escape velocity at the footpoint of the streamline. Thus the poloidal velocity is greatest for stream lines that originate from the inner regions of the disk, since the gravitational potential that must be overcome is greatest there.

The mass loss per unit surface area $\delta\dot{m}/\delta A$ of the disk is controlled by a parameter λ such that

$$\frac{\delta\dot{m}}{\delta A} \propto \dot{m}_{wind} r_o^{\lambda} cos(\theta(r_o))$$

With this prescription, the overall mass loss rate declines with radius if λ is somewhat less than -2.

To use the SV93 prescription, therefore, one must provide the basic parameters of the system, the mass of the WD, the accretion rate, the inner and outer radius of the disk, and in addition, for the wind $\dot{m}_{wind}$, r_{min}, r_{max}, θ_{min}, θ_{max}, γ, R_ν, α, λ, and the multiple of the escape velocity to be used for v_∞.

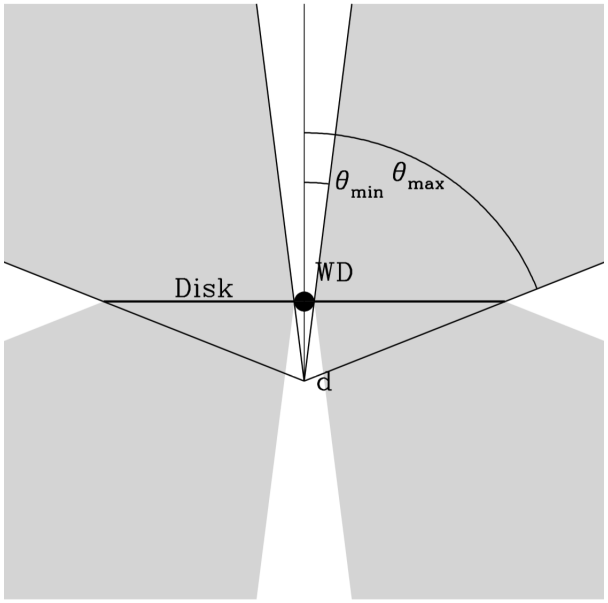
The following variables are used:

```
Wind.mdot(msol/yr)                          1e-9
SV.diskmin(units_of_rstar)                  4
SV.diskmax(units_of_rstar)                  12
SV.thetamin(deg)                            20
SV.thetamax(deg)                            65
SV.mdot_r_exponent                          0
SV.v_infinity(in_units_of_vescape           3
SV.acceleration_length(cm)                  7e10
SV.acceleration_exponent                    1.5
SV.v_zero_mode(fixed,sound_speed)                   fixed
SV.v_zero(cm/s)                                             6e5
```

3.9.2 KWD biconical wind prescription

Knigge, Woods & Drew (1995) developed a parameterization for a bi-conical flow, which in slightly modified form is built into Python. in this parameterization, the wind is envisioned to have poloidal streamlines that all point to a position a distance d below the disk, as is shown below:

Todo: This figure needs modification to account for the fact that we allow rmin and rmax to be specified.



As descried by KWD95, streamlines emerge thoughout the entire disk, with the innermost streamline just grazing the surface of the central object and the outermost streamline emerging from the outer radius of the disk. In the current version of Python, while this is the default choice, the wind region can be restricted to streamlines that arise from between r_{min} and r_{rmax}. For fixed values of r_{min} and r_{rmax}, the wind will tend to be more collimated the larger the value of d.

In the KWD parameritization, the mass loss per unit area per unit area of the disk is given by

$$\frac{\delta \dot{m}}{\delta A} \propto T(R)^{4\alpha}$$

where T(R) is the temperature of the disk at radius R. With this parameterization, the mass loss rate per unit area is constant for $\alpha = 0$ and is porportional to the luminoous flux is $\alpha = 1$.

The KWD95 model incorporates a velocity law reminiscent of a stellar wing, viz.

$$v_l = (nc_s) + (v_\infty - nc_s)\left(1 - \frac{R_v}{l + R_v}\right)^\beta$$

where nc_s is the speed at the base of flow,a multiple of the sound speed, R_v is the scale length, $beta$ is the exponent that determines the number of scale lengths over which the wind acclerates, and v_∞ is defined as a multiple of the escape velocity at the footpoint of each stream line.

For the model, the sound speed of the disk is defined to be

$$c_s(R) = 10\sqrt{\frac{T_{eff}(R)}{10^4 K}} kms^{-1}$$

The variables that must be defined are as follows:

```
Wind.mdot(msol/yr)        1e-9
KWD.d(in_units_of_rstar)        16.0
KWD.mdot_r_exponent        1.0
KWD.v_infinity(in_units_of_vescape)        3.0
KWD.acceleration_length(cm)        10000000000.0
KWD.acceleration_exponent                        1.5
KWD.v_zero(multiple_of_sound_speed_at_base)                        1
KWD.rmin(in_units_of_rstar)                        1
KWD.rmax(in_units_of_rstar)                    55.6329
```

3.9.3 The homologous wind model

In the homolgous model the wind/outflow is assumed to have spherical symmetry and to have a particularly simply velocity law: specifically, homolgous expansion

$$v \propto r$$

This sort of velocity law has the advantage of being very simple to work with, and is generally a good approximation for supernovae.

In pracise, the outflow (wind) is assumed to extend from an inner radius $r_{\rm min}$ to an outer radius $r_{\rm rmax}$. The physical idea here is not necessarily that the wind stops at the maximum radius, but rather that it is sufficiently dilute that spectrum formation beyond this point becomes unimportant.

In our implementation, the specifics of the velocity law are determined by giving the outflow speed at $r_{\rm min}$ via a parameter $v_{\rm base}$, keyword `Homologous.vbase`. It then follows that the velocity at all other points in the wind is

$$v = v_{\rm base} \frac{r}{r_{\rm min}}$$

The density in the wind is determined by setting a mass flux at the inner boundary ($boundary\_mdot$, in solar masses per year). The variation of the density at larger radii is the controlled by an exponent (β), keyword `Homologous.density_exponent` such that

$$\rho \propto r^{-\beta}$$

3.9.4 The stellar wind model

The stellar wind models implements the common Caster & Larmers velocity law , where

$$v(r) = V_o + (V_\infty - V_o)(1 - R_o/r)^\beta$$

Evidently, if β is 1, then the velocty will expand uniformly between V_o and V_∞

3.9.5 Importing Models

Python can read 1D or 2.5D grids of density and velocity, instead of setting up the model from an analytic prescription. Caution should be exercised with this mode, as it is still in a development phase, and the mode requires the user to ensure that things like mass and angular momentum conservation are enforced.

This mode is activated via wind type option “imported”, which triggers an extra question, e.g.

```
Wind.type(SV,star,hydro,corona,kwd,homologous,shell,imported)            imported
Wind.coord_system(spherical,cylindrical,polar,cyl_var)          cylindrical
Wind.model2import                    cv.import.txt
```

An example in cylindrical geometry, `cv_import.pf`, is given with a supplementary grid file in `examples/beta/`. The format expected in the grid input file for such a cylindrical model is as follows, although the column headers lines are actually not read.

```
i  j  inwind    x      z      v_x   v_y   v_z     rho   t_e    t_r
-- -- ------  -----  -----  -----  -----  -----  -----  -----  -----
0  0    -1    1.4e9  3.5e9   0.0   0.0   6e5     0.0   0.0    0.0
0  1     0    1.4e9  3.5e10  1e5   0.0   2e6     1e9   0.0    0.0
```

where all physical units are CGS. i and j refer to the rows and columns of the wind cells respectively, while inwind tells the code whether the cell is in the wind (`0`), or out of the wind (`-1`). If a partially in wind flag is provided (`1`), the code defaults to treating this cell as not in the wind. This could in principle be adapted, but means that for the moment this mode is most useful when using models with sufficiently high resolution or covering factors that partially in wind cells are unimportant.

The other input files have slightly different formats. The best way to see the format is use the process described at the end of the page.

Creating your own model

In order to create your own model, there are a few important things to consider:

- all units should be CGS (except for indices and flags, which are integers)
- x and z for cylindrical (or r and theta for spherical polar) coordinates are supplied at the edges, rather than centres, of cells. Thus, a given cell is described by the location of it's bottom left hand corner in (x,z) space.
- Ghost cells **must** be included. This means that additional rows and columns of cells must be included at the edges of the grid, and they must be excluded from the wind so that their temperatures and densities are set to zero, but have a velocity that python can interpolate with.
- i and j correspond to rows and columns respectively, so that the first row of cells at the disk plane has i = 0.
- rho the density of the cell in cgs units
- The t_e and t_r columns are optional and correspond to the electron and radiation temperature

Although `cv_import.pf` is designed to closely match the `cv_standard.pf` model, it does not match the model perfectly as the imported model does not deal with 'partially in wind' cells. As such, we generally recommend imported models are used for either wind models that entirely fill the grid or that have sufficiently high resolution that the partial filled cells are relatively unimportant.

Spherical Grids

Using a spherical coordinate system, a 1D spherically symmetric model can be read into Python.

To read in a grid of this type, the following columns are required for each cell:

- i : the element number for each cell
- r : the radial coordinate in CGS
- v_r : the radial velocity in CGS

- ρ : the mass density in CGS
- T_e (optional) : the electron temperature in Kelvin
- T_r (optional) : the radiation temperature in Kelvin

Grid Coordinates

The radial coordinates of the cells must be constantly increasing in size.

Cylindrical Grids

Using cylindrical coordinates, a 2.5D model can be read into Python.

Grid Coordinates

Note that the grid coordinates and the velocity is specified in Cartesian coordinates.

To read in a grid of this type, the following columns are required for each cell:

- i : the i element number (row)
- j : the j element number (column)
- inwind : a flag indicating whether the cell is in the wind or not
- x : the x coordinate in CGS
- z : the z coordinate in CGS
- v_x : the velocity in the x direction in CGS
- v_y : the velocity in the y direction in CGS
- v_z : the velocity in the z direction in CGS
- ρ : the mass density in CGS
- T_e (optional) : the electron temperature in Kelvin
- T_r (optional) : the radiation temperature in Kelvin

Unstructed/non-linear Grids

In principle, it is possible to read in an unstructured or non-linear cylindrical grid, i.e. where the cells are not regularly spaced, however, Python has been designed for structured grids with regular grid spacing, and as such there may be undefined behaviour for unstructured grids.

Polar Grids

Using polar coordinates, a 2.5D model can be read into Python.

Cartesian Velocity

The velocity in for the polar grid is required to be in Cartesian coordinates due to conventions within the Python programming style. As such, any polar velocity components must first be projected into their Cartesian equivalent.

- i : the i element number (row)
- j : the j element number (column)
- inwind : a flag indicating whether the cell is in the wind or not
- r : the radial coordinate in CGS
- θ : the θ coordinate in degrees
- v_x : the velocity in the x direction in CGS
- v_y : the velocity in the y direction in CGS
- v_z : the velocity in the z direction in CGS
- ρ : the mass density in CGS
- T_e (optional) : the electron temperature in Kelvin
- T_r (optional) : the radiation temperature in Kelvin

θ-cells

The θ range should extend from at least 0 to 90°. It is possible to extend beyond 90°, but these cells should not be inwind and should be reserved as ghost cells.

Setting Wind Temperatures

Reading in a temperature is optional when importing a model. However, if one temperature value for a cell is provided, then Python assumes that this is the electron temperature and the radiation temperature will be initialised as,

$$T_r = 1.1T_e.$$

However, if two temperature values are provided for the cells, then the first temperature will be assumed as being the electron temperature and the second will be the radiation temperature.

If no temperature is provided with the imported model, then the radiation temperature will be initialised using the parameter, e.g.,

Wind.t.init 40000

The electron temperature is then initialised using the Lucy approximation,

$$T_e = 0.9T_r$$

Ghost Cells and Setting Values for *inwind*

The *inwind* flag is used to mark if a grid cell is either in the wind or not in the wind. The following enumerator flags are used,

```
W_IGNORE      = -2   // ignore this grid cell
W_NOT_INWIND  = -1   // this cell is not in the wind
W_ALL_INWIND  =  0   // this cell is in the wind
```

Whilst it is possible to set in *inwind = 1* for a grid cell, that is that the cell is partially in the wind, Python will instead set these cells with *inwind = -2* and ignore these grid cells.

Spherical

Three guard cells are expected. One guard cell is expected at the inner edge of wind and two are expected at the outer edge of the wind. Guard cells should still have a velocity, but the mass density and temperatures should be zero.

Cylindrical

For cylindrical grids, the outer boundaries of the wind should have two layers of guard cells in the same way as the a spherical grid, as above. For these cells, and all cells which do not make up the wind, an inwind value of -1 or -2 should be set.

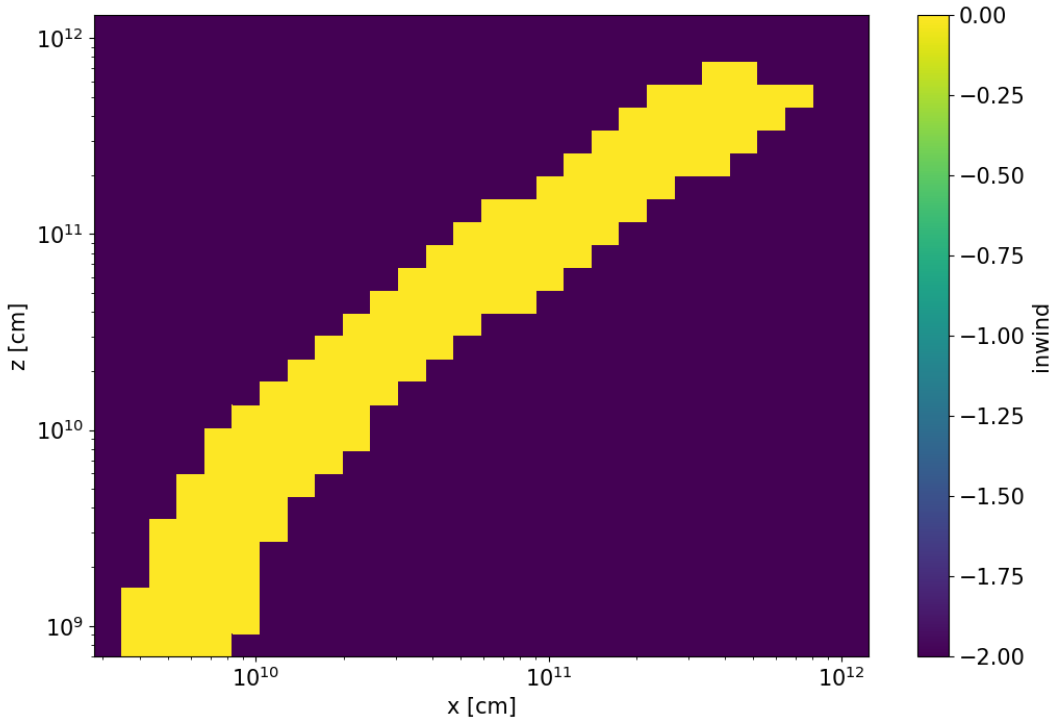


Fig. 3: A colour plot of the inwind variable for the cv_standard.pf example. Here, a SV model is being imposed on a cylindrical coordinate grid.

Polar

For polar grids, the outer boundaries of the wind should have two layers of guard cells in the same way as the a spherical grid, as above. For these cells, and all cells which do not make up the wind, an inwind value of -1 or -2 should be set.

In this example, the theta cells extend beyond 90°. But, as they are not inwind, Python is happy to include these cells. For a stellar wind in polar coordinates, these extra θ cells extending beyond 90° are required.

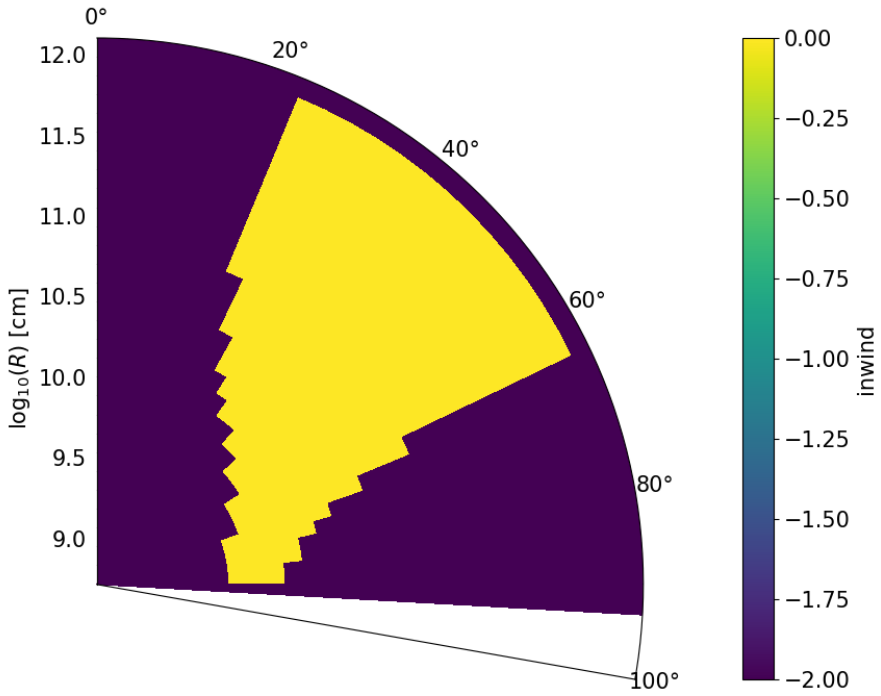


Fig. 4: A colour plot of the inwind variable for the rtheta.pf example. Here, a SV model is being imposed on an polar coordinate grid.

Maximum and Minimum Wind Radius

The maximum and minimum spherical extent of the wind is calculated automatically by Python, and does not take into account guard cells when it is doing this.

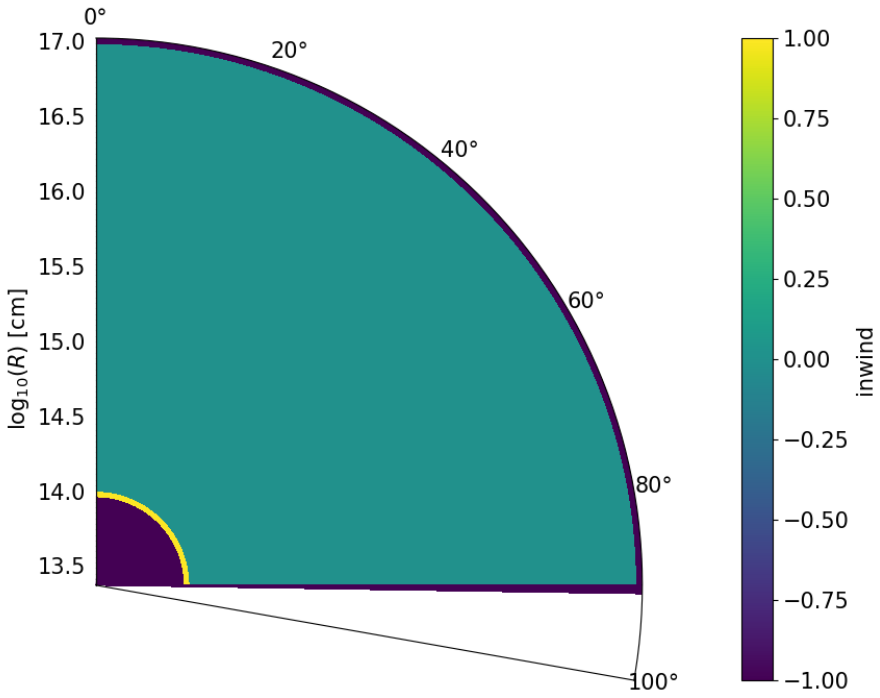


Fig. 5: A colour plot of the inwind variable for a stellar wind imposed on a polar coordinate grid. Important to note is the "halo" of inwind = -1 cells surrounding the inwind cells. The cells with inwind = 1 will be set to inwind = -2 when imported into Python and ignored.

Generating example inputs for testing and familiarizing oneself with Python's import capability

If one is trying to use the import capability of Python for the first time, it will be useful to familiarize oneself with the process, and the file format for a particular coordinate system, by running first running Python on a model that is something similar to model to be imported, but which takes advantage of one of the kinematic models available with the code.

For example, suppose you have a hydrodynamical simulation of an AGN wind which is in polar coordinates and you want to use Python to calculate the spectrum. Then you might create a model of an AGN with a similar coordinate system using, say, a Knigge Wood & Drew wind (and similar atomic data). For specificity, suppose this model has the root name "test"

Once you have run the model, you can create an import file file by first running the routine `windsave2table`, or more specifically:

```
windsave2table test
```

This produces a large number of ascii tables, which are described elsewhere

In the py_progs directory, you will find 3 scripts, `import_1d.py`, `import_cyl.py` and `import_rtheta.py`, which will convert one of the output files `test.0.master.txt` to an import file, `test.import.txt`, that can be used with the import mode of Python. The 3 different routines are for 1d spherical coordinates, and polar (r-theta) coordinates respectively.

Assuming the py_progs directory is in your PATH, and given that our example is for cylindrical coordinates, one would run:

```
import_cyl.py test
```

At that point, you can test this import file, by modifying the first .pf file to import mode (imported). Running Python on this file, will result in your being asked the name of the import file, and give you a "baseline" to import the hydrodynamical simulation to work.

Note that one should not assume that spectra produced by the original run of Python and the run of the imported model will be identical. There are several reasons for this:

First, in creating the original model, Python accounts for the possibility that some cells are partially in the wind. This is not possible in the imported models. Only cells that are complete in the wind are counted.

Second, within Python, positions and velocities are assumed defined at the corners of cells, whereas densities are assumed to be cell centered. If one provides a table where all of the quantities are at the same exact position (namely density is at the same position as x), there will be a slight discrepancy between the way in model as calculated internally and as represented within Python.

Todo: Combine with *Wind Models*?

3.10 Coordinate grids

Python supports 3 main coordinate gridding schemes, as well as one that is tailored to handle vertically extended disks. These schemes are

- 1-d spherical
- 2-d cylindrical
- 2-d polar or r-theta coordinates

These options are controlled by the *Wind.coord_system* keyword. For vertically extended disks, a modified version of the a cylindrical schme is provided where the cells viewed along the x,z plane are parallelograms, so that the grid follows the disk surface.

Although Python incorporates several models, such as the SV model for disk winds, that are continuous, the velocities and other proporites are placed on a grid, as part of the setup that goes on at the beginining of program execution.

It is up to the user to choose an appropriate coordinate system and the number of grid points to include in any particular run of the program.

As implemented within Python, the cells are created with a logarithmic spacing, that is the cells are larger the further they are from the central source (or disk plane). The one exception to this is that for polar coordinates, the angular separation of cells is fixed. For imported models, on the other hand, the user sets the exact coordinate gridding.

Obviously, the larger the coordinate grid, 100 x 100, say, compared to 30 x 30, the better the grid reflects a model. Equally obviously, the larger the coordiante grid, the larger the amount of memory the program will consume, and the larger the amount of computer time the program will take to run to completion. The increased computing is largely associated with the fact that one needs a good representation of the spectral energy distribution in each cell in order to properly calculate the ionization state in each cell.

Although the amount of memory for particular model generaly scales with the size of the grid, different 100 x 100 models, can consume very different amounts of memory. This is because for the KWD and SV parameterizations, the wind does not fill all of space. What really matters is the numbers of cells that are in the wind, because these are the cells for which all of the information about plasma conditions and the radition field needs to be maintained. So a wide angle wind with a 100 x 100 grid can take much more memory than a narrow angle wind on the same grid.

It is the number of cells that are actually in the wind that determine the fidelity of the model.

3.10.1 Partial cells

As note above, parameterized models often have region of space that are in the wind and regions whch are not. If one overlays, a coordinate grid on such a model, there will be cells that cross edges of the wind. These partial cells present particular problems.

In Python, velocities are interpolated on the corners of wind cells, but densities are are calculated based on the average radiation field in a cell, and hence ion densities are actually cell centered. As photons pass through a cell, they encounter resonances and the actuall opacities are based on an interpolated value of the densities. This presents no particular problem in regions inside the wind, but it is an issue for partial cells.

Currently, by default these cells are excluded by the calculation, and the densities of these cells are set to zero. Because of densities are interpolated this affects the first cell that is completely in the wind.

There are two other alternatives:

- The partial cells can be included in the calculation. This is reasonable for the KWD model, which has a velocity law that is easy to extend out side the wind region, but is less valid for the SV model, where this is not the case. For the KWD model, the only issue with including partial cells is that they are “smaller” than cells which are fully in the wind, and as a result are less likely to converge as adjacent cells that are fully in the wind.

- As an advanced option, the partial cells can be excluded, but instead of setting the density of the partial cell to zero, one can assign it the density of the "nearest" cell that is totally in the wind. In this case the ionization balance of that cell is calculated using the information in the full cell, and the ion densities that are calculated in the edge of that cell during photon transfer are just those of the center of the cell, rather than an intepolation that has on ef the endpoints at zero.

Most of the time, the treatment of partial cells does not change the predicted spectrum significantly, but this is something that is worthwhiled chacking. Users should be wary in situations where there are directions in which significant numbers of photons will pass though very few cells in the wind. This could happen for a "narrow" wind with a very small opening angle. Having a small number of cells in the wind is, of course, one should be concerned about in any event.

3.11 Examples

python is a large and complicated code with a very wide range of capabilities. Some are very non-intuitive. This section of the documentation provides worked examples on how to set up, run, and analyse the outputs of the code.

The files relating to these examples can be found in the *examples/* section of the *python* repository.

3.11.1 Reverberation Mapping

`Python` has the capability to generate transfer functions/reverberation signatures for the systems it models. These describe how a change in the ionising continuum is reprocessed into a change in line emission. These signatures can (approximately) be recovered from observation, if there's a sufficiently series of line spectra with sufficiently high time- and wavelength-resolution.

The transfer function is the term Ψ in the equation $L(v,t) = \int_0^\infty \Psi(v,\tau)C(t-\tau)d\tau$. `Python` can also generate response functions, Ψ_r, used in $\delta L(v,t) = \int_0^\infty \Psi_r(v,\tau)\delta C(t-\tau)d\tau$ (a more observationally-accessible property).

The paper discussing our implementation, and the differences between Ψ and Ψ_r are:

- Mangham 2018 [https://ui.adsabs.harvard.edu/abs/2017MNRAS.471.4788M/abstract]
- Mangham 2019 [https://ui.adsabs.harvard.edu/abs/2019MNRAS.488.2780M/abstract]

'Basic' Transfer Function

This example uses the `examples/basic/reverb.pf` file to generate the transfer function for a biconical outflowing disk wind around an AGN, driven by continuum emission of both X-rays from a spherical corona and UV emission from the hot inner regions of the accretion disk.

Building the Model

To generate the data for reverberation mapping, `Python` tracks the paths of photons as they travel throughout the wind, using them to build up a map of the response delay for each region of the wind. The settings to govern that can be found in the `### Parameters for Reverberation Modeling` section of the `.pf` file.

reverb.pf

```
### Parameters for Reverberation Modeling (if needed)
Reverb.type(none,photon,wind,matom)                wind
Reverb.disk_type(correlated,uncorrelated,ignore)  ignore
Reverb.path_bins                                   100
Reverb.visualisation(none,vtk,dump,both)           none
Reverb.filter_lines(0=off,-1=continuum,>0=count)  -1
```

There are several `Reverb.type` modes that can be used for this, but the standard one is `wind`. This handles emission in the wind by assigning photons generated there a delay compared to the central source drawn from the distribution of delays of those photons that contributed to the heating & ionisation state of that region of the wind. This distribution is discretised in `Reverb.path_bins` bins; in this example we use `100` but `1000` is more suitable for real applications.

Values that are too low will result in clear 'striping' in the transfer functions, whilst values that are too high give no real benefit and result in increased memory overhead (as each cell in the wind contains a `path_bins`-length array of doubles).

Line Filtering

An important setting is `Reverb.filter_lines`. If this is set to `0`, *every* single photon that contributes to every single spectrum (including the pseudo-photons generated after each scattering event in `extract` mode). This produces unwieldy and incredibly vast output files! The `-1` filter mode instead includes only those photons whose last interaction was a line scatter or line emission. This produces large, but less overwhelming output files.

`Reverb.filter_lines N` allows the user to include `N` different `Reverb.filter_line X` lines, where `X` is the internal `Python` data file line number for the line. This is a bit clunky and is easiest done by exploring the output files. This exploratory process is covered later on.

Ideally, this could be done in the input file by specifying the transition by species, upper and lower lines.

Running the Model

If you'd like to run `Python` directly through this notebook, you'll have to make sure whatever virtual environment you're running it in has access to the `Python` executables, and set the notebook itself to be trusted using `jupyter trust reverb.ipynb`. Then:

[ ]:
```
import subprocess
with open("reverb.txt", "w") as output_file:
    subprocess.run(['mpirun', '-np', '4', 'py', 'reverb'], stdout=output_file,␣
↪stderr=output_file)
```

Or alternatively, run it manually on the command line as:

```
mpirun -np 4 py reverb &> reverb.txt
```

This will write one `reverb.delay_dump` file per thread, and concatenate them together. If run in serial you should see the following message once it finishes:

```
cat: 'reverb_high.delay_dump[0-9]*': No such file or directory
rm: cannot remove 'reverb_high.delay_dump[0-9]*': No such file or directory
```

This can be ignored. Ideally, this would all be done directly to **HDF5** or **SQLite** or a similar file format but that's yet to be implemented.

Processing the Data

Processing the data involves using the `py4py.reverb` module provided in the `py_progs` directory. You can install this into your virtual environment using `pip`. This works by processing the very large flat text `.delay_dump` files into a **SQLite** database for further processing. This is done by:

[1]:
```
%matplotlib inline
from py4py.reverb import open_database

db = open_database('reverb')
```

```
Opening database 'reverb'...
Found existing filled photon database 'reverb'
```

You should see the output

```
Opening database 'reverb'...
No existing filled photon database, reading from file 'reverb.delay_dump':
Successfully read in ([Time]s)
```

If there are errors during reading, this will leave a malformed `reverb.db` database- to restart generation with a fixed DB file, you'll have to delete the `reverb.db` file.

Once the database has been built, you can begin producing transfer functions. Each is only meaningful for a single emission line, so you need the `python` line number. This is an internal number that depends on your choice of data files. In practical terms, the easiest way to determine it is to filter the output data file by its second (`Lambda`, wavelength in $\mathring{A}$) column to get the last (`Res.` for resonance) column, e.g. to find the C_{IV} line you would do something like:

```
awk '{if($2<1551 && $2>1549)print}' < reverb.delay_dump
```

This should give an output along the lines of the following:

```
1.9339e+15     1550.205  5.916e+35 +3.2373e+16 +2.527e+16 -2.0825e+15   0       0     2.
↪3796e+06     0     3   417
1.9343e+15     1549.889 1.9913e+35 +1.456e+16 -2.1381e+15 +1.1648e+14   0       0     7.
↪8831e+05     0     3   418
 1.934e+15     1550.089 3.7613e+35 +1.8536e+16 -2.8783e+15 +4.1224e+14   0       0     1.
↪0195e+06     0     3   418
 1.934e+15     1550.078 1.3523e+36 -2.0765e+16 -2.4398e+16 +2.5734e+15   2       1     9.
↪1914e+05     0    13   418
```

This would suggest that the doublet covers lines `417` and `418`. We can take this number and use it to generate a transfer function as

[2]:
```
from py4py.reverb import TransferFunction

c_4 = TransferFunction(db, 'C-IV', continuum=1e43+9.20e43, wave_bins=50, delay_bins=50) \
    .line(418, 1550) \
    .spectrum(1) \
    .delay_dynamic_range(3) \
    .run(limit=1000)
```

```
'C-IV' successfully run (0.0s)
```

The `TransferFunction` initialiser accepts several arguments:

- The database opened earlier, the name of the transfer function (in this case `C-IV`).

- The continuum luminosity used to generate it (central source **plus** disk continuum).
- The number of bins in time (`delay_bins`) and wavelength (`wave_bins`) for the transfer function.

The photons used to generate this transfer function are then limited by a set of functions:

- The `line()` function limits the produced transfer function to only include photons in the specific line. The wavelength of the line is also needed- ideally, it would be possible to get this straight from the `Python` atomic data files eventually.
- As the data file contains multiple spectra, we select one with `spectrum()`, providing the spectrum number (1-indexed).
- The `delay_dynamic_range()` function limits the range of delays used to calculate the delay bins. Normally, the code will generate `delay_bins` number of bins between `0` and the longest delay in the dataset. However, for models with dense regions, there can be a very long tail of photon delays, such that setting the bins to include *all* photons means condensing 90%+ of the delay distribution into a small number of bins. Instead, the `delay_dynamic_range()` function caps the delay at the $1 - 10^{argument}$th percentile, i.e. `delay_dynamic_range(3)` means the transfer function will cover 99.9% of the range of photon delays.
- The `run()` function queries the photons on disk and bins them to produce the transfer function. The `limit` parameter caps the query at `argument` photons. As a full query can run to multiple minutes, you can run quick tests using the `limit` parameter!

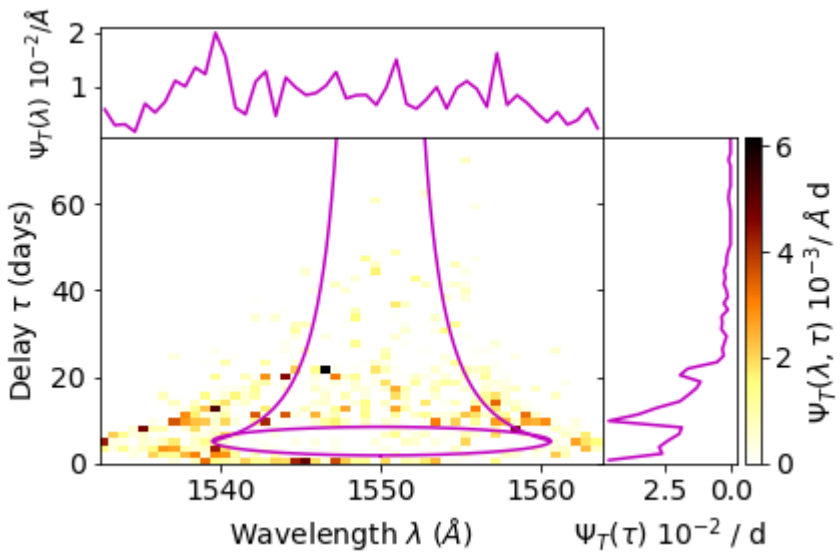
Once the data has been processed using `run()`, you can plot it.

[3]:
```
c_4.plot(
    format='png',
    keplerian={
        'angle': 40.0, 'mass': 1e7, 'radius': [1500, 3000]
    }
)

from IPython.display import Image
Image(filename='C-IV.png')
```

```
Plotting to file 'C-IV.eps'...
Total line: 4.490e-06
Successfully plotted 'C-IV.eps'(1.3s)
```

[3]:



You can overlay a plot of the expected envelope for a Keplerian rotating disk by passing a dictionary as an argument to plot:

- **angle:** The angle of observation
- **mass:** The central object mass
- **radius:** The inner and outer disk radii in terms of the central object r_g ($2 \times r_{\mathrm{schwartzchild}}$)

By default, the transfer function is output to disk (as it can take a very long time to run and there's no point risking people accidentally not saving it!). Satisfied that the transfer function has been generated successfully and looks relatively OK, we can create one with the full dataset. If we call `run()` again without a `limit` parameter, the output TF will use the same delay and wavelength bins. We'll create a new one from scratch just to be sure (as the 99.9th% of the first 1000 photons may differ a bit from the full set).

Given the previous TF appears to have a fairly long and weak outflow signature, it is reasonable to reduce the dynamic range a bit. The relatively small size of the input file as well means we should reduce the number of bins.

[4]:
```
c_4 = TransferFunction(db, 'C-IV', continuum=1e43+9.20e43, wave_bins=15, delay_bins=15) \
    .line(418, 1550) \
    .spectrum(1) \
    .delays(0, 40) \
    .wavelengths(1500, 1600) \
    .run(scaling_factor=1/5)
```

```
'C-IV' successfully run (1.8s)
```

- Instead of using the dynamic range, we can set the delay range manually. We'll select 2000 days as most of the features fall in this range.
- The `scaling_factor` parameter is used to account for the fact that the full photon DB is built up of photons from many spectral cycles. You **must** provide a scaling factor equal to 1/`Spectrum_cycles`. Ideally, this would be caught from the input `.pf` file automatically.

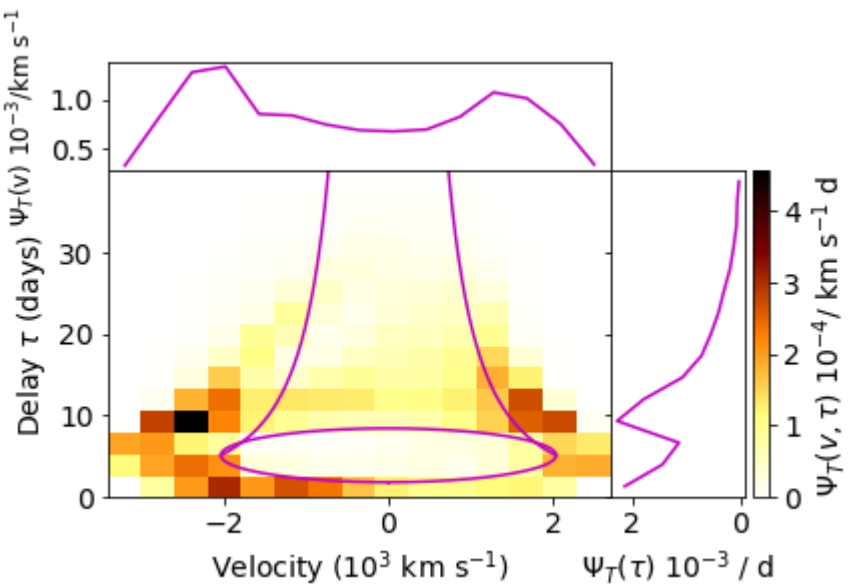
Now we can actually plot the final transfer function. For this, we will pass the `velocity=True` argument to plot to show the velocity shift on the line emission:

[5]:
```
c_4.plot(
    format='png',
    keplerian={
        'angle': 40.0, 'mass': 1e7, 'radius': [1500, 3000]
    },
    velocity=True,
)

from IPython.display import Image
Image(filename='C-IV.png')
```

```
Plotting to file 'C-IV.eps'...
Total line: 1.529e-04
Successfully plotted 'C-IV.eps'(1.0s)
```

[5]:



Intepreting the Transfer Function

The velocity-delay plane projects information on both the kinematics and geometry of the system, but in a way that is not necessarily easy to interpret. Two-dimensional transfer functions have *less* degeneracy than one-dimensional ones, but not *no* degeneracy. In addition, different behavious may appear at different timescales; in particular, in systems with both outflow and rotation will exhibit unusual behaviour around the regions where the outflow and Keplerian velocities are equivalent.

- Diagonal features, with a 'gap' in the bottom-right section (highly red-shifted at short delays), a.k.a. blue-leads-red, are associated with outflows.
  - Any highly red-shifted material has to have been reprocessed in the outflows heading away from the observer; thus, the travel time for this is longer.
  - These typically also have a gap in the top-left section of the transfer function (highly blue-shifted at short delays). In classic reverberation models this is a crisp, well-defined line. Taking multiple-scattering into account, photons from any position in the wind can be reprocessed again in the outflow region; so this region bleeds upwards.
  - **However:** Be aware that these signatures might be convolved with rotation (see below).
- Oval features are associated with Keplerian rotation.
  - 'Squashed' ovals, with high red- & blue-shift at short delays, are typically disk or wind inner edges of reprocessing material in Keplerian rotation. The closer material is to the central object, the faster it rotates. Again, classic reverberation models have very crisp edges to these signatures, but multiple-scattering will smear them out in our code.
  - 'Tall' ovals, with low red- & blue-shift at delays ranging from short to long, are typically disk or wind outer edges. As before, these signatures are less crisp than classical ones due to multiple-scattering. Emission typically falls off in the middle of the oval, and drops dramatically beyond it.
    - If there is no distinct edge, just a smooth falloff, this is likely due to the reprocessing region extending off beyond the region at which the falloff due to distance from the central source reduces the response to zero.

  - **However:** Be aware that these signatures might be convolved with outflow (see below).
- 'Tilted tall' ovals are associated with Keplerian rotation convolved with outflow.
  - These arise due to the interaction between outflow and Keplerian rotation. Those regions that are furthest from the observer (i.e. at longest delays) are the most red-shifted, and those that are at short delays are the most blue-shifted.
  - See Mangham 2018 for more details, as this is quite hard to explain without a good image.
  - Crucially, over short distances these can *appear* to display blue-leads-red signatures at low resolution.

The transfer function above shows clear signs of keplerian rotation with an oval inner-edge signature. However, there is no distinct longer-delay outer edge signature, suggesting that emission is not from a well-defined disk with a hard edge but that it tails out slowly. There is also a small blue-leads-red signature, more visible if the transfer function is plotted with a logarithmic response, with a slight 'diagonal' signature visible. This suggests that the kinematics of the emission region are largely rotationally-dominated with a hint of outflow. Notably, this does not imply that the entire system **itself** is rotationally dominated or in outflow, simply that the kinematics of the material that is emitting in $C_{\rm IV}$ are so.

If our initial run didn't provide enough detail, we can create a new file with more spectral cycles, and limit it to output only the photons we want to keep the file size down, for example as below to keep only the $C_{\rm IV}$ photons from the upper line of the doublet:

reverb_extended.py

```
### Parameters associated with photon number, cycles,ionization and radiative transfer␣
↪options
[...]
Spectrum_cycles                             50
[...]

### Parameters for Reverberation Modeling (if needed)
[...]
Reverb.filter_lines                         1
Reverb.filter_line                          418
```

Responsivity-Weighted Transfer Function

The basic transfer function generated is more properly the **'emissivity-weighted'** transfer function; it assumes that there are no changes in ionisation state in the wind, and that a 10% increase in central source luminosity results in a 10% increase in reprocessed luminosity in the wind. Given observations of the breathing BLR in AGN or of periods of anticorrelation between continuum and H luminosity in the AGN NGC5548, this is unlikely to be true!

Python can produce **responsivity-weighted** transfer functions, A.K.A 'response functions'. It does this by performing two separate runs of models that bracket the main model in luminosity, and approximates the response function at the central luminosity by using the gradient between the two transfer functions. We produce two copies of the main model, each with all sources of luminosity adjusted up and down by the same value. Typically we choose 10%, assuming that the responsivity is constant across this region. As the regions of peak response typically move through the wind with changing luminosity (e.g. ionisation fronts being pushed back when continuum luminosity increases), selecting too large a bracket can result in issues with the emitting wind regions failing to overlap.

Building the Models

So we create two new copies of the file `reverb.pf`, `reverb_low.pf` and `reverb_high.pf`.

The original lines in reverb.pf are:

reverb.pf

```
### Parameters for the Disk (if there is one)
[...]
Disk.mdot(msol/yr)                          0.02
[...]

### Parameters for Boundary Layer or the compact object in an X-ray Binary or AGN
[...]
Central_object.luminosity(ergs/s)           1e43
[...]

### Parameters for Reverberation Modeling (if needed)
[...]
Reverb.filter_lines(0=off,-1=continuum,>0=count)   -1
```

In our low and high versions of the file (in the same directory), they are:

reverb_low.pf

```
### Parameters for the Disk (if there is one)
[...]
Disk.mdot(msol/yr)                          0.018
[...]

### Parameters for Boundary Layer or the compact object in an X-ray Binary or AGN
[...]
Central_object.luminosity(ergs/s)           0.9e43
[...]

### Parameters for Reverberation Modeling (if needed)
[...]
Reverb.filter_lines                         1
Reverb.filter_line                          418
```

reverb_high.pf

```
### Parameters for the Disk (if there is one)
[...]
Disk.mdot(msol/yr)                          0.022
[...]

### Parameters for Boundary Layer or the compact object in an X-ray Binary or AGN
[...]
Central_object.luminosity(ergs/s)           1.1e43
[...]

### Parameters for Reverberation Modeling (if needed)
[...]
Reverb.filter_lines                         1
Reverb.filter_line                          418
```

NOTE: You may have to change the `Reverb.filter_line` entry if the line number you found earlier is different. Line numbers *depend on the data files*, so any update to the data files may change the line numbers.

These two runs are used to calculate the response function. The final response function will *only* use the output of these two, and not use any of the photons used to generate the transfer function, so you may want to increase the number of photons run. However, for diagnostic purposes it's useful to generate a response functions from each pair (`-10%` $\rightarrow$ `+10%`, `-10%` $\rightarrow$ `0%`, `0%` $\rightarrow$ `10%`); any substantial differences between them will highlight that you're at a point of inflection in the ionisation profile.

Given we know which emission line number we're looking for now, we can set the filter to exclude all other photons from being stored using `Reverb.filter_lines`; this will dramatically reduce the file-size.

Running the Models

Now, we run these two models as well. They'll similarly output one `reverb_[low/high].delay_dump[thread]` file per thread, which will be concatenated together at the end of the run to produce a single `reverb_[low/high].delay_dump` file.

[ ]:
```
import os
os.system("mpirun -np 8 py reverb_low.pf > reverb_low.txt")
os.system("mpirun -np 8 py reverb_high.pf > reverb_high.txt")
```

You may want to change the number of processors used, depending on your system.

Processing the Data

Now, we need to go back to the `py4py.reverb` package. Each one of the new runs needs to be processed into a **sqlite** database, and we can then produce a response function from them. We can base the analysis on the previously-made `TransferFunction` as

[7]:
```
from py4py.reverb import TransferFunction

db_low = open_database('reverb_low')
c_4_low = TransferFunction(db_low, 'C-IV_high', template=c_4, continuum=0.9e43+8.29e43) \
    .delay_dynamic_range(2) \
```

(continues on next page)

(continued from previous page)

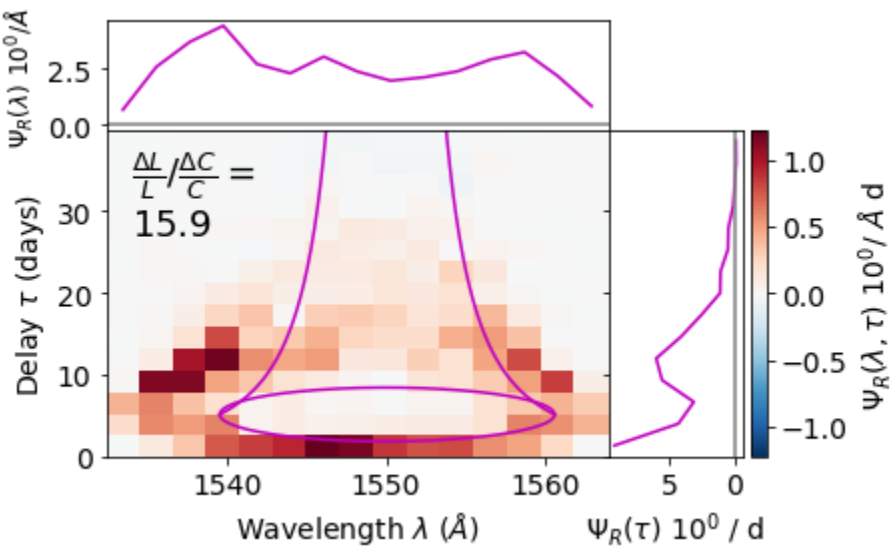
```
    .run(scaling_factor=1/5)

db_high = open_database('reverb_high')
c_4_high = TransferFunction(db_high, 'C-IV_high', template=c_4, continuum=1.1e43+1.
↪01e44) \
    .delay_dynamic_range(2) \
    .run(scaling_factor=1/5)

c_4.response_map_by_tf(c_4_low, c_4_high).plot(
    name='resp', format='png', response_map=True,
    keplerian={
        'angle': 40.0, 'mass': 1e7, 'radius': [1500, 3000]
    }
)
Image(filename='C-IV_resp.png')
```

```
Opening database 'reverb_low'...
Found existing filled photon database 'reverb_low'
Templating 'C-IV_high' off of 'C-IV'...
'C-IV_high' successfully run (1.2s)
Opening database 'reverb_high'...
Found existing filled photon database 'reverb_high'
Templating 'C-IV_high' off of 'C-IV'...
'C-IV_high' successfully run (1.0s)
Plotting to file 'C-IV_resp.eps'...
Total response: 2.435e-03
Successfully plotted 'C-IV_resp.eps'(0.9s)
```

[7]:



- The `template=` argument is required to produce a response function. A templated `TransferFunction` will share the same wavelength and delay bins, along with the same line and spectrum.
- The argument `template_different_line=True` can be passed when declaring a new `TransferFunction`. This can be used to ensure consistency between e.g. C_{IV} and $H\alpha$ transfer/response function plots. These will share the same *velocity* bins, delay bins and spectrum.
- Once the two `TransferFunctions` for the different continuum values have been generated, we subtract the two

and divide by ΔC. This assumes that the difference in line response is linear across ΔC!

- Notably, whilst the response function data is stored on the template `TransferFunction` object, none of that object's transfer function data is used to calculate it.

Interpreting the Response Function

Critically, unlike transfer functions, response functions can have both positive and negative regions. `py4py` by default uses **red** to indicate regions of positive response and **blue** to indicate regions of negative response.

- Naively, we would expect the response function to have the same distribution as the transfer function; if plotted with `rescaled=True` (such that the maximum value in any given bin is 1) the two should be identical. If the response profile of the wind differs at all across the interval ΔC, then the response function should be different.
- As the luminosity of the central source increases, this can push ionisation fronts back into the wind. This can result in a **reduction** in response at low delays and high Doppler shifts.
- It can also result in over-ionization of low-density extended wind regions. This shows most clearly as a **reduction** in the response at long delays.
- In some models, there can in fact be a net negative response. It can be difficult to gauge this from the response function, but if this is the case it should be shown on the relative line increase against the global responsivity $\frac{\Delta L}{L} / \frac{\Delta C}{C}$. Naively, a 10% increase in continuum should result in a 10% increase in line luminosity giving a value of 1, a negative value indicates a reduction in line luminosity with increasing luminosity.

For a response function to work, both runs used to produce it need to fully sample the whole velocity-delay space, otherwise spurious positive and negative responses are visible based solely on the presence or absence of photons in any given bin.

This response function is relatively well-behaved. It closely tracks the emissivity map, indicating that (for this system) the traditional assumptions of reverberation mapping hold. The response is strong at low delays and around the line of Keplerian rotation, and line emissivity increases at a substantially higher rate than continuum emissivity. This suggests that the inner wind is in a relatively low ionisation state and that increasing luminosity affects it significantly, substantially increasing the proportion of C_{IV}.

3.11.2 Demo: Quasar, M20

The collaboration has published a series of papers using parameterised, biconical disc wind models. The initial model focus mostly on broad absorption quasars (Higginbottom et al 2013), since the emission line were too weak in that case too match observed BLR properties. In Matthews et al 2016, we included a treatment of clumping and found some. Finally, in Matthews et al 2020 (hereafter M20) we used a similar model to the previous clumpy wind model, but explored some of the behaviour in the ionizing flux density plane, and also used an isotropic illuminating SED.

This particular document focuses on Model A from M20. As with most of the demo models discussed here, the model makes use of the *Shlosman & Vitello (1993) wind prescription*.

The wind is equatorial, and illuminated by an isotropic SED.

Todo: more description needed

Important Parameters

Central Source Parameters:

$$M_{\rm BH} = 10^9 M_\odot$$
$$\dot{M}_{\rm acc} = 5 M_\odot \ {\rm yr}^{-1}$$
$$L_{2-10 \ {\rm keV}} = 10^{43} \ {\rm erg \ s}^{-1}$$

Wind parameters:

$$\dot{M}_{\rm wind} = 5 M_\odot \ {\rm yr}^{-1}$$
$$\theta_{\rm min} = 70^\circ$$
$$\theta_{\rm max} = 85^\circ$$
$$r_{\rm min} = 300 r_g$$
$$r_{\rm max} = 600 r_g$$
$$R_v = 10^{19} \ {\rm cm}$$
$$f_V = 0.01$$

Illuminating SED

Runtime

3h25min on 64 cores (218 core hours).

Outputs

References

3.11.3 Demo: Tidal Disruption Event

One of the recent applications of Python is modelling outflows in Tidal Disruption Events (TDEs). We have explored how line formation in an accretion disc wind could explain the BAL vs. BEL dichotomy observed in the UV spectra of TDEs. We have also explored how reprocessing in an accretion disc wring could give rise to the, at one point, unexpected optically bright TDEs.

We now describe a model used to simulate both the UV and optical features of TDEs.

Model Setup

Key Model Parameters

We model a disc wind outflow using the kinematic Shlosman & Vitello (1993) (SV93) biconical disc wind model. This model has seen extensive use throughout the history of Python to model across all length scales of accretion, from CVs to QSO winds. Further information about the SV93 model can be found in the documentation here.

The key parameters controlling the geometry and central object in this model are as follows.

Schwarzschild black hole parameters:

$$\begin{aligned} \mathrm{R_{disc,\,in}} &= 10^{13}\ \mathrm{cm} \\ &= 22.8\ \mathrm{R_g} \\ \mathrm{R_{disc,\,out}} &= 10^{15}\ \mathrm{cm} \\ &= 2258\ \mathrm{R_g} \\ \mathrm{M_{BH}} &= 3 \times 10 \mathrm{M_\odot} \\ \dot{\mathrm{M}}_{\mathrm{disc}} &= 9.99 \times 10^{-3}\ \mathrm{M_\odot\ yr^{-1}} \\ &= 0.15\ \dot{\mathrm{M}}_{\mathrm{Edd}} \end{aligned}$$

Wind geometry parameters:

$$\begin{aligned} r_{\mathrm{min}} &= 10^{13}\ \mathrm{cm} \\ &= 22.8\ \mathrm{R_g} \\ r_{\mathrm{max}} &= 10^{15}\ \mathrm{cm} \\ &= 2258\ \mathrm{R_g} \\ \alpha &= 1.5 \\ R_v &= 5 \times 10^{16} \\ &= 1.13 \times 10^5\ \mathrm{R_g} \\ \theta_{\mathrm{min}} &= 20^\circ \\ \theta_{\mathrm{max}} &= 65^\circ \\ \mathrm{R_{max}} &= 5 \times 10^{17}\ \mathrm{cm} \\ &= 1.13 \times 10^6\ \mathrm{R_g} \\ \gamma &= 1 \\ \lambda &= 0 \\ f_v &= 0.1 \end{aligned}$$

For parameters controlling the radiative transfer and flow of Python, the parameter file for this model can be found `here`.

Radiation Sources

There are two radiation sources in this model; the accretion disc and the wind itself. Although, the wind does not act as a *net* source of photons, but rather as a reprocessing medium. We assume that the wind is in radiative equilibrium meaning any energy absorbed is reprocessed and re-radiated, i.e. via radiative recombination. We treat the accretion disc as an ensemble of black bodies, using a standard α-disc effective temperature profile (Shakura & Sunyaev, 1973). The emergent SED is hence specified entirely by the mass accretion rate of the accretion disc and the mass of the black hole.

The figure below shows the angle integrated SED for this model.

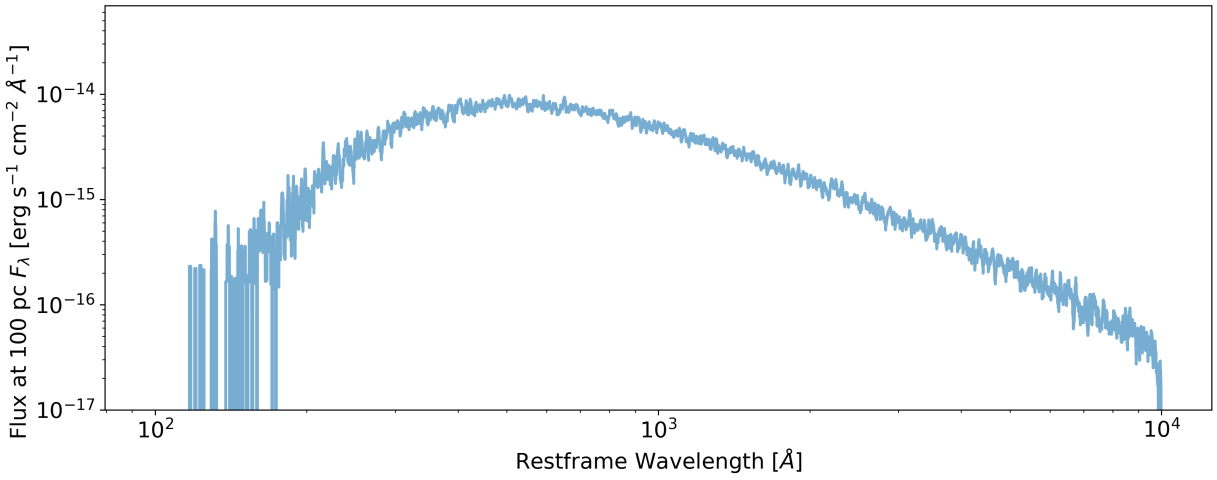


Fig. 6: The angle integrated accretion disc SED for the TDE model.

Runtime

As the TDE outflow is optically thick, the model requires a fair amount of computing power to be completed within a reasonable time frame. We ran this model using two Intel Xeon Platinum 8160 processors with 24 processor cores each for a total of 48 cores. Each processor core runs at a clock frequency of 2.1 GHz, with a maximum boost clock of 3.7 GHz. The model uses roughly 70 GB of the available DDR4 2666 MHz memory available in this system.

With this configuration using 10^8 photons and Python's "-p 2" option for logarithmic photon number stepping, the model takes roughly 10 ionization cycles to converge in roughly 7.5 hours, or 360 total CPU hours. The spectral cycles take a significantly longer time to complete. For six inclination angles and 10^8 photons, a single spectral cycle takes in excess of three hours. However, with 10^6 photons a spectral cycles takes roughly 100 seconds. We find that 5 - 10 spectral cycles with 10^6 photons result in reasonable sacrifice between noise in the final spectrum and the run time of the spectral cycles.

Outputs

Synthetic Spectra

Below is a figure of three inclination angles of the emitted spectrum for this model.

The model produces the strong resonance lines of N V, Si IV and C IV often seen in UV spectra of TDEs and other objects with mildly ionized winds. We also reproduce the BAL vs. BEL behaviour seen, as described in, i.e. Parkinson et al. (2020),. For inclinations which look into the wind, BALs are preferentially produced and for inclinations looking above or below the wind, BELs are instead seen.

In the optical portion of the spectrum, the model produces broad recombination emission features for the Balmer series of lines as well as for He II. These features have extended red wings, clearest at low inclination angles. At intermediate and high inclinations, the emission features are double peaked due to the high rotational velocity of the wind near the base of the wind, where these features are forming.

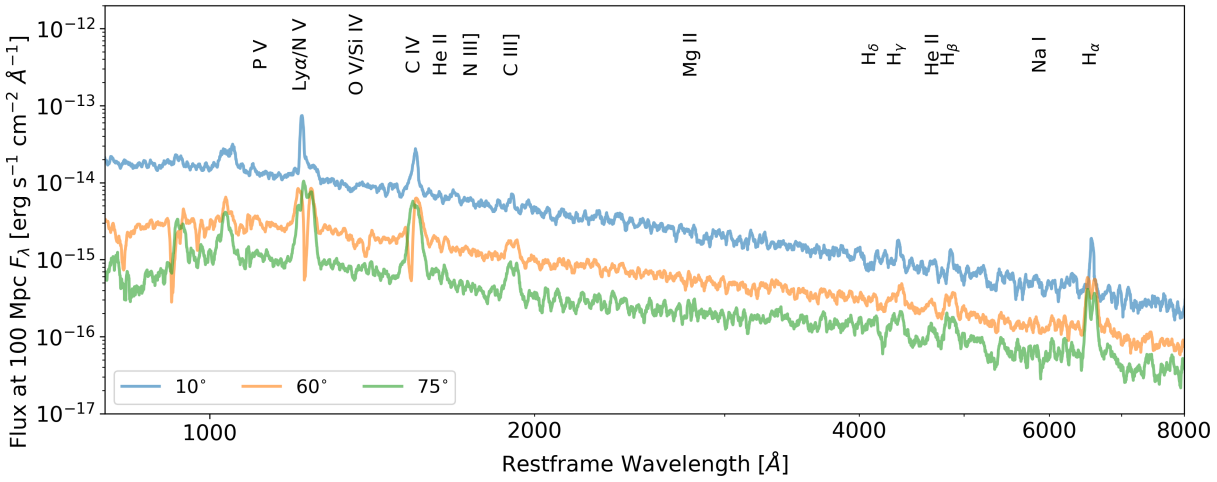


Fig. 7: Synthetic spectra of the TDE model for three inclination angles, as labelled in the lower left. The 60 ° sight line is looking down, into, the wind, whereas both noth the 10 ° and 75 ° sight lines are not looking above and below the wind respectively. Important line transitions have been labelled at the top of the plot.

Physical Properties

In the figure below, the physical properties of the outflow are shown.

At the base of the wind, the velocity is dominated by rotation. The rotational velocity decreases with radius, due to conserving angular momentum. Far out in the wind, the velocity is dominated by the polodial velocity, as set by the velocity law in the model. The electron temperature and density are both greatest at the base of the wind. The density decreases with radius, resulting in line formation processes which scale with electron density, such as collisional excitation, decreasing with radius also.

The outer top edge of the wind is cool, reaching temperature as low as $T_e \sim 10^3$ K. Python does not implement any dust or molecular physics, hence the treatment of this region of the wind is highly approximate. However, since the line formation we are interested in does not occur in this region, our neglect of this physics should not effect the emergency spectrum.

To measure the ionization state of the wind, we define the ionization parameter $U_{\rm H}$,

$$U_{\rm H} = \frac{4\pi}{n_{\rm H} c} \int_{13.6\frac{\rm eV}{h}}^{\infty} \frac{J_\nu}{h\nu}\, d\nu,$$

where ν denotes frequency, $n_{\rm H}$ is the number density of Hydrogen, h is Planck's constant and J_ν is the monochromatic mean intensity. The ionization parameter measures the ratio of the number density of Hydrogen ionizing photons to the local matter density. For values of $U_{\rm H} > 1$, Hydrogen is ionized making it a useful predictor of the global ionization state. The ionization parameter is fairly constant throughout the wind with $U_{\rm H} \sim 10^4$, indicating that the Hydrogen is ionized in much of the wind. At the very top of the wind, the wind is highly ionized with $U_{\rm H} \sim 10^8$. There is, however, a portion of the wind where $U_{\rm H} < 1$. This part of the wind is at the base of the wind and large disc radii, $\rho \sim 10^{15}$ cm, where Hydrogen is neutral. The density of neutral Hydrogen is, naturally, greatest here with $n_{\rm H\,I} \sim 10^7$ cm^{-3} and is where the majority of H α photons are emitted.

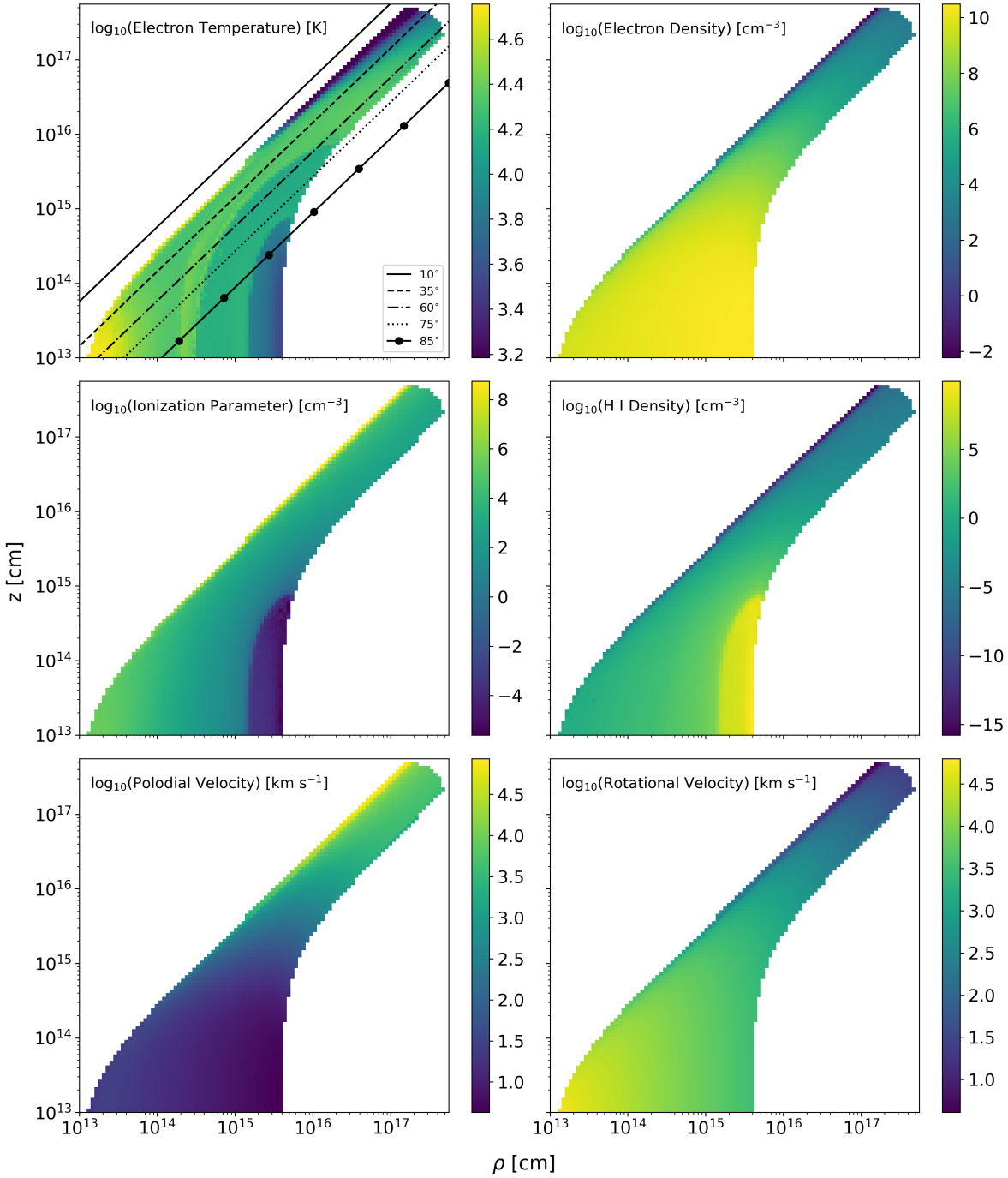


Fig. 8: Contour plots of various physical parameters for the wind model, plotted on a log-log spatial scale. The top left panel shows which parts of the wind four inclination inclinations intersect.

Files

Attached below is the parameter file for the model and three spectrum files.

- `tde_fiducial.pf`
- `tde_fiducial.spec`
- `tde_fiducial.log_spec`
- `tde_fiducial.spec_tot`

3.11.4 Benchmark: 1D Stellar Wind, CMFGEN

3.11.5 Benchmark: 1D Homologous SN, Tardis

3.12 Physics & Radiative Transfer

Various physical concepts are incorporated into Python. Some of these are descibed below:

3.12.1 Radiative Transfer Modes

Python has a number of different radiative transfer modes, which affect the treatment of lines and scattering, and also whether we use indivisible packet constraints or allow photon weights to be attenuated by continuum absorption. These modes are selected with the parameter *Line_transfer*. The different modes are briefly described on that parameter page. This page is designed to give an overview of the assumptions and concepts behind, as well as the basic operation of, the different techniques. The aim is that, in partnership with the parameter page and the atomic data documentation, all the information regarding the different radiative transfer modes should be present.

For introductions and references regarding Monte Carlo radiative transfer techniques generally, we recommend reading Noebauer & Sim 2019. For specifics regarding Python, we recommend reading Long & Knigge 2002 as well as PhD theses by Higginbottom and Matthews.

Sobolev Approximation

Python always uses the Sobolev approximation to treat line transfer. In this approximation, it is assumed that the thermal line width is small compared to the velocity gradient. The Sobolev approximation is described extensively in astrophysics literature, and so we do not describe it in detail here. We refer the users to section 8.2 of Noebauer & Sim 2019 and references there in for a discussion of the Sobolev escape probabilities approach.

Weight Reduction v Indivisible Packets

Python was originally written in such a way that photon packet weights were not indivisible and allowed to be attenuated. This is the way the code is described in the original Long & Knigge 2002 paper. In the standard, weight reduction mode, photon weights are attenuated by continuum opacities (free-free, bound-free). Conservation of energy is then hopefully achieved by calculating the emission from the wind .

In indivisible packet mode, there is a fundamental shift in philosophy. All energy packets are strictly indivisible and conserve energy whenever they undergo radiative processes (the only exception is adiabatic cooling). Thus, even bound-free absorption is dealt with at a single interaction point.

Indivisible packet mode is activated by setting the *Line_transfer* parameter to either `macro_atoms` or `macro_atoms_thermal_trapping`. The terminology adopted here is slightly confusing, since the line transfer mode does not explicitly include a macro-atom treatment of atomic species (see next subsection).

Developer Note

The radiative transfer mode is stored using the code variable geo.rt_mode.

Macro-atoms and 2-level-atoms

The macro-atom scheme was devised by Leon Lucy in the early 2000s (Lucy 2002, Lucy 2003). It involves the re-formulation of the process of radiation transport through a plasma in radiative equilibrium into a traffic-flow problem. Lucy showed that, when in radiative equilibrium, the energy flows through a system depend only on the transition probabilities and atomic physics associated with the levels the energy flow interacts with. By quantising this energy flow into radiant (r-) and kinetic (k-) packets, we can simulate the energy transport through a plasma discretised into volume elements (*macro-atoms*), whose associated transition probabilities govern the interaction of radiant and kinetic energy with the ionization and excitation energy associated with the ions of the plasma.

Todo: add refs, describe properly.

Developer Note

Macro-atoms are identified using their atomic data, in particular by providing data with identifiers LevMacro, LinMacro, PhotMacro.

Simple-atoms still interact with r- and k-packets, but do not possess internal transition probabilities. As a result, they are analogous to the two-level atom treatment, as any excitation is immediately followed by a deactivation into an r- or k-packet. The choice of radiative or kinetic deactivation is made according to the relative rates in the two-level atom formalism.

Isotropic v Anisotropic Line Scattering

Python always treats electron scattering as an isotropic process, and continuum emission processes are also treated as isotropic, except for Compton scattering. For Compton scattering, the direction and energy change is calculated self-consistently according to the energy change formula $E/E' = 1 + (h\nu/mc^2)(1 + \cos\theta)$. We first draw a random cross section that our photon packet will see. This cross section represents an energy change and hence a direction. The distribution of angles is taken care of by using a differential cross section vs energy change function.

Caution

Compton scattering is currently not accounted for when using indivisible packet mode.

Line emission and scattering is isotropic unless one of the `thermal_trapping` line transfer modes is selected. In the thermal trapping mode, any line interaction or emission results in an anisotropic direction being generated. This direction is generated by a rejection method which samples the Sobolev escape probability in each direction from the line interaction region. Unless you specifically want to consider isotropic line emission, we recommend always using the anisotropic thermal trapping mode.

Todo: move the below to where we describe photon sources and generation?

In the case of isotropic emission, the direction of a photon packet is chosen so that the probability of emission in each bin of solid angle is the same. It follows that

$$p(\Omega)d\Omega \propto \cos\theta \sin\theta d\theta d\phi$$

where the angles are in polar coordinates and relative to the local outward normal. For a spherical emitting source, such as a star, one must first generate a location on the star's surface and then calculate the photon direction relative to the normal at the point. For emission from optically thick surfaces the above equation can be modified to include linear limb darkening, $\eta(\theta)$, such that

$$p(\theta, \phi)d\theta d\phi = \eta(\theta) \cos\theta \sin\theta d\theta d\phi.$$

The Eddington approximation is usually adopted in the code, so that $eta(theta)$ is given by

$$\eta(\theta) = a(1 - \frac{3}{2}\cos\theta).$$

The constant a is normalised such that the total probability sums to 1. Whenever a radiation packet undergoes an electron scatter, the new direction is chosen to be isotropic. However, when the photon is a line photon, the new direction is chosen according to a line trapping model, which samples a probability distribution according to the Sobolev escape probability in different directions.

Doppler Shifts and The Comoving Frame

When calculating opacities, the photon frequency must be shifted from the rest frame of the photon into the rest frame of the plasma. This shift depends on the before and after directions of the photon. Let us denote these two directions with unit vectors $\vec{n}_i$ and $\vec{n}_f$, respectively, and consider a situation when a photon scatters off an electron in a region of the wind moving at velocity $\vec{v}$. The final frequency of the photon with initial frequency is

$$\nu_f = \nu_i \frac{1 - (\vec{v} \cdot \vec{n}_i)/c}{1 - (\vec{v} \cdot \vec{n}_f)/c}.$$

In the case of a resonance scatter with line transition u to j, the new frequency is

$$\nu_f = \frac{\nu_{uj}}{1 - (\vec{v} \cdot \vec{n}_f)/c}.$$

The above formulae are the non-relativistic case, which is currently used in the code. However, this should in general be improved to use the special relativistic formula. This would produce more accurate Doppler shifts for the fastest regions of an outflow, as the current treatment introduces errors of order 5 Angstroms at the blue edges of the highest velocity absorption lines in quasar and CV wind models.

When real photons resonantly (or electron) scatter off real plasma in a flow, they conserve energy and frequency in the co-moving frame of the plasma. In the case of an outflow, doing the frame transformation from system->flow->system over the course of an interaction results in a redshifting of a photon, and as a result an energy loss - in other words, the photon does work on the flow even though energy is conserved in the co-moving frame. Indivisible packet schemes (such as macro-atoms) often enforce strict energy conservation in the frame of a given cell (physically, but see also Lucy 2002). This means that, when keeping track of packets in the observer frame, one needs to correct the energies (not just the frequencies) using a Doppler shift. Python does **not** currently conserve energy in the co-moving frame.

Todo: test whether this is an issue.

3.12.2 Macro Atoms

Todo: This page shoud contain a description of how macro atoms work. The below is copied from JM's thesis.

Todo: Add description of accelerated macro-atom scheme

The macro-atom scheme was created by Leon Lucy and is outlined in his 2002/03 papers. It was implemented in Python by Stuart Sim, initially for the study of recombination lines in YSOs (Sim et al. 2005).

Lucy (2002,2004) hereafter L02, L03) has shown that it is possible to calculate the emissivity of a gas in statistical equilibrium without approximation for problems with large departures from LTE. His *macro-atom* scheme allows for all possible transition paths from a given level, dispensing with the two-level approximation, and provides a full non-LTE solution for the level populations based on Monte Carlo estimators. The macro-atom technique has already been used to model Wolf-Rayet star winds (Sim 2004), AGN disc winds (Sim et al. 2008), supernovae (Kromer and Sim 2009, Kerzendorf and Sim 2014) and YSOs (Sim et al. 2005). A full description of the approach can be found in L02 and L03.

Following L02, let us consider an atomic species interacting with a radiation field. If the quantity ϵ_j represents the ionization plus excitation energy of a level j then the rates at which the level absorbs and emits radiant energy are given by

$$\dot{A}_j^R = R_{lj}\epsilon_{jl}$$

and

$$\dot{E}_j^R = R_{jl}\epsilon_{jl}$$

where $\epsilon_{jl} = \epsilon_j - \epsilon_l$. Here, we have adopted Lucy's convention, in which the subscript l denotes a summation over all lower states ($< j$), and (u) a summation over all upper states ($> j$). Similarly, the rates corresponding to _kinetic_ (collisional) energy transport can then be written as

$$\dot{A}_j^C = C_{lj}\epsilon_{jl}$$

and

$$\dot{E}_j^C = C_{jl}\epsilon_j,$$

Let us define $\mathcal{R}$ as a total rate, such that $\mathcal{R}_{lj} = R_{lj} + C_{lj}$. If we now impose statistical equilibrium

$$(\mathcal{R}_{lj} - \mathcal{R}_{jl}) + (\mathcal{R}_{uj} - \mathcal{R}_{ju}) = 0 \quad ,$$

we obtain

$$\begin{aligned} &\dot{E}_j^R + \dot{E}_j^C + \mathcal{R}_{ju}\epsilon_j + \mathcal{R}_{j\ell}\epsilon_l \\ &= \dot{A}_j^R + \dot{A}_j^C + \mathcal{R}_{uj}\epsilon_j + \mathcal{R}_{lj}\epsilon_l. \end{aligned}$$

This equation is the starting point for the macro-atom scheme. It shows that, when assuming radiative equilibrium, the energy flows through a system depend only on the transition probabilities and atomic physics associated with the levels the energy flow interacts with. By quantising this energy flow into radiant ($r-$) and kinetic ($k-$) packets, we can simulate the energy transport through a plasma discretised into volume elements (macro-atoms),whose associated transition probabilities govern the interaction of radiant and kinetic energy with the ionization and excitation energy associated with the ions of the plasma.

Although the equation above assumes strict radiative equilbrium,it is trivial to adjust it to include non-radiative source and sink terms. For example, in an expanding parcel of plasma, adiabatic cooling may be included with a simple modification to the RHS. Currently, we include adiabatic cooling by destroying packets with a probability $p_{i,\mathrm{destruct}} = \mathcal{C}_{\mathrm{adiabatic}}/\mathcal{C}_{\mathrm{tot}}$.

A Hybrid Scheme

A pure macro-atom approach with only H and He can be easily used for some situations – for example, in the YSO application described by, which uses a H-only model. However, in accretion disc winds, the densities can be very high, and higher Z elements must be included. Including all these elements as macro-atoms is not currently computationally feasible in the code for anything but the simplest models. We thus often use a *hybrid scheme*, which treats H and He with the macro-atom approach, but models all other atoms as *simple-atoms*.

Simple-atoms still interact with r- and k-packets but do not possess internal transition probabilities. As a result, they are analogous to the two-level atom treatment, as any excitation is immediately followed by a deactivation into an r- or k-packet. The choice of radiative or kinetic deactivation is made according to the relative rates in the two-level atom formalism. For a bound-bound transition $u \rightarrow j$, these two probabilities

are then

$$p_{uj}^{S,R} = \frac{A_{uj}\beta_{uj}}{A_{uj}\beta_{uj} + C_{uj}\exp(-h\nu_{uj}/kT_e)} = 1 - q$$

and

$$p_{uj}^{S,C} = \frac{C_{uj}\exp(-h\nu_{uj}/kT_e)}{A_{uj}\beta_{uj} + C_{uj}\exp(-h\nu_{uj}/kT_e)} = q.$$

For a bound-free transition, the code assumes radiative recombination, and thus any bound-free simple-atom activation is immediately followed by the creation of an r-packet. This approximates the bound-free continuunm, even when compared to other two-level atom radiative transfer schemes. This is discussed further and tested in section~ref{sec:line_test}.

This hybrid approach preserves the fast treatment of, for example, UV resonance lines, while accurately modelling the recombination cascades that populate the levels responsible for, e.g., H and He line emission. As a result of this hybrid scheme, a separate set of estimators must be recorded for simple-atoms, and the ionization and excitation of these elements is calculated with a different, approximate approach. In order to include simple-atoms, we must add in a few extra pathways, so that energy packets can also activate simple-atoms, through either bound-free or bound-bound processes. The relative probabilities of these channels are set in proportion with the simple-atom opacities.

Macro-atom Emissivity Calculation

In order to preserve the philosophy that a detailed spectrum is calculated in a limited wavelength regime, Python carries out a macro-atom emissivity calculation before the spectral cycles. The aim of this step is to calculate the luminosity contributed by macro-atoms – equivalent to the total amount of reprocessed emission – in the wavelength range being considered.

This process can be very computationally intensive, especially if the wavelength regime being simulated has very little emission from bound-free and line processes in the wind, but the overall broad-band emissivity is high. During the ionization cycles, the amount of energy absorbed into k-packets and every macro-atom level is recorded using MC estimators. Once the ionization cycles are finished, and the model has converged, these absorption energies are split into a certain number of packets and tracked through the macro-atom machinery until a deactivation occurs. When this happens, the emissivity of the level the macro-atom de-activated from is incremented if the packet lies in the requested wavelength range. If it does not, then the packet is thrown away. It is easy to see how what is essentially a MC rejection method can be an inefficient way of sampling this parameter space. Fortunately, this problem is parallelised in the code.

Once the emissivities have been calculated, the spectral synthesis can proceed. This is done in a different way to the ionization cycles. Photons are generated from the specified photon sources over the required wavelength range, but are now also generated according to the calculated macro-atom and k-packet emissivities in each cell. These photons are "extracted" as with normal photon packets. In order to ensure that radiative equilibrium still holds, any photon that interacts with a macro-atom or k-packet is immediately destroyed. The photons are tracked and extracted as normal until they escape the simulation; resonant scatters are dealt with by a combination of macro-atom photon production and destruction.

Developer note: Emissivities

We are a little lax in terms of what we actually call an emissivity in the code. The quantities stored in variables like `kpkt_emiss` and `matom_emiss` in the plasma and macro-atom structures are actually *comoving-frame energies* in erg, which are sampled when generating r-packets in each cell. Roughly speaking, these are luminosities given that the code assumes a time unit of 1s. Similarly, when the code prints out level *emissivities* to screen and to the diag file, these are really a sum over all these quantities (and can approximately be thought of as level *luminosities*).

Bound-free Continua of Simple Atoms

Todo: this section is not yet completely accurate.

Historically, when using the indivisible packet form of radiative transfer (*macro_atoms_thermal_trapping*, for example), the bound-free continua of simple atoms were treated in a simplified two-level framework. In this case, simple atoms are those *without* a full macro-atom model atom, usually the metals. In this two-level scheme, whenever a simple atom undergoes a bound-free interaction, it is excited into the continuum state, and this is immediately followed by recombination, and an r-packet or k-packet is created immediately. As a result, the scheme does not capture the physical situation whereby a recombination cascade can occur from an initial recombination to excited levels, leading to a gradual reddening of the photon if there are many interactions. This situation **is** modelled well by a full macro-atom treatment.

To try and slightly improve this scheme, we implemented a "total emissivity" upweighting scheme around 2018. The basic idea is that we pay attention to only the heating and cooling. In particular, the rates of all simple atom bound-free emission are governed by the *emissivity* of the bound-free process.

This result in two changes to the code for ionization cycles:

- whenever a k-packet is eliminated via a bound-free channel of a simple macro atom (simulating energy flow from the k-packet pool to the radiation pool, $k \rightarrow r$), we have that packet carry additional energy corresponding to the required ionization energy for that particular bf process. This means we upweight the energy of the packet by a factor $f_{\rm up} = \nu/(\nu - \nu_0)$, where ν is the frequency of the new bound-free photon and ν_0 is the threshold frequency. This quantity is the ratio of the total energy carried by photons in the packet to the energy supplied to photons in the packet from the thermal pool.
- whenever an r-packet is "absorbed" by a simple macro atom bound-free process we track explicitly only the flow of energy to the thermal pool. This means we force the creation of a k-packet, whereas before there woud be a choice, but we only take the contribution of the absorption to heating only: i.e. we downweight the packet energy by a factor $f_{\rm down} = (\nu - \nu_0)/\nu$.

In the spectral cycles, interactions with simple bound-free continua now kill the photon, and $k \rightarrow r$ follow the same behaviour as above, because in these cycles we introduce a precalculated band-limited k-packet emissivity.

It is possible for some numerical problems to occur. For example, there is nothing to stop the value of $f_{\rm up}$ being quite large, if the photon is being emitted close to the edge. This is most likely to happen when the electron temperature T_e is quite low, but there is nothing to stop it happening anywhere. This is most likely to lead to problems when the factor $f_{\rm up}$ is comparable to the typical number of photon passages per cell, since then a single photon can dominate the heating or ionization estimators in a given cell and lead to convergence problems by dramatically exacerbating shot noise.

Deactivating the scheme

This mode can be turned off using the *Diag.turn_off_upweighting_of_simple_macro_atoms*. In this case the code will go back to using the two-level framework for simple atom bound free continua.

3.12.3 Special Relativity and Co-Moving Frames

The current version of Python incorporates special relativity and takes co-moving frame effects into account by default.

Global properties of the wind, such as a densities are defined in the observer , or global frame, but are immediately converted to co-moving frame values.

(As an example, if the density of cell and volume (or a cell) in the global frame are ρ_{obs} and V_{obs} then

$$\rho_{cmf} = \frac{\rho_{obs}}{\gamma}$$

$$V_{cmf} = \gamma V_{obs}$$

the product of the two quantities, being a Lorentz invariant.)

Photon generation takes place in the local, or co-moving, frame (of the disk or wind), but photons are immediately converted to the observer, or global, frame for photon transport, allowing both for Doppler frequency shifts and directional correction due to Doppler abberation. The number of photons generated is the number of photons that would be generated in the in one observer frame second. Photons are transported in the observer frame, but coverted back to the local frame within i ndividual wind cells to determine whether the interact with the wind.

Interactions take place in the local frame. Estimators used to calculate, for example, photoionization rates also take place in the local frame. This allows one to calculate ionization fractions in the local frame as is required, since the numbers of ions in a region defined by the edges of the cell must also be Lorentz invariant. Allowances are made for time dilation effects in calculating the rates in the co-moving frame.

For mainly historical and diagnostic reasons, command line options exist to fall back to simple order v/c corrections. `

3.12.4 Anisotropic Scattering

Python has a number of radiative transfer modes, controlled via the *Line_transfer* keyword. Included in this mode is the treatment of line anisotropy; whether re-emission of a line photon is isotropic or not. When the scattering is isotropic, a new direction is simply chosen by choosing a random direction in the co-moving frame of the plasma.

If anisotropic scattering is on, via one of the thermal trapping modes, the new direction is chosen according to a rejection method. The aim is to account for the fact the photon undergoes many interactions in the resonant zone due to the thermal width of the line, and finds it easier to escape along the direction in which the optical depth is lowest (where the velocity gradient is highest). Specifically, the code undergoes the following steps:

- choose a random number between 0 and 1, z
- choose random direction, $\hat{n}$
- evalute Sobolev optical depth along this direction, $\tau_{\hat{n}}$
- calculate the escape probability along this direction $P_{\rm esc}(\tau_{\hat{n}})$.
- If $P_{\rm esc} \geq z$, then escape the loop, otherwise increment a counter `nnscat` and proceed to the first step.

This process is repeated until the loop is exited or 10,000 iterations are reached. The rejection method is trying to sample the volume bounded in θ, ϕ space by the complicated surface $P_{\rm esc}(\theta, \phi)$.

In highly optically thick regions, escape probabilities in all directions can be small, in which case the above rejection method can be extremely inefficient (the volume bounded by $P_{\rm esc}(\theta, \phi)$ is extremely small). Because of this, the code

re-normalises the rejection method by first calculating the escape probability along the maximum velocity gradient, which is the maximum escape probability.

Developer note: the re-normalisation scheme

Describe.

Anisotropy within the viewpoint technique

Within the viewpoint technique (also called extract or the peel-off method), described under *Spectral Cycles*, anisotropy has to be accounted for. At each interaction or wind photon generation, the photon packet is forced along a direction θ, with its weight adjusted according to

$$w_{\rm out} = \frac{P(\theta)}{\langle P(\theta) \rangle} w_{\rm in}.$$

For anisotropic scattering, $P(\theta) \neq \langle P \rangle$. To deal with this, we need to calculate the escape probability along the desired direction, given by

$$P(\theta) = \frac{1 - \exp[-\tau(\theta)]}{\tau(\theta)}$$

where $\tau(\theta)$ is the Sobolev optical depth in a given direction. This is a local quantity evaluated at the point of resonance. $\langle P(\theta) \rangle$ is calculated using a by-product of the rejection method. For a rejection method that samples a properly normalised probability space – a probability space that has a (hyper)volume of 1 – the number of iterations in the rejection method, $N_{\rm it}$ tells us (in this case) about the mean escape probability. More correctly, the expectation value of $1/N_{\rm it}$ is the mean escape probability. Thus, we multiply by a factor of $1/N_{\rm it}$ in the code to account for the $\langle P(\theta) \rangle$ factor in the denominator.

Todo: check the above statement about the expectation value of $1/N_{\rm it}$ is really true – I think it must be, since it's basically the definition of a probability. Does $N_{\rm it}$ also correspond to the actual physical number of scatters?

Developer note

The above calculation is split up within the code. The factor $P(\theta)$ is applied in the function `extract_one`, whereas the division by $\langle P \rangle$ is applied using the variable `nnscat` in extract, which is $N_{\rm it}$ in the above notation. This is because the mean escape probability is (statistically speaking) equal to $1/N_{\rm it}$ as described above.

Note, also, that in practice we have to account for the renormalisation of the rejection method, so rather than multiply by $N_{\rm it}$, we multiply by $N_{\rm it}/P_{\rm norm}$ (see pevious developer note).

3.13 Atomic Data

Any Python model is only as good as the atomic data which goes into making the model. All of the atomic data that Python accepts is read in by the routine get_atomicdata, and all of the data is read in from a series of ascii data files and stored in structures that are defined in atomic.h.

The purpose of documentation is as follows:

- to explain the atomic data formats used by Python and the relationship of different sets of data to one another

- to explain where the data currently used in Python and to explain how the raw data is translated in to a format the Python accepts

The routines used to translate raw data format for two-level atoms (as well as much of the raw data) are contained in a separate github repository These routines are very "rough-and-ready", and not extensively documented, but so users should beware. On the other hand, they are not exceptionally complicated so in most cases it should be fairly clear from the code what the various routines do.

The routines used to generate data for MacroAtoms are described in Generating Macro Atom data

The "masterfile" that determines what data will be read into Python is determined by the line in the parameter file, which will read something like:

```
Atomic_data                                    data/standard80.dat
```

where the file data/standard80.txt will contain names (one to a line) of files which will be read in sequentially. All of the atomic data that comes standardly with Python is stored in the data directory (and its subdirectories) but users are not required to put their data there.

Every line in the atomic data files read by Python consists of a keyword that defines the type of data and various data values that are required for that particular data type. Lines that beging with # or are empty are ignored.

The data from the various files are read as if they were one long file, so how the data is split up into files is a matter of convenience.

However, the data must be read in a logical order. As an simple example, information about elements must be read in prior to information about ions. This allows one to remove all data about, say Si, from a calculation simply by commenting out the line in the atomic data that gives the properties of the element Si, without having to removed all the ion and other information about a data file from the calculation.

The main hierarchy is as follows

- Elements
- Ions
- Energy levels
- Lines

Once these sets of data have been read in the order in which other information is read in is irrelevant, that is one can read the collision data (which is indexed to lines) and photoionization cross sections (which are indexed to energy levels) in either order

(Note that although the approach has advantages of allowing one to comment out atoms or ions, it also has the disadvantageby ignoring data, the program does not give you a particularly good summary of what data has been omitted. If concerned about this one should use the advanced command:

```
@Diag.write\_atomicdata(yes,no)
```

which prints out an ascii version of the input data that was used.

More information on the various types of input data can be found below:

3.13.1 Bound Bound

This is the data for computing bound bound, or line interactions in simple atoms.

Source

The Kurucz data used to create simple lines was taken from the Kurucz website. The file gfall.dat was used, though a few extra lines known to have been missing were likely added.

There are two main sources of data currently used in Python.

- Kurucz
- Chianti

Kurucz is normally used for simple atoms whereas Chianti is the most common source for information about lines used in macro-atom versions We have also used a line list from Verner in the past

Translation to Python format

There are several steps to creating the data used in Python from that in gfall.dat, that are carried out by py_read_kurucz and py_link. The first routine reads the gfall.dat file and creates two output files, a file containing the lines and the associated such as the effective oscillatory strength and a file which contains information about the ion levels. py_read_kurucz chooses only a portion of the Kurucz lines, namely those associated with ions with ionization potentials in a certain range and lines with gf factors exceeding a certain value. The second program py_link attempts to create a model ion with links between the levels and the ions. Both of these routines are driven by .pf files, similar to what are used in python. Examples of the .pf files are in the directory py_kurucz

In practice we have not used these data for any Python publications. At some point early in the AGN project, NSH increased the number of lines, and generated lines_linked_ver_2.py and levels_ver_2.py. I think this was because there was a small bug which meant the oscillator strength cut that was stated was not that which was applied.

Data format

The lines have the following format

For lines, we did not create a specific topbase format, but most of the recent sets of data use a format that is similar to what is need for macro atoms:

```
Line  1  1  926.226013  0.003184   2   4     0.000000    13.387685    0    9
Line  1  1  930.747986  0.004819   2   4     0.000000    13.322634    0    8
Line  1  1  937.802979  0.007798   2   4     0.000000    13.222406    0    7
Line  1  1  949.742981  0.013931   2   4     0.000000    13.056183    0    6
```

whereas for MacroAtoms:

```
# z = element, ion= ionstage, f = osc. str., gl(gu) = stat. we. lower(upper) level
# el(eu) = energy lower(upper) level (eV), ll(lu) = lvl index lower(upper) level
#        z ion       lambda      f         gl  gu    el          eu        ll   lu
LinMacro    1   1          1215.33907            0.41620     2     8            0.
↪00000            10.19883     1     2
LinMacro    1   1          1025.44253            0.07910     2    18            0.
↪00000            12.08750     1     3
LinMacro    1   1           972.27104            0.02899     2    32            0.
↪00000            12.74854     1     4
```

For LinMacro the columns are

- an identifier,
- the element z,
- the ion number,
- the wavelength of the line in A,
- the absorption oscillator strength,
- the lower and upper level multiplicities,
- the energy of the lower level and upper level.

The ultimate source for this information is usually NIST . The main issue with all of this is that one needs to index everything self-consistentl

Python structure

Line data is stored in the data structure **lines**

Comments

The version of gfall.dat that is used in Python is out of date, though whether this affects any of the lines actually used in python is unclear. The website listed above has a link to newer versions of gfall.dat.

3.13.2 Bound-bound electron collision strengths

Source

We use the Chianti atomic database, specifically the *.scups files. These "contain the effective electron collision strengths scaled according to the rules formulated by Burgess & Tully 1992, A&A, 254, 436 The values in the file are functional fits to $\Upsilon(T)$Omega` in our calculations for collisional de-excitation rate coefficient

$q_{21} = \Omega \frac{8.629 \times 10^{-6}}{g_u \sqrt{T}}$

In the g-bar formulation

$\Omega = 4.77 \times 10^{16} g_l \overline{g} \frac{f_{abs}}{\nu}$

These values of Υ simply replace Ω.

In the asbsence of data in this format, the Van Regemorter approximation is utilized.

Translation to Python format

It is necessary to link each line in our line list with the relevant electron collision strength. This is achieved using the python script "coll_stren_lookup.py" which first reads in the "lines_linked_ver_2.py" line list, then attempts to work out which lines are which by comparing the energy and the oscillator strength of the line. If these match to within a factor of 10% then the code logs this as a possible match. If better matches come along, then the code adopts those instead.

Each matched line get a line in the data file which is basically all of the line data for the matched line. This is to give Python the best chance of linking it up with the line internally.

Data format

The collision strength data has the following format:

```
CSTREN Line  1  1 1215.673584  0.139000   2   2     0.000000    10.200121    0    1     ␣
↪ 1      3   7.500e-01   2.772e-01   1.478e+00    5    1   1.700e+00
SCT   0.000e+00   2.500e-01   5.000e-01   7.500e-01   1.000e+00
SCUPS    1.132e-01   2.708e-01   5.017e-01   8.519e-01   1.478e+00
CSTREN Line  1  1 1215.668213  0.277000   2   4     0.000000    10.200166    0    2     ␣
↪ 1      4   7.500e-01   5.552e-01   2.961e+00    5    1   1.700e+00
SCT   0.000e+00   2.500e-01   5.000e-01   7.500e-01   1.000e+00
SCUPS    2.265e-01   5.424e-01   1.005e+00   1.706e+00   2.961e+00
CSTREN Line  1  1 1025.722900  0.026300   2   2     0.000000    12.089051    0    3     ␣
↪ 1      6   8.890e-01   5.268e-02   2.370e-01    5    1   1.600e+00
```

Each record has three lines. The first line has a keyword CSTREN and this contains all the line data for the line to which this collision strength refers as the first 10 fields. These fields are identical to the 10 fields that appear in a standard line file. The next 8 fields are

- 1-2 Upper and lower level of transition - Chianti nomenclature
- 3 Energy of transition - Rydberg
- 4 Oscillator strength x lower level multiplicity (GF)
- 5 High temperature limit value
- 6 Number of scaled temperatures - 5 or 9
- 7 Transition type cite{1992A&A...254..436B} nomenclature
- 8 Scaling parameter (C) (Burgess & Tully 1992) nomenclature

The next two lines, with labels SCT and SCTUPS are the 5 or 9 point spline fits to Υ vs T in reduced units y,x.

There are four different types of transitions, each with their own scaling between temperature of the transition and $Upsilon$

For example, for type 1 (the most common)

$$x = 1 - \frac{lnC}{ln\left(\frac{kT}{E_{ij}} + C\right)}$$

and

$$y(x) = \frac{\Upsilon}{ln\left(\frac{kT}{E_{ij}} + e\right)}$$

So, to get Υ for a given T, one converts T to x via the correct equation,then linearly interpolate between values of y(x), then convert back to $Upsilon$

Python structure

The data is stored in Python in the Coll_stren structure which has memebers

- int n - internal index
- int lower,upper - the Chianti levels, not currently used
- double energy - the energy of the transition
- double gf - the effective oscillator strength - just oscillator strength x multiplicity
- double hi_t_lim - the high temperature limit of y

- double n_points -The number of points in the spline fit
- int type - The type of fit, this defines how one computes the scaled temperature and scaled coll strength
- float scaling_param - The scaling parameter C used in the Burgess and Tully calculations
- double sct[N_COLL_STREN_PTS] -The scaled temperature points in the fit
- double scups[N_COLL_STREN_PTS]- The sclaed coll sttengths in ythe fit

There is also a member in the line structure (coll_index) which points to the relevant record

Comments

There are currenly 4 types of transitions that are read from the Chianti data

- 1 - Allowed transition
- 2 - Forbiddent transitions
- 3 - Intercombination trantions
- 4 - Allowed transitions with

which correspond to the transition types idenitfied by Burgess & Tully

There are addtional transition types in the Chianti database

- 5 - Dielectronic capture tranisitions
- 6 - Proton transitions

The latter are not currently used in **Python**

Discussion of how Chianti handles transitions can be found in The CHIANTI upsilon files (ups and scups)

3.13.3 Bound Free (Overview)

Source

Photonization data in Python is gennerally obtained from two sources

- The Opacity project. See also Cunto et at 1993, A&A, 275, L5

or

- Verner & Yakovlev 1995 : Inner and Outer Shell Cross-sections
- Verner et al. 1996 : Improved Outer Shell Cross-sections-

Details of how each type of photonionzation x-section is treated can be found in

- *Bound Free (Verner)*
- *Bound Free (TopBase)*

3.13.4 Bound Free (TopBase)

Source

Obtained from The Opacity project. See also Cunto et at 1993, A&A, 275, L5

Translation to Python format

ksl - It's not clear that we are now making use of the topbase data in this way but my original attempt to incorporate topbase photoinization data into Python is contained in the directory topbase. Processing of these files was done by py_top_phot. My feeling is that we can replace these remarks with those that are more up to date, once Nick and James discuss this section, and delete any mention of my original attempt on this in the data-gen archive.

Data format

Our original photoionization cross sectiions came from a combination of cite{verner96}, supplemented by a set of older values from cite{verner95}

The TopBase photoionization files have the following format:

```
PhotTopS  1  1 200    1    13.605698  50
PhotTop    13.605698 6.304e-18
PhotTop    16.627193 3.679e-18
```

whereas the Macro Atoms look like:

```
PhotMacS       1       1       1       1       13.598430     100
PhotMac       13.598430   6.3039999e-18
PhotMac       13.942675   5.8969998e-18
```

The meaning of the columns is the same in both cases here. It may be simply a historical accident that we have both formats, or probably we were worried we would need to change. Topbase is generally the source for this information.

The initial line contains (a) the element, (b) the ion, (c) the level, (d) the level up, (e), the energy threshold in eV, and (f) the number of x-sections to be read in. The following lines gives the photon energy in eV, and the cross-section in cm^{2}. To be a valid file the photon x-section must not appear below the energy threshold

"Level up" corresponds to how many levels the electron is moving in the transition: this is simply 1

For the simple atom case, the header line can be parsed as follows

- z: the atomic number
- ionization stage, in the astronomical convention of the lower level ion, the one that gets ionized
- ISLP
- ll: the lower level index in the ion that gets ionized (with that ISLP)
- e: the energy threshold in eV (only photons above this energy ionize)
- npts: the number of points where the cross-section is measured

For the simple atom case the combination ISLP and level is unique

For the macro-atom case the entries in the header line are

- z: the atomic number
- ionization stage, in the astronomical convention of the lower level ion, the one that gets ionized

- ll: the lower level index in the ion that gets ionized (with that ISLP)
- ul: the upper level index in the ion after ionization (usually 1)
- e: the energy threshold in eV (only photons above this energy ionize)
- npts: the number of points where the cross-ection is measured

For the macro atom case, the indices relate to the energy levels, that is these must be connected up properly

The TopBase energies are inaccurate and so generally adjustments are made using Chianti or some other source to fix up the energy levels.

Python structure

The data are stored in the Topbase_phot stucture which can be found in atomic.h

Criteria for usage in Python run

Data has to be read into Python in a logical order. For a set of phototionization x-sections to be accepted, the energy levels (or configuratios) must already have been defiend. See *Levels*

The items that must match are:

- the element (z)
- the ion (istate)
- the upper level, which will be a level in the next ion (ilv)
- the lower level, which will be in the ion that is being photoionized

Comments

The upper level in the MacroAtom case

A common error that creates problems in reading in photoionization x-sections in the MacroAtom case is not to include the next ion up, partiulary the bare ion. If one encounters errors where the upper level is not found, one should check the level file to verify that that the upper level ion is present, and that the inputs allow for the existence of at least the first level of that ion.

For example, if one wishes to read in photoionization x-sections for N VII (hydrogenic), the levels file should include lines like:

```
IonM    N   7   7   2  667.05100 1000   5     1s(2S_{1/2})
IonM    N   7   8   1 1.0000e+20   1   1     Bare
```

The following is incorect:

```
IonM    N   7   7   2  667.05100 1000   5     1s(2S_{1/2})
IonM    N   7   8   1 1.0000e+20   0   0     Bare
```

because although the bare ion is present, the maximum number of levels is set to 0. This is not an issue for the simple atom case.

Extrapolation to higher energies

Some topbase cross-sections do not extend to very high energies, for reasons that are not obvious. This can cause non-physical edges to appear in spectra. Therefore, is is important to inspect any additions to the atomic data based on x-sections retrieved from TopBas

Some tools have been developed To address this probllem. In particularly, JM wrote a script to extrapolate the cross-section to higher energies, by calculating the gradient in log-space at the maximum energy and extrapolating to 100 keV. A number of cross-sections had unrealistic gradients at the original maximum energy, and were identified by eye and then forced to have a ν^{-3} shape. This is the shape of a hydrogenic cross-section and whilst it is not accurate for non-hydrogenic ions, it is more realistic (and conservative) than some of the unphysically shallow gradients that were being found. This is also briefly described in section~3.7.2 of Matthews PhD thesis. The python scripts can be found in the data-gen repository progs/extrapolate_xs/ with docstrings describing their use.

3.13.5 Bound Free (Verner)

This is data for bound free or photoionization data. There is information for both inner shell (auger) and outer shell PI.

Source

There are three sources for this data

- Verner & Yakovlev 1995 : Inner and Outer Shell Cross-sections
- Verner et al. 1996 :Improved Outer Shell Cross-sections
- Kaastra & Mewe 1993 :Electron and photon yield data

Translation to Python format

Tabulation Process

The raw VFKY data comes in a series of fit parameters. We decided, circa Python 78, to tabulate this data instead. Partly, I think I because the on the fly method was time consuming (yes, computing all the pow() commands to commute the cross sections on the fly took a huge amount of time) and we decided that tabulating pre program was better than doing it in the program, so that everything was of the same format.

The script which does this is progs/tabulate_xs/photo_xs.py – it creates a file like photo_vfky_tabulated.data.

Inner and Outer Shells

For the ground states, we split the cross sections up into outer shell and inner shell cross sections. This allows us to calculate possible auger ionization as ions relax after an inner shell ionization. This is done using the python script "verner_2_py.py. This script takes the normal verner cross sections, which truncate at the first inner shell edge and firstly appends the outer shell data from VY to that to make a full outer shell cross section. These are written out into "vfky_outershell_tab.data" It then writes out the inner shell cross sections into "vfky_innershell_tab.data". There is a lot of complicated machinery to try and work out the exact shell that is being ionized to allow these rates to be linked up to the relevant electron yield (and flourescent) records.

Data format

Explain the ascii format of the file which is read into Python

VFKY_outershell_tab.data

Label	z	state	islp	level	threshold_energy	n_points
PhotVfkyS	1	1	-999	-999	1.360e+01	100

This data is linked to the relevant ion via z and state, islp and level are not used. the last number n_points, says how many points are used for the fit, and the next n_points lines in the file, each preceeded by the label PhotVfky are pairs of energy (in eV) vs cross section which make up that fit.

VY_innershell_tab.data

label	z	state	n_shell	l_subshell	threshold_energy	n_points
InnerVYS	3	1	1	0	6.439e+01	100

This data is linked to the relevant ion via z and state. the n_shell and l_subshell numbers are used to cross reference to the electron yield records. As above, the last record shows how many points are in the fit, and the data pairs making up the fit follow with the keyword InnerVY.

Python structure

Where the data is stored internally in Python

Comments

The manner in which this data is read into Python is a bit labyrinthine at the moment. The intention is to use a combination of VFKY and VY for all ground states, an

3.13.6 Direct Ionization

This is the data to compute ionization rates from collisions between ions and hot electrons.

Source

The data comes directly from Dere 2006, A&A, 466, 771 . This paper gives direct ionization and excitation-autoionization rate coefficients for many ions as a function of temperature for Maxwellian electron distributions.

Translation to Python format

The data table is downloaded in its entirety from the data table associated with the paper. All that happens is that the table is saved to a text file, and the keyword DI_DERE is just prepended to each row.

Data format

Each line starts with the label DI_DERE and then follows

- Nuclear Charge - z - used to identify the ion
- Ion - state in our normal notation, so 1=neutral
- Number of splines N- the number of spline points for the fit of rate coefficients vs scaled temperature
- Scaled temperatures - there are N of these
- Scaled Rate coefficients - N of these

The scaled temperatures are given by

$x = 1 - \frac{\log f}{\log(t+f)}$

where t=kT/I. I is the ionization potential, and f=2.0. The rate coefficient R(T) is recovered from the scaled rate coefficient in the table, rho using

$\rho = t^{1/2} I^{3/2} R(T)/E_1(1/t)$

where E_1 is the first exponential integral. In python we use the gsl_sf_expint_E1 routine in gsl.

Python structure

This data is stored in the dere_di_rate structure with members

- int nion- Internal cross reference to the ion that this refers to
- int nspline - the number of spline points that the fit is evaluated over
- double temps[DERE_DI_PARAMS]- temperatures at which the rate is tabulated
- double rates[DERE_DI_PARAMS]- rates corresponding to those temperatures
- double xi - the ionization energy of this ion
- double min_temp -the minimum temperature to which this rate should be applied

Comments

This data is also in Chianti , although in a different form. So we could potentially use this data as part of a push to just use Chianti for all our data uses. An updated set of DI data is available here

3.13.7 Auger Electron Yields

This data is linked with the inner shell photoionization data. It gives probabilities for different numbers of electrons to be ejected following inner shell ionizations.

Source

This data comes from Kaastra and Mewe 1993, A&A, 97, 443 . The data is downloaded from the vizier site linked and put into a file called "electron_yield.data"

Translation to Python format

The translation takes place using the python script "kaastra_2_py.py" which takes the saved raw data file "electron_yield.data" and compares it line by line to the inner shell cross section data in "vy_innershell_tab.data"(see above). The n shell and l subshell to which each record applies is coded in the KM data and needs to be decoded. This is what the script does, and all the script then does is output the yield data into a new file "kaastra_electron_yield.data" which contains the n and l cross reference.

Data format

This is the data format of the electron yield data

Label	z	state	n	l	IP(eV)	<E>(eV)	P(1e)	P(2e)
Kelecyield	4	1	1	0	1.15e+02	9.280e+01	0	10000
Kelecyield	5	1	1	0	1.92e+02	1.639e+02	6	9994
Kelecyield	5	2	1	0	2.06e+02	1.499e+02	0	10000

The data is linked to the correct inner shell photoionization cross section (and hence rate) via z, state, n shell and l subshell. The IP is not used. <E> is the mean electron energy ejected, used to calculate the PI heating rate in radiation.c. The last ten columns in the file (2 shown in the table above) show the chance of various numbers of electrons being ejected in units of 1/10000.

Python structure

The data is stored in python in the inner_elec_yield structure which contains

- int nion - Index to the ion which was the parent of the inner shell ionization
- int z, istate - element and ionization state of parent ion
- int n, l - Quantum numbers of shell
- double prob[10] - probability for between 1 and 10 electrons being ejected
- double I - Ionization energy
- double Ea - Average electron energy

Comments

3.13.8 Elements and Ions

The first file that must be read into **Python** is the file that defines the elements and ions. The

Source:

This data comes from Verner, Barthel & Tytler, 1994, ApJ 108, 287.

Translation to python:

The original data and the translation can be found in py_verner. A simple awkscript converts the downloaded data to Python format.

Data Format

There are two sections to the file, first elements are defined

The first portion of a typical file is as follows:

```
# Source Verner, Barthel & Tytler 1994 A&AS 108, 287
Element    1    H   12.00     1.007940
Element    2   He   10.99     4.002602
#Element    3   Li    3.31     6.941000
#Element    4   Be    1.42     9.012182
#Element    5    B    2.88    10.811000
Element    6    C    8.56    12.011000
Element    7    N    8.05    14.006740
Element    8    O    8.93    15.999400
```

And the columns are as follows

- A label for this type of data entry
- The z of the elment
- The common abbreviation for the elemen
- The atomic weight of the element

Lines beginning with # (and empty lines) are treated as comments. So in this case, Li, B and B are ignored, because of their relatively low abundance.

Abundances are generally defined logarithmically with respect to H at 12.00. In principle, there are two choices if one wished to defien a plasma where, for example, He was the dominant element. One could leave the H abundance at 12 and define the He abundance as for example 13.00 Alternatively, one could set the He abundance to 12.00 and define all of the other elements with respect to this. Either choice should work but none has been tested. It is unclear whether code will work at all for a plasma with no H.

The ion section (which could be in a separate file) has the following format:

```
IonV    H   1   1   2   13.59900 1000  10     1s(2S_{1/2})
IonV    H   1   2   1 1.0000e+20   1   1     Bare

IonV   He   2   1   1   24.58800 1000  10     1s^2(1S_0)
IonV   He   2   2   2   54.41800 1000   10     1s(2S_{1/2})
IonV   He   2   3   1 1.0000e+20   1   1     Bare
```

and the columns have the following meaning

- Label for an ion that will be treated as a simple ion
- The common abbreviation for the element

- z of the ion
- ionization state of the ion
- g-factor for the ground state
- the ionizaton potential in eV
- maximum number of (simple) levels allowed
- maximum number of nlte (macro-atom) levels
- the configurations (which for information only)

The label for the ion entries determines whether an element will be treated as simple atom or as a macro-atom. For case where H is to be treated as a macro atom, but He is to be treated as a simple atom, this file would become:

```
IonM    H   1   1   2   13.59900 1000  10     1s(2S_{1/2})
IonM    H   1   2   1 1.0000e+20   1   1     Bare

IonV   He   2   1   1   24.58800 1000  10     1s^2(1S_0)
IonV   He   2   2   2   54.41800 1000   10     1s(2S_{1/2})
IonV   He   2   3   1 1.0000e+20   1   1     Bare
```

Note that only evident changed is the label, but in this case the number of nlte levels, and not the number of levels is what is important.

Python structure:

This data is held in Python in various fields in structures **elements** and **ions**.

Comments:

Maximun numbers of levels

As indicated the numbers here are maximum values, and the actual numbers of levels for particular ion will depend on the data that follows. One can use the numbers here to limit the complexity of, for example, a macro-atom to see whether making a more complicated macro-atom affects the reusult of a calculation. One does not need to change the "downstream" data to make this happen, **Python** will simply ignore the extra data.

3.13.9 Free-Free Emission

Source

The free-free Gaunt factors are taken from Sutherland 1998, MNRAS, 300, 321. The data is available for download here where three files exist

- gffew.dat : Free-Free Emission Gaunt factors as a function of scaled electron and photon energy.
- gffgu.dat : Free-Free Emission Gaunt factors for Maxwellian electrons as a function of scaled temperature and scaled frequency.
- gffint.dat : Total Free-Free Emission Gaunt factors for Maxwellian electrons.

The last file is the one we use to calculate free-free emission, since this in integrated gaunt factor over a population of electrons with a Boltzmann distribution of velocities. The other two files could be of use in the future should we wish to have gaunt factor corrections for the heating rates,in which case we should use the gffgu.dat data file. However

generally speaking free-free heating is never important and there would be significant overhead in calculating a gaunt factor for each photon.

Translation to python

The file is simply modified by hand to put a label "FF_GAUNT" at the start of each data line and a hash at the start of each comment line.

Datafile - gffint.dat:

The format of the data file to be read into python is as follows:

Label	$\log(\gamma^2)$	$< g_{ff}(\gamma^2) >$	s_1	s_2$	s_3
FF_GAUNT	-4.00	1.113E+00	1.348E-02	1.047E-02	-1.855E-03
FF_GAUNT	-3.80	1.117E+00	1.744E-02	9.358E-03	5.565E-03
FF_GAUNT	-3.60	1.121E+00	2.186E-02	1.270E-02	4.970E-03

where γ^2 is the "scaled inverse temperature experienced by an ion" and the other four numbers allow the free-free Gaunt factor to be computed at any scaled inverse temperature

$x = \frac{Z^2}{k_B T_e} \frac{2\pi^2 e^4 m_e}{h^2}$

through spline interpolation between the two bracketing values of $\log(\gamma^2)$

$< g_{ff}(x) >=< g_{ff}(\gamma^2) > +\Delta\left[s_1 + \Delta\left[s_2 + \Delta s_3\right]\right]$

where

$\Delta = \log(x) - \log(\gamma^2)$

Python structure

This data is held internally in Python in the structure **gaunt_total** which has members

- log_gsqrd
- gff
- s1, s2, s3

Comments

We currently just use the total free free emission gaunt factor as a function of temperature for a Maxwellian distribution of electrons. This is OK for cooling, however we should really use the frequency dependant gaunt factor for free free heating. If we ever have a model where free-free heating is dominant, this should be looked into.

3.13.10 Levels

Once the element and ion data has been read into textsc{Python}, the next step is to read in the level information.

Source:

Level information can be derived from a variety of sources, by:

- deriving it from a line list such as that of Kurucz, or
- more commonly, for data abstracted from TopBase or Chianti

Translation to python:

Various Formats of Level Files

The original format:

```
Comment-- Ion H  1
# There are 17 unique levels for 1 1
Level   1   1   0    2   0.000000
Level   1   1   1    2  10.200121
Level   1   1   2    4  10.200166
Level   1   1   3    2  12.089051
```

where the colums are z, istate (in conventional notation), a unique level no, the multiplicity of the level and the excitation energy of the level in eV

Level files with this type of format are used in files such as levels_kur.dat, when the levels are derived from linelists such as Kurucz. It is only allowable when dealing with simple atoms.

The current format:

```
# Maximum excitation above ground (in Ryd)  for inclusion  4.000000
# Miniumum excitation below continuum (in Ryd) for inclusion -0.100000
# Topbase levels: Order changed to move config to end of line
#LevTop  z ion  iSLP iLV iCONF   E(eV) Te(eV) gi RL(s) eqn RL(s)
# ================================================================================
#      i NZ NE iSLP iLV iCONF                E(RYD)      TE(RYD)   gi     EQN    RL(NS)
# ================================================================================
# ================================================================================
LevTop  1  1  200  1 -13.605698   0.000000  2  1.0000 1.00e+21 () 1s
LevTop  1  1  200  2  -3.401425  10.204273  2  2.0000 1.00e+21 () 2s
LevTop  1  1  211  1  -3.401425  10.204273  6  2.0000 1.60e-09 () 2p
```

whereas the for Macro Atoms we have:

```
#         z ion lvl ion_pot   ex_energy   g  rad_rate
LevMacro  1  1  1 -13.59843    0.000000   2  1.00e+21 () n=1
LevMacro  1  1  2  -3.39960    10.19883   8  1.00e-09 () n=2
LevMacro  1  1  3  -1.51093    12.08750  18  1.00e-09 () n=3
```

The columns are similar in the two cases.

Each level is described by an element number and ion number and a level number. In the macro-atom case the level number is unique; in the simple atom case the combination of iSLP and the level number are unique.

For the Topbase case for simple atoms, the columns are:

- the atomic number
- the ion number in the usual astronomical convention
- iSLP
- the level number
- the energy in eV relative to the continuum
- the energy in eV relative to the ground state
- the multiplicity (g) of the level, usually 2J+1
- the equivalent quantum number
- the radiative lifetime
- the configuration

There are some specific differences. In particular, for LevMacro levels, the excitation energy needs need to be on an absolute scale between ions, and so it includes the ionization energy of the lower level ionization states. Note that the radiative rates are not used. The original intention was to use this to define the difference between metastable and normal levels, with the expectation that if the level was metastable it would be put in Boltzmann equilibrium with the ground state. Right now python uses 10**15 seconds, essentially a Hubble time to do this, but this portion of the code is not, according to ss, tested.

The primary source for this is usually the NIST database, although similar information is usually available in Chianti. One normally wants text output, and eV to describe the levels, and then one needs to put things in energy order. Since they quote J, one converts to g = 2J+1

The ionization potential is not used, as it is redundant with the excitation energy which is, and the last column giving the configuration is also for information only.

Python structure:

This data is held in Python in various fields in structure **config**.

Comments:

3.13.11 Auger Photon Yields

When inner shell (or Auger) ionization takes place - there is a chance of photons being eected as the inner shells are re-filled. This data provies the information to compute the photons thus made. It is currently not used.

Source

This data comes from Kaastra and Mewe 1993, A&A, 97, 443 . The data is downloaded from the vizier site linked and put into a file called “fluorescent_yield.data”

Translation to Python format

The translation takes place using the python script "kaastra_2_py.py". All identical to electron yield, but input file is "fluorescent_yield.data" and output is "kaastra_fluorescent_yield.data"

Data format

This is the data format of the electron yield data

Label	z	state	n	l	photon_energy(eV)	yield
Kphotyield	5	1	1	0	1.837e+02	6.000e-04
Kphotyield	5	1	1	0	1.690e+01	7.129e-01
Kphotyield	6	1	1	0	2.768e+02	2.600e-03

The data is linked to the correct inner shell photoionization cross section (and hence rate) via z, state, n shell and l subshell. The photon energy field is thew energy of the fluorescent photon in eV, and yield is the number of said photons emitted per ionization multiplied by 10^4.

Python structure

The data is stored in python in the inner_fluor_yield structure which contains

- int nion - Index to the ion which was the parent of the inner shell ionization
- int z, istate - element and ionization state of parent ion
- int n, l - Quantum numbers of shell
- double freq - the rest frequency of the photon emitted
- double yield - number of photons per ionization x 10^4

Comments

This data is not currently used

3.14 Meta-documentation

3.14.1 How to document Python

This documentation is written in **ReStructured Text**, and parsed by **Sphinx**. A general guide to **ReStructured Text** can be found here. We're trying to maintain a roughly consistent format for the documentation.

Installing the documentation tools

This guide is produced using **Sphinx**. **Sphinx** is written in **python** and available from the **pip** package manager. We require the **Python 3** version of **Sphinx**. Install it, and the other modules required, as:

```
cd docs/sphinx
pip3 install -r requirements.txt
```

Building the documentation

Once **Sphinx** is installed, you can make the documentation using a **Makefile** as:

```
make html
```

You can tell if the documentation was built successfully by looking at the output of `make html`. You should see:

```
build succeeded.

The HTML pages are in html.
```

If the build was successful then the documentation can be viewed by opening `docs/sphinx/html/index.html`. Many errors will not stop the build process. Details on the build errors can be found in the section on *Common errors & warnings*.

You can make minor changes to the documentation and recompile using `make html` again. If you add new pages or move existing ones, the table of contents will need to be regenerated. Do this via:

```
make clean
make html
```

3.14.2 General documentation

Conventions

Each file should have a title, with subsections within it created by underlining the titles as:

```
Title
#####

Section
=======

Subsection
----------

Subsubsection
^^^^^^^^^^^^^
```

When referring to a parameter, link to the documentation page as:

```
The number of domains can be set by :ref:`Wind.number_of_components`.
```

When referring to files, code (e.g. shell script) or values used by the code, render it as `monospaced` text as:

```
Run the program using ``py``.
Set the parameter :ref:`Wind.type` to ``SV``.
Outputs can be found in ``filename.rst``.
```

When referring to a library or program name, render it in **bold** as:

```
Though this program is called **Python**, it is written in **C**, using the **GSL**␣
↪library.
```

Content of interest to developers but not users should be broken out into a callout as:

```
.. admonition :: Developer Note

    This value is only stored to single-precision
```

Developer Note

This is a developer note

Documentation that needs expanding should be indicated with a to-do callout as:

```
.. todo :: Expand this section
```

Todo: This is a to-do note

Content relating to a specific **GitHub** issue/pull request can be linked directly to it as #1/#56:

```
This arose due to issue :issue:`1`, which was fixed by :user:`kslong` using :pr:`56`.
```

When writing a table, use the full form where possible as:

```
+----+----+
|Name|X   |
+----+----+
```

Name	X

3.14.3 Parameter documentation

Formatting

Parameters are documented in a consistent way. They have a set of properties. Not every parameter will have all properties but you should fill them all in where possible. A full example outline is:

```
Title
=====
Description.
Use :ref:`Parameter.name` to link to other parameters, or other pages within the␣
↪documentation.
```

(continues on next page)

(continued from previous page)

```
Type
  Enumerator

Values

  option
    Description
    Multi-line if desired

  other
    More description

    Child(ren)
    * :ref:`Corona.radmin`

  yet_another
    More description

    Child(ren)
    * :ref:`KWD.rmin`
    * :ref:`KWD.rmax`

File
  `filename.c <https://github.com/agnwinds/python/blob/master/source/filename.c>`_

Parent(s)
  * :ref:`System_type`: `agn`, `binary`
```

The sections we expect are entered as a definition list. A definition list consists of titles followed by a definition block indented by 2 characters. The headings, in the order we expect, are:

Name
: The parameter name, as used by Python input files.

Description
: A description of the parameter and its function. This can include links to other pages and parameters, using the format

```
Use :ref:`Parameter.name` to link to other parameters, or other pages within the␣
↪documentation.
```

Type
: This is whether the parameter is an integer, float, or enumerator (a list of choices).

Unit
: This is the unit. It can be something like `cm`, `m` or even derived from other parameters (e.g. *Central_object.radius*).

Values
: If the parameter is an integer or float, this should describe the range of values it can take. For example, `Greater than 0` or `0-1`.

  If the variable type is `Enumerator`, then instead it should include a nested definition list of the possible choices. Where each choice implies a different set of possible children (e.g. *Wind.type*) then each choice should have its

own **Children** definition list, e.g.

```
SV
  * :ref:`SV.thetamin`
  * :ref:`SV.thetamax`
```

File
The file the parameter is found in. This is a link to the file on the *master* branch.

Child(ren)
If the parameter implies any others. For example, *Spectrum.no_observers* has child parameters *Spectrum.angle*.

Parent(s)
If the parameter depends on another. For example, *KWD.rmax* is only required for a specific choice of *Wind.type*.

Locations

Parameters are stored in `docs/sphinx/source/inputs/parameters/`.

If multiple parameters share a root (i.e. `SV.radmin`, `SV.radmax`), then they should be stored within a directory with the same root name as the parameters (i.e. `SV/SV.radmin.rst`, `SV/SV.radmax.rst`). In the level above that directory, there should be a `.rst` file with the same name that serves to link those files to the table of contents, as:

```
SV
==

Some description of the parameter group.

.. toctree::
   :glob:

   SV/*
```

Storing all the parameters in one folder would result in it being unreadably busy. Instead, we sift the parameters into groups. Where multiple different parameters or parameter folders fall into the same rough category (e.g. central object parameters, wind types and the like) we create subfolders to group them into. The order that these appear in the sidebar can be set if you enter the filenames explicitly in the `docs/sphinx/source/input/parameters.rst` file.

3.14.4 Common errors & warnings

Undefined Label

```
/path/to/file.rst:line_number:
WARNING: undefined label: label_name (if the link has no caption the label must␣
↪precede a section header)
```

This warning occurs when the `:ref:'location'` format is used to link to a section that does not exist. Check the spelling

Duplicate Label

```
/path/to/file.rst:line_number:
WARNING: duplicate label label_name, other instance in /path/to/other/file.rst
```

This warning occurs when two sections have the same name. The **autosectionlabel** addon automatically creates a label for each title and section component. This is generally not a problem unless you *really* need to

Inline emphasis

```
/path/to/file:line_number:
WARNING: Inline emphasis start-string without end-string.
```

This warning occurs when a line contains an un-escaped * character, as * is used to denote *emphasis*. Either escape it with \ (i.e. *) or wrap it in a :code: tag.

Bullet list ends without a blank line

```
/path/to/file.rst:line_number:
WARNING: Bullet list ends without a blank line; unexpected unindent.
```

This warning occurs when a bullet-list doesn't have a blank line between it and the next bit of text. It commonly happens when you accidentally forget to space a bullet and the text following it, e.g.

```
* blah1
* blah2
*blah3
```

Inline substitution_reference

```
/path/to/file:line_number:
WARNING: Inline substitution_reference start-string without end-string.
```

This warning occurs when you have a word immediately followed by a pipe that is not part of a table, e.g. `something|`. It tends to occur during typos in table creation e.g.

```
+---+---+
| a||b  |
+---+---+
```

3.14.5 Documenting Python Scripts

The *Python Scripts* page is intended to document various python scripts contained within the py_progs folder. The aim is to do this using Sphinx's autodoc extension, invoked by adding `sphinx.ext.autodoc` to extensions list in the conf.py file. py_progs is also added to the path using `sys.path.insert(0, '../../py_progs/')`.

The above link contains full documentation of the commands. A module in py_progs can be documented by adding the following text to the rst file, where module.py is the name of the module you wish to document.

```
.. automodule:: py_read_output.py
    :members:
```

For this to work properly, docstrings have to be in a reasonable rst format. We might consider using the napoleon extension if this is not to our taste.

3.15 Developer Documentation

This page contains documentation intended for developers.

3.15.1 Programming Notes

Python is written in C and is normally tested on linux machines and on Macs, where the compiler usually turns out to be clang. It is also regularly compiled with gcc as part of the travis-CI tests. Certain portions of the code are parallelized using the Message Parsing Interface (MPI).

Version control is (obviously) managed through **git**. The stable version is on the *master* branch; the main development is carried out on the *dev* branch. This is generally the branch to start with in developing new code. If possible, a developer should use the so-called Fork and Pull model for their version control workflow. See e.g. this gist post.

If one modifies the code, a developer needs to be sure to have `$PYTHON/py_progs` both in `PYTHONPATH` and `PATH`. One should also have a version of indent installed, preferably but not necessarily gnu_indent installed. This is because, the Makefile will call a script run_indent.py on files that the developer has changed, which enforces a specific indent style on the code.

In addition to indent, one should have cproto or something equivalent installed. cproto is used to prototypes for all of the subroutines (through the make command

```
make prototypes
```

(The many warnings that appear when cproto is run on OSX can so far be ignored. cproto for macs is available with brew)

All new routines should have Doxygen headers.

printf statements should be avoided, particular in the main code. Python has its own replacements for these commands, e.g Log and Error which standardize the outputsand allow for managing what is printed to the screen in multprocessor mode. There is alsoa command line switch that contorls the amount of information that is printed to the screen. Specific errors are only logged for a limited number of times, after which theyare merely counted. A log of the number of times each error has occurred is printed outat the end of each run and for each thread. (Additional detailes can be found in the Doxygenheader for xlog.c)

Structures

In order to understand the code, one needs to understand the data structures.

The main header files are:

- atomic.h - This contains all of the structures that hold atomic data, e.g oscillator strengths, photoionization cross-sections, elemental abundances, etc. These data are read in at the beginning of the program (see atomicdata.c and other similarly named routines)
- python.h - This contains the structures and other data that comprise the wind as well as the parameters of the model. (This is fairly well-documented, or should be)

Program Flow

Basically, (as decribed from a more scientific perspective elsewhere), the program consists of a number stages

- Data gathering and initialization: This consiss of reading in all of the parameters for the model to be calculated, reading in the associated atomic data, and setting up program to run. This procuess involves allocating space for many of the data structures.
- Ionization cycles: During this portion of the program fleets of photons are generated and propogated through the wind, interacting with it in various ways. These photons are generated over a large range of frequencies, because their purpose is to allow the program to determine the ionization state of the wind. During this process various estimators are accumulated that describe the interaction of the photons with the wind. Once all of the photons have propagated through the wind the various estimators are used to calculate a new estimate of the ionization state of the various wind cells that constitute the wind. This process is repeated for a number of cycles, by which time, hopefully, the wind will have reached a "steady state".
- Spectral cycles: Once the ionization cycles have been completed, the ionization state of the wind is fixed, and more detailed spectra are calculated. For this, photons are generated in a limited spectral range, depending on the interests of the user. In contrast to the ionization state, where "cycles" are crucial concept, the only reason to have spectrl cycles in the "Spectral cycle" phase is to allow one to gradually improve the statistics of the spectrum. At the end of each spectral cycle, the detailed spectra are written out, so one can see the spectra building up.

Parallel Operation

Python uses MPI to parallelize the most compute intensive portions of the routine. It has been run on large machines with 100s of cores without problem.

The portions of the routine that are parallelize are:

- Photon generation and transfer: When run in multiprocesser mode, each thread creates only a fraction of the total number of photons. The weight of the photons in each thread is such that the sum of the weights is the total energy expected to be produced in one observer frame second. These photons are propagated through the wind, and estimators based on these photons are accumulated. At the end of photon transfer by all threads, the various quantities, including the spectra, that have been accumulated in the separate threads are gathered together and averaged or summed as appropriate. For ionization cycles, this means that all of the data needed to calculate the ionization in any cell is available on each of the threads.
- Ionization calculation: Although all of the threads have all of the data needed to calculate the ionization in any cell, in practice what happens is that the program assigns a different set of cells to each thread to calculate the ionization. After the thread calculates the new ionization state for its assigned cells, the ionization states are then gathered back and broadcast to all of the threads, in preparation for the next cycle.
- Preparation for detailed radiative transfer in the macro-atom mode. When photons go through the grid in the simple-atom mode, photon frequencies do not change a great deal, however in macro-atom mode the frequencies can shift by large amounts. This creates a problem during the detailed spectral generation stage, because one does not know before hand how many photons that started out of the desired band end up in the desired band. (In macro-atom mode, a photon bundle that starts out at 8 keV photoionizes an Fe ion can turn (for example) into an Hbeta photon). To handle this, one needs to estimate how often this happens and include this (effectively as a source function) in radiative transfer involving macro-atoms. This is parallelized, in the same manner as the ionization calculation by assigning various cells to various threads and gathering the results back before the radiative transfer step in the detailed spectrum phase.

MPI requires intialization. For python this is carried out in python.c. Various subroutines make use of MPI, and as a result, programmers need to be aware of this fact when they write auxiliary routines that use the various subroutines called by Python.

Input naming conventions

As is evident from an inspection of a typical input file, we have adopted a somewhat hierarchical scheme for the naming of the input variables, which groups variables associated with the same part of the system together. So for example, all of the variables associated with the central object have names like:

```
### Parameters for the Central Object
Central_object.mass(msol)                    0.8
Central_object.radius(cm)                    7e+08
Central_object.radiation(yes,no)                     yes
Central_object.rad_type_to_make_wind(bb,models)                      bb
Central_object.temp                          40000
```

that is, they all begin with Central_object. This convention should be followed.

External variables

Python uses lots (and likely too many), what are properly know as external variables. (In C, a global variable is a variable whose scope is all of the routines in a speciric file. An external varriable is one that is shared across multiple files.)

In the latest generations of gcc, the standards for extenral variiables have been tightened.

If one wishes to define an external variable, one must first declare it as eternal, and then one must initialize it outside a specific routine exactly in one place.

The standard convention is that the variables are declared as external in a header file, e.g python.h, and then intialized in a separate .c file, e.g python_extern_init.c. Unless, a variable is actually initialized, no space will be allocated for the variable.

So if variables are added (or subtracted), one must make a change both in the relavant .h file.

Currently has three.c files atomic_extern_init.c, models_extern_init.c, python_extern_init.c corresponding to the three main .h files, atommic.h, models.h and python.h

3.15.2 Matrix Acceleration using CUDA

Python can use CUDA/GPUs to accelerate solving linear systems using the cuSOLVER library, which is part of the NVIDIA CUDA Toolkit.

This pilot study into using GPUs in Python was conducted as an HPC RSE project at the University of Southampton.

When should you use CUDA?

Given the pilot study nature of this work, the current matrix acceleration implementation (September 2023) is simple. In most small to mid-sized models, using GPU acceleration will not improve model performance. As of writing, CUDA is only used to accelerate matrix calculations specifically, so there are no performance improvements other than in models which are bogged down by matrix calculations; e.g. such as when calculating the ionization state for a large number of ions, or models with lots of macro atom levels/emissitivies. Even so, there may only be modest improvements in model performance if the majority of time is spent transporting and scattering photons.

It is therefore only preferable to use CUDA when matrices are large enough to warrant GPU acceleration. The size of where this is important is tricky to know, as it is hardware dependent - both on your CPU and GPU. If you are using an old CPU, then you are likely to see improvements from matrices as small as circa 200 x 200. Otherwise, you may need to reach up to matrix sizes of 500 x 500 (or larger!) before there is any tangible benefit.

This situation will be improved with smarter memory management and further parallelisation. Most time is spent copying data back and forth between the GPU and CPU. As an example, consider the matrix ionization state calculation. Currently only the actual step to stop the linear system (to calculate the ionization state) has been ported to the GPU. This means each iteration toward a converged ionization state requires memory to be copied to and from the GPU, which slows things down quite a bit. If you could instead port the entire iterative procedure to the GPU (which is not that easy), there is no longer the need to make expensive memory copies each iteration which will significantly speed up the algorithm.

Requirements

To use the CUDA matrix acceleration in Python, your machine needs to have the following installed,

- A CUDA-capable NVIDIA GPU
- NVIDIA CUDA Toolkit
- NVIDIA GPU drivers
- A supported operating system (Windows or Linux) with a gcc compiler and toolchain

NVIDIA provides a list of CUDA-enabled GPUs here. Whilst the GeForce series of NVIDIA GPUs are more affordable and generally *good enough*, from a purely raw computation standpoint NVIDIA's workstation and data center GPUs are more suited due differences (and additional) in hardware not included in the GeForce line of GPUs.

Installing the CUDA toolkit

The NVIDIA CUDA Toolkit is installed either through an installer downloaded from NVIDIA or can be installed via a package manager on Linux systems. It should be noted that the CUDA Toolkit *does not* come with NVIDIA drivers and need to be installed separately. The NVIDIA CUDA Toolkit is available at https://developer.nvidia.com/cuda-downloads and NVIDIA drivers at https://www.nvidia.co.uk/Download/index.aspx.

On Ubuntu 22.04, the toolkit and NVIDIA's proprietary drivers are available through `apt`,

```
sudo apt install nvidia-cuda-toolkit nvidia-driver-535
```

How to Enable and Run CUDA

Compilation

CUDA is an additional acceleration method and is therefore not enabled by default. To enable CUDA, Python has to be compiled with the additional `-DCUDA_ON` flag and linked with the appropriate libraries using the NVIDIDA CUDA compiler (nvcc). There are several ways to enable the CUDA components of Python. The most simple is to run the configure script in the root Python directory with the arguments `--with-cuda`,

```
[$PYTHON] $ ./configure --with-cuda

Configuring Makefile for Python radiative transfer code
Checking for mpicc...yes
Checking for gcc...yes
Checking for nvcc...yes
Preparing Makefile...Done.
```

If the NVIDIA CUDA Toolkit is found, you will see the output informing that the CUDA compiler `nvcc` was found.

What essentially happens when you run `code` is that a value for the variable `NVCC` is set in the Makefile in Python's source directory. If you re-run `configure` without `--with-cuda`, then `NVCC` will be unset and CUDA will not be used. CUDA can be disbaled or enabled *"on the fly"* by modifying this variable without running the configure script and by modifying the Makefile or passing the value of the variable when calling the Makefile, e.g.,

```
[$PYTHON/source] $ make clean
[$PYTHON/source] $ make python NVCC=nvcc
```

`make clean` has to be run whenever CUDA is switched been enabled or disabled, due to code conditionally compiling depending on if CUDA is enabled or not.

Running

To run Python with CUDA, you run it in the exact way even parallel models running with MPI. On a HPC system the appropriate GPU resources will need to be requested in a job submission script. For example, on Iridis at the University of Southampton, a functional job submission script may look like this,

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time=06:00:00
#SBATCH --partition=gpu

module load openmpi/4.1.5/gcc

mpirun -n $SLURM_NTASKS py model.pf
```

If CUDA is enabled and no GPU resources are found, Python will exit early in the program with an appropriate error message. Note that a CUDA-aware MPI implementation is not required, as no data is communicated between GPUs.

Implementation

In this part of the documentation, we will cover the implementation details of cuSolver in Python. cuSolver is a matrix library within the NVIDIA CUDA ecosystem, designed to accelerate both dense and sparse linear algebra problems, including matrix factorisation, linear system solving and matrix inversion. To use cuSolver, very little GPU specific code needs to be written, other than code to allocate memory on the GPU. There are therefore a number of similarities between writing functions which use the cuSolver (and other CUDA mathematical libraries) and GSL libraries.

The CUDA parallel model

The main difference between CPU and GPU parallel programming is the number of (dumb) cores in a GPU. Whereas on a CPU where we divide work on a matrix into smaller chunks, on a GPU it is realistic to have each core of the GPU operate on a single element of the matrix whereas a CPU will likely have multiple elements. CUDA is a type of shared memory parallel programming, and at its core are kernels, which are specialised functions designed for massive parallelism. These kernels are executed by each thread (organized in blocks and grids), where thousands are launched and execute the code concurrently allowing for massive parallelism.

As an example, consider matrix multiplication. If the calculation is parallelised, each CPU core will likely need to calculate the matrix product for multiple elements of the matrix. On a GPU, each thread that is launched will calculate the product for only a single element. If there are enough GPU cores available, then the calculation can be done in effectively a single step which all threads calculating the product for each element at once.

A more detailed and thorough explanation of the CUDA programming model can be found in the CUDA documentation.

Basics

Python uses the dense matrix functions in cuSolver, which are accessed through the `cusolverDn.h` header file. To use cuSolver, it must first be initialised. To do so, we use `cusolverDnCreate` to create a `cuSolverDnHandle_t` variable which is used by cuSolver internally for resource and context management.

cuSolver is based on the Fortran library LAPACK and as such expects arrays to be ordered in column-major order like in Fortran. In C, arrays are typically ordered in row-major order and so arrays must be transposed into column-major ordering before being passed to cuSolver (an explanation of the differences between row and column major ordering can be found here). Matrices can be transposed either whilst still on the CPU, or on the GPU by using a CUDA kernel as shown in the example below,

Listing 1: A CUDA kernel to transpose a matrix from row to column major

```
__global__ void  /* __global__ is used by kernels, all of which return void */
transpose_row_to_column_major(double *row_major, double *column_major, int matrix_size)
{
    /* Determine the x and y coordinate for the thread -- these coords could be
       outside the matrix if enough threads are spawned */
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;

    /* Only transpose for threads inside the matrix */
    if (idx < matrix_size && idy < matrix_size) {
        column_major[idx * matrix_size + idy] = row_major[idy * matrix_size + idx];
    }
}
```

The syntax of the above is covered in detail in the CUDA documentation. The purpose of the kernel is take in a row major array and to transpose it to column major.

Every cuSolver (and CUDA) function returns an error status. To make code more readable, a macro is usually defined which checks the error status and raises an error message if the function does not execute successfully. This type of macro is used extensively throughout the implementation.

Listing 2: A useful macro for error checking cuSolver returns

```
#define CUSOLVER_CHECK(status)                                                 ␣
↪                              \
    do {                                                                       ␣
↪                              \
        cusolverStatus_t err = status;                                         ␣
↪                              \
        if (err != CUSOLVER_STATUS_SUCCESS) {                                  ␣
↪                              \
            Error("cuSolver Error (%d): %s (%s:%d)\n", err, cusolver_get_error_
↪string(err), __FILE__, __LINE__);   \
            return err;                                                        ␣
↪                              \
        }                                                                      ␣
↪                              \
```

(continues on next page)

(continued from previous page)

```
    } while (0)

/* Here is an example of using the macro to create a handle */
CUSOLVER_CHECK(cusolverDnCreate(&handle));
```

Structure

When writing CUDA C, it is convention to put the CUDA code into `.cu` files and the CPU code in `.c` files. Even when using a library like cuSolver, it is still convention to place that code into `.cu` files as we still need to access some CUDA library functions, such as `cudaMalloc` or `cudaMemCpy`.

The CUDA code associated with matrix parallelisation has been written in the file `$PYTHON/source/matrix_gpu.cu` with the header file `$PYTHON/source/matrix_gpu.h` which includes the function prototypes for the GPU matrix code. The GSL matrix code is kept in `$PYTHON/source/matrix_cpu.c` with function prototypes in `$PYTHON/source/templates.h`.

To be able to switch between the CUDA and GSL matrix implementations with the minimal amount of code changes, a `solve_matrix` wrapper function has been created. Either GSL or cuSolver is called within this wrapper, depending on if Python was compiled with the flag `-DCUDA_ON` as discussed earlier. This wrapper takes on the same name as the original GSL implementation, meaning no code changes have occurred in that regard.

Listing 3: The wrapper function which calls the appropriate matrix solver

```
#include "matrix_gpu.h"  /* The function prototype for gpu_solve_matrix is in here */

int
solve_matrix(double *a_matrix, double *b_vector, int matrix_size, double *x_vector)
{
    int error;

#ifdef CUDA_ON
    error = gpu_solve_matrix(...);  /* CUDA implementation */
#else
    error = cpu_solve_matrix(...);  /* GSL implementation */
#endif

    return error;
}
```

The following code exert is an example of using the wrapper function to solve a linear system.

Listing 4: The API to solve a linear system hasn't changed

```
#include "python.h"

double *populations = malloc(nions * sizeof(*populations));
double *ion_density = malloc(nions * sizeof(*populations));
double *rate_matrix = malloc(nions * nions * sizeof(*populations));

populate_matrices(rate_matrix, ion_density);

/* The wrapper function is named the same as the original GSL implementation
```

(continues on next page)

(continued from previous page)

```
   and accepts the same arguments */
int error = solve_matrix(
    rate_matrix, ion_density, nions, populations, xplasma->nplasma
);

/* One user difference is that error handling is more robust now, and there
   is a function to convert error codes into error messages */
if (error != EXIT_SUCCESS) {
    Error(
        "Error whilst solving for ion populations: %d (%d)\n",
        get_matrix_error_string(error), error
    );
}
```

Here is an example of using a similar wrapper function to calculate the inverse of a matrix.

Listing 5: The API has changed slightly for calculating the inverse, now that it has a wrapper function

```
#include "python.h"

double Q_matrix = malloc(matrix_size * matrix_size * sizeof(double));
double Q_inverse = malloc(matrix_size * matrix_size * sizeof(double));

populate_Q_matrix(Q_matrix);

/* The API is only different in the sense that a wrapper function now
   exists for matrix inversion */
int error = invert_matrix(
    Q_matrix, Q_inverse, matrix_size
);

if (error != EXIT_SUCCESS) {
    Error(
        "Error whilst solving for ion populations: %d (%d)\n",
        get_matrix_error_string(error), error
    );
}
```

To write the cuSolver implementation is similar to the GSL implementation, in that memory/resource are allocated for cuSolver and then the appropriate library functions are called. The code exert below shows an illustrated (and simplified) example of the cuSolver implementation to solve a linear system.

Listing 6: An illustrative example of using cuSolver to solve a linear system using LU decomposition

```
#include <cuSolverDn.h>

extern "C" int  /* extern "C" has to be used to make it available to the C run time */
gpu_solve_matrix(double *a_matrix, double *b_vector, int matrix_size, double *x_vector)
{
    /* First of all, allocate memory on the GPU and copy data from the CPU to the
       GPU. This uses the CUDA standard library functions, such as cudaMemCpy and
```

(continues on next page)

(continued from previous page)

```
       cudaMalloc. This is part of the code is what takes the most time. */
    allocate_memory_for_gpu();
    copy_data_to_gpu();

    /* cuSolver and cuBLAS are both ports of Fortran libraries, which expect arrays to
    be in column-major format and we therefore need to transpose our row-major arrays */
    transpose_row_to_column_major<<<grid_dim, block_dim>>>(
        d_matrix_row, d_matrix_col, matrix_size
    );

    /* Perform LU decomposition. Variables prefixed with d_ are kept in GPU memory where
↪we
    allocated space for them in `allocate_memory_for_gpu` */
    CUSOLVER_CHECK(cusolverDnDgetrf(
        CUSOLVER_HANDLE, matrix_size, matrix_size, d_matrix_col, matrix_size,
        d_workspace, d_pivot, d_info
    ));

    /* Solve the linear system A x = b. The final solution is returned in the
       variable d_v_vector */
    CUSOLVER_CHECK(cusolverDnDgetrs(
        CUSOLVER_HANDLE, CUSOLVER_OP_N, matrix_size, matrix_size, d_matrix_col,
        matrix_size, d_pivot,
        d_b_vector, matrix_size, d_info
    ));

    /* We now have to copy d_b_vector back to the CPU, so we can use that value in
    the rest of Python */
    copy_data_to_cpu();

    return EXIT_SUCCESS;
}
```

The naming conventions of cuSolver are discussed here. In the case above, `cuSolverDnDgetrf` corresponds to: cuSolverDn = *cuSolver Dense Matrix*, D = *double precision (double)* and getrf = *get right hand factorisation*.

The most important thing to note, which may appear trivial, is the `extern` keyword. Without this, when the program is compiled the function `gpu_solve_matrix` will not be available to the C runtime. By labelling the function as `extern "C"`, we make it clear that we want this function to be available to C source code. This only needs to be done at the function definition, and not the function prototype in, e.g., a header file.

Compiling and Linking

CUDA code is compiled using the NVIDIA CUDA Compiler `nvcc`. To combine both CPU and GPU code, the source must be compiled with the respective compilers (e.g. `gcc`/`mpicc` for C and `nvcc` for CUDA) to object code (`.o` files) and which are linked together using the C compiler with the appropriate library flags. In addition to needing to link the cuSolver library (`-lcusolver`) we also need to link the CUDA runtime library (`-lcudart`) when linking with the C compiler, which makes the standard CUDA library functions available to the C compiler and runtime.

The steps for compiling and link GPU and CPU code are outlined below in pseudo-Makefile code.

Listing 7: A brief overview on how to compile and link C and CUDA code

```
# Define compilers for C and CUDA. When creating a CUDA/MPI application, we can
# just as easily use mpicc for our C compiler. It makes no difference.
CC = mpicc
NVCC = nvcc

# Define C and CUDA libraries. We still include GSL as other GSL numerical routines are
# used in Python
C_LIBS = -lgsl -lgslcblas -lm
CUDA_LIBS = -lcudart -lcusolver

# Define flags for C and CUDA compilers. -DCUDA_ON is used to conditionally compile
# to use the CUDA wrappers and other things related to the CUDA build
C_FLAGS = -O3 -DCUDA_ON -DMPI_ON -I../includes -L../libs
CUDA_FLAGS = -O3 -DCUDA_ON

# Compile CUDA source to object code using the CUDA compiler
$(NVCC) $(CUDA_FLAGS) $(CUDA_SOURCE) -c -o $(CUDA_OBJECTS)

# Compile the C code using the C compiler
$(CC) $(C_FLAGS) $(C_SOURCE) -c -o $(C_OBJECTS)

# Link the CUDA and C object code and libraries together using the C compiler
$(CC) $(CUDA_OBJECTS) $(C_OBJECTS) -o python $(CUDA_LIBS) $(C_LIBS)
```

These steps are effectively replicated in the Makefile `$PYTHON/source/Makefile`, where a deconstructed example is shown below.

Listing 8: The variables and recipes associated with CUDA are all conditional on NVCC being defined

```
# If NVCC has been set in the Makefile, then we can define CUDA_FLAG = -DCUDA_ON,
# and the CUDA sources, which, at the moment, uses a wildcard to find all .cu files
ifneq ($(NVCC), )
    CUDA_FLAG = -DCUDA_ON
    CUDA_SOURCE = $(wildcard *.cu)
else
    CUDA_FLAG =
    CUDA_SOURCE =
endif

# Then the recipe to create CUDA object code looks like this. If NVCC is blank,
# nothing happens in the recipe
$(CUDA_OBJECTS): $(CUDA_SOURCE)
ifneq ($(CUDA_FLAG),)
    $(NVCC) $(NVCC_FLAGS) -DCUDA_ON -I$(INCLUDE) -c $< -o $@
endif

# So to compile Python, we have something which looks vaguely like this. Note that
# we use the CUDA_OBJECTS recipe as a requirement for the python recipe. This CUSOLVER_
↪STATUS_SUCCESS
```

(continues on next page)

(continued from previous page)

```
# the CUDA source to be compiled to object code *if* NVCC is defined
python: startup python.o $(python_objects) $(CUDA_OBJECTS)
    $(CC) $(CFLAGS) python.o $(python_objects) $(CUDA_OBJECTS) $(kpar_objects)
↪$(LDFLAGS) -o python
```

3.15.3 Unit Test Framework

Unit tests are an excellent way to ensure that any code you write is robust and correct. To manage unit testing, Python uses the CUnit test framework. Unit tests are run by using `make check` in either the root directory of Python, or in the source/test directory.

Installing CUnit

Python has been tested to work with CUnit (and CUnity) versions newer than 2.1-3. A recent version of CUnit is provided in the `$PYTHON/software` directory and can be installed as a static library by using the Makefile in Python's root directory. To build CUnit from source, you will need CMake installed, which is a modern build system for C and C++ projects.

CUnit will be installed (as a static library) at the same time as GSL and Python during the first-time install, e.g.,

```
$ [$PYTHON] ./configure
$ [$PYTHON] make install
```

It is also possible to install only CUnit, using the same Makefile, if Python and GSL are already installed on your system,

```
$ [$PYTHON] make cunit
```

If compilation of CUnit fails, it's more than likely that you could install a dynamic version of an older version of the library from your system's package manager, e.g.

```
# on macOS using homebrew
$ brew install cunit

# on Debian based Linux distributions
$ sudo apt install libcunit1 libcunit1-doc libcunit1-dev
```

Running Tests

To run the tests, navigate into one of three directories,

- `$PYTHON`
- `$PYTHON/source`
- `$PYTHON/tests`

Then run the command `make check` which will compile and run the unit tests,

```
$ [$PYTHON/source] make check

CUnit - A unit testing framework for C - Version 3.2.7-cunity
```

(continues on next page)

(continued from previous page)

```
http://cunit.sourceforge.net/

Suite: Compton Processes
  Test: Klein-Nisina Formula ...passed
  Test: Compton Alpha - heating cross section  ...passed
  Test: Compton Beta - cooling cross section ...passed
  Test: Compton Formula ...passed
Suite: Matrix Functions: GNU Science Library
  Test: Solve Matrix ...passed
  Test: Invert Matrix ...passed

Run Summary       -      Run    Failed  Inactive   Skipped
     Suites       :        2         0         0         0
     Asserts      :       15         0       n/a       n/a
     Tests        :        6         0         0         0

Elapsed Time: 0.009(s)
```

The important output is the "Run Summary", which lists the number of tests run and how many failed. To explain the output a bit more, a suite is a collection of tests and a test is a function which evaluates the output of the function being tested. If the output is deemed to be correct, the test passes. Otherwise, the function is counted as a failure. The number of failed tests/suites is recorded in the "Failed" column of the table.

The asserts row in the table corresponds to the number of "checks" done, e.g. the number of function outputs checked for correctness. In CUnit terminology, we assert that the output from a function should be something. If the output is that something, then the test is a PASS otherwise it is marked as FAIL.

If a single test in a suite fails, the entire suite is marked as failed. In most cases, there are always more asserts than suites and tests and there are always more, or an equal number of, tests than there are suites.

Writing Tests

Creating a new test

To create a test, we need to make a function which contains an assert statement from the CUnit library. An assert statement is used to fail a test, so that if the condition in the assert statement is not true a failure is reported to the CUnit test registry (more or that later). Test functions should not take any arguments and return an integer, which is typically used to return an exit code which CUnit can use to determine is the test is successful or not it there are no assert statements.

Assert statements come from the `CUnit.h` header, with an exhaustive list of assertions available here. The code example below is a modified exert from the one of matrix unit tests. In the function, test data is retrieved and compared to the output from `solve_matrix` using an assert which compares two floating point arrays to within a tolerance.

It should be noted that this assertion is not part of the standard CUnit assertions. It is possible to make a new assertion by writing a macro (or function) which implements the base `CU_assertImplementation` assert implementation. If you need to create your own assertion, these should be kept in `$PYTHON/source/tests/assert.h`.

Listing 9: `$PYTHON/source/tests/tests/test_matrix.c`

```
#include "assert.h"

#include <CUnit/CUnit.h>
```

(continues on next page)

(continued from previous page)

```
int test_solve_matrix(void) {
  double *matrix_a;
  double *vector_b;
  double *vector_x;

  /* Get input data to `solve_matrix` and `vector_x` which is the "correct"
     answer we will use to compare to the output from `solve_matrix` */

  int vector_size;
  const int get_err =
    get_solve_matrix_test_data(..., &matrix_a, &vector_b, &vector_x, &vector_size);

  if (get_err) {  /* If we can't get the data, fail the test */
    CU_FAIL("Unable to load test data");  /* Assertion from CUnit.h */
  }

  /* Call `solve_matrix` with the input data from above */

  double *test_vector_x = malloc(vector_size * sizeof (double));
  const int matrix_err = solve_matrix(matrix_a, vector_b, vector_size, test_vector_x, -
↪1);

  if (matrix_err) {  /* If there is some numerical error (or otherwise) fail the test */
    CU_FAIL("`solve_matrix` failed with error");
  }

  /* Use the following assertion to compare the value of the "correct" values (vector_x)
     against the output from `solve_matrix` (test_vector_x) */

  CU_CHECK_DOUBLE_ARRAY_EQ_FATAL(test_vector_x, vector_x, vector_size, EPSILON);  /*␣
↪Custom from assert.h */

  free(matrix_a);
  free(vector_b);
  free(vector_x);
  free(test_vector_x);

  return EXIT_SUCCESS;
}
```

Including `python.h` in your tests

If you need to access various structures or other things defined in `python.h`, it is possible to include the header file in your test source code as in the example below (there are some data structures which depend on values defined in `atomic.h`),

```
#include "../../atomic.h"
#include "../../python.h"
```

In some situations this might complicate compilation of the unit test. In those cases, it could be better to re-define

anything you need in the source file for the unit test.

Creating a test suite

Unit tests belong in test suites and not by themselves. This means to create and run a unit test, we need a test suite for that unit test to belong to. A test suite can be thought as a collection of tests, which are usually related. As an example, there is a test suite for testing functions related to the Compton process and a test suite for matrix functions.

The code exert below shows how to create a test suite and to add tests to the suite. The first step is to create a suite to the CUnit test registry (the test registry is a global repository of test suites and associated tests) using `CU_add_suite`, which takes three arguments: the name of the suite, a function (pointer) to run when the suite starts and a function to run after the suite has finished.

When a suite is added to the test registry, a pointer (`CU_pSuite`) to the suite is returned from `CU_add_suite`. This pointer is used to add tests to the suite using `CU_add_test` which takes three arguments: a pointer to the suite to add the test to, the name of the test and the function (pointer) containing the test. `CU_add_test` returns a pointer to the test in the suite. If for whatever reason this fails, `NULL` is returned instead.

Listing 10: `$PYTHON/source/tests/tests/test_matrix.c`

```
void create_matrix_test_suite(void) {
    /* Create a test suite - if suite can't be made, return error code */
    CU_pSuite suite = CU_add_suite(suite_name, matrix_suite_init, matrix_suite_teardown);
    if (suite == NULL) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Add some tests tests to suite - if one of them fails, return error code */
    if (CU_add_test(suite, "Solve Matrix", test_solve_matrix) == NULL) {
        CU_cleanup_registry();
        return CU_get_error();
    }
}
```

The final two arguments for `CU_add_suite` are used to initialise and clean up any additional data structures or resources required to run the tests in the suite. In the matrix suite, for example, the cuSolver runtime is initialized in *matrix_suite_init* and cleaned up in *matrix_suite_teardown*. An example of one of these functions, for the matrix unit tests, is shown in the code exert below. These functions should not take any arguments and return an integer to indicate if everything went OK or not.

Listing 11: `$PYTHON/source/tests/tests/test_matrix.c`

```
int matrix_suite_init(void) {
    int error = EXIT_SUCCESS;

#ifdef CUDA_ON  /* initialise cusolver */
    error = cusolver_create();
#else  /* for GSL, we want to disable the default error handler */
    old_handler = gsl_set_error_handler_off();
#endif

    return error;
```

(continues on next page)

(continued from previous page)

```
}
```

In the examples above, the code to create a suite and add tests is wrapped in a function `create_matrix_test_suite` with no arguments or return. All we need to do now to add those tests is to call that function in the main function of the unit test framework, ensuring we do so after the test registry has been initialized; this is done by the function `CU_initialize_registry`.

Listing 12: `$PYTHON/source/tests/unit_test_main.c`

```
int main(int argc, char **argv) {
    /* Create the test registry */
    if (CU_initialize_registry() != CU_SUCCESS)   {
        return CU_get_error();
    }

    /* Add any test suites to the registry */
    create_matrix_test_suite();

    /* Set how verbose logging should be - CU_BRM_VERBOSE gets you the
       output shown in the running tests section */
    CU_basic_set_mode(CU_BRM_VERBOSE);

    /* Run the test suites */
    CU_basic_run_tests();

    /* Check how many tests failed */
    const int num_tests_failed = CU_get_number_of_tests_failed();

    /* Report on the number of tests failed, or if everything passed */
    if (num_tests_failed > 0) {
        printf("\033[1;31m%d test(s) failed\n\033[1;0m", num_tests_failed);  /* red text␣
↪*/
    } else {
        printf("\033[1;32mAll tests ran successfully\n\033[1;0m");  /* green text */
    }

    /* Clean up the CUnit registry */
    CU_cleanup_registry();

    return num_tests_failed;
}
```

Directory and structure

Unit tests should be kept in logically named files within the unit test directory located at `$PYTHON/source/tests/tests`. Any file in this directory should be added to the unit test Makefile, which is located at `$PYTHON/source/tests/Makefile`, specifically to the `TEST_SOURCES` variable which is a list of all the source code required specifically for the unit test framework; this includes both the unit tests themselves and any other code required to, e.g., build and control the test registry. Prototypes for wrapper functions for creating test suites (which are called in the main function) should be placed in `$PYTHON/source/tests/tests/tests.h` header file. Any data required for the tests should be kept in the data directory, `$PYTHON/source/tests/data`, in appropriately organised directories as shown below.

Listing 13: $PYTHON/source/tests

```
$ tree $PYTHON/source/tests

├── Makefile
├── assert.h
├── data
│   └── matrix
│       ├── inverse_macro
│       │   ├── inverse.txt
│       │   └── matrix.txt
│       └── small_matrix
│           ├── A.txt
│           ├── b.txt
│           └── x.txt
├── tests
│   ├── test_matrix.c
│   └── tests.h
└── unit_test_main.c
```

We also need to include the Python source code we are testing in the PYTHON_SOURCES variable of the Makefile. If there are any CUDA files required, these should be added to the CUDA_SOURCES variable. In theory, we should only need to include the files containing the code we are testing. But in practise, we choose to instead include all of Python's source files (as it makes our lives easier) which increases compile time and the size of the final binary.

3.16 Python Scripts

There are several *Python* (the scripting language) scripts written to prepare input for and analyse the output of *python* (the C code).

Some of the more useful scripts/modules are documented below. Alternatively, you can generate documentation for all the scripts by navigating to `docs/pydocs` and running `write_docs.py`. The resulting output file can be found here.

Warning to user

The scripts documented here form an incomplete and inhomogenous list, in the sense that they have been developed by different people at different times and do not fit nicely together as a single python package. Some of the scripts should still be useful, particularly if you consult example notebookes, but use with caution!

Todo: Finish adding modules below.

3.16.1 Plotting

py_plot_output

Synopsis:
various plotting routines for making standard plots from Python outputs

Usage:
Either import as a module in a python session e.g. import py_plot_output as p

or run from the command line e.g.

py_plot_output root mode [roots to compare]

Arguments:

root
root filename to analyse

mode
mode of plotting wind plot of common wind quantites ions plot of common ions spec spectrum for different viewing angles spec_comps components contributing to total spectrum e.g. disk, wind compare compare the root to other roots to compare all make all the above plots

py_plot_output.**make_spec_comparison_plot**(*s_array*, *labels*, *fname='comparison'*, *smooth_factor=10*, *angles=True*, *components=False*)

make a spectrum comparison plot from array of astropy.table.table.Table objects. Saves output as "spectrum_%s.png" % (fname)

Parameters:

s_array: array-like of astropy.table.table.Table objects
table containing spectrum data outputted from Python

labels: array-like
strings of labels for each spectrum

fname: str
filename to save as e.g. sv

smooth_factor: int
factor you would like to smooth by, default 10

angles: Bool
Would you like to plot the viewing angle spectra?

components: Bool
would you like to plot the individual components e.g. Disk Wind

Returns:
Success returns 0 Failure returns 1

py_plot_output.**make_spec_plot**(*s*, *fname*, *smooth_factor=10*, *angles=True*, *components=False*, *with_composite=False*)

make a spectrum plot from astropy.table.table.Table object. Saves output as "spectrum_%s.png" % (fname)

Parameters:

s: astropy.table.table.Table
table containing spectrum data outputted from Python

fname: str
filename to save as e.g. sv

smooth_factor: int
factor you would like to smooth by, default 10

angles: Bool
Would you like to plot the viewing angle spectra?

components: Bool
would you like to plot the individual components e.g. Disk Wind

Returns:
Success returns 0 Failure returns 1

py_plot_output.**make_spec_plot_from_class**(*s*, *fname*, *smooth_factor=10*, *angles=True*, *components=False*)

make a spectrum plot from py_classes.specclass object. Saves output as "spectrum_%s.png" % (fname)

Parameters:

s: specclass object
table containing spectrum data outputted from Python

fname: str
filename to save as e.g. sv

smooth_factor: int
factor you would like to smooth by, default 10

angles: Bool
Would you like to plot the viewing angle spectra?

components: Bool
would you like to plot the individual components e.g. Disk Wind

Returns:
Success returns 0 Failure returns 1

py_plot_output.**make_wind_plot**(*d*, *fname*, *var=None*, *shape=(4, 2)*, *axes='log'*, *den_or_frac=0*, *fname_prefix='wind'*, *lims=None*)

make a wind plot from astropy.table.table.Table object. Saves output as "spectrum_%s.png" % (fname)

Parameters:

d: astropy.table.table.Table
table containing wind data outputted from Python if == None then this routine will get the data for you

fname: str
filename to save as e.g. sv

var: array type
array of string colnames to plot

angles: Bool
Would you like to plot the viewing angle spectra?

components: Bool
would you like to plot the individual components e.g. Disk Wind

axes: str
lin or log axes

den_or_frac: int
0 calculate ion densities 1 calculate ion fractions

lims: array-like
limits of plot, specified as ((xmin,xmax), (ymin, tmax)) can be array or tuple. Default is Nonetype.

Returns:
Success returns 0 Failure returns 1

plot_wind

Synopsis:
These are routines for plotting various parameters in of the wind after these parameters have been saved to an astropy-compatible ascii table

Command line usage
plot_wind filename var

to make a plot of a single variable from the command line

Description:

Primary routines:

doit
[Create a plot of a single variable in a file made with] windsave2table. This is the routine called from the command line. Additional options are available when called from a python script.

compare_separate
[Compare a single variable in two] different runs of python and produce 3 separate plots one for each run and one containing the difference

compare: Similar to compare_separate but produces a
single file

`plot_wind.compare(`*f1='sv_master.txt', f2='sv_master.txt', var='t_r', grid='ij', inwind='', scale='guess', zmin=-1e+50, zmax=1e+50, fig_no=5*`)`

Compare results of two variables within a single plot

The three plots are of from the first file, the second file, and the difference respectively

`plot_wind.compare_separate(`*f1='fiducial_agn_master.txt', f2='fiducial_agn_master.txt', var='t_r', grid='ij', inwind='', scale='guess', zmin=-1e+50, zmax=1e+50, fig_no=5*`)`

This routine compares the same variable from two different runs of python, and produces Three separate plots. The plots represent the variable in the first file, the variable in the second file and the difference between the two

`plot_wind.doit(`*filename='fiducial_agn.master.txt', var='t_r', grid='ij', inwind='', scale='guess', zmin=-1e+50, zmax=1e+50, plot_dir='', root=''*`)`

Plot a single variable from an astropy table (normally created with windsave2table, with various options

where var is the variable to plot where grid can be ij, log, or anything else. If ij then the plot will be in grid coordinates, if log the plot will be in on a log scale in physical coordiantes. If anything else, the plot will be on a linear scale in physical coordiantes where scale indicates how the variable should be plotted. guess tells the routine to make a sensible choice linear implies the scale should be linear and log implies a log scale should be used where zmin and zmax overide the max and mimimum in the array (assuming these limits are with the range of the variable)

`plot_wind.get_data(`*filename='fiducial_agn_master.txt', var='t_r', grid='ij', inwind='', scale='guess', zmin=-1e+50, zmax=1e+50*`)`

This routine reads and scales the data from a single variable that is read from an ascii table representation of one or more of the parameters in a windsave file

`plot_wind.just_plot(`*x, y, xvar, root, title, xlabel, ylabel, fig_no=1, vmin=0, vmax=0*`)`

This routine simply is produces a plot of a variable from that has been printed to an astropy table with a routine like windsave2table. This function is simply a plotting routine

3.16.2 Checking Runs and Testing

run_check

Synopsis:
Sumarize a model run with python, ultimately generating an html file with various plots, etc.

Command line usage (if any):
run_check.py root1 [root2 . . .]

run_check.py root1.pf [root2.pf . . .]

run_check -all

run_check -h

Description:
This routine performs basic checks on one or more python runs and creates an html file for each that is intended to provide a quick summary of a run.

The user can enter the runs to be tested from the command line, either in the form of a set of root names or .pf names. Wildcarding, e.g `*.pf` can be used. `*.out.pf` files will be ignored.

Alternatively to process all of the files in a directory, one can use the switch -all (which supercedes anything else).

In all cases the routine checks to see if the appropriate wind_save file exists before attempting to run.

-h delivers this documentation

Primary routines:
doit - processes a single file steer - processes the command line and calls doit for each file.

Notes:

run_check.**check_completion**(*root*)

Verify that the run actually completed and provide information about the timeing, from the .sig file

run_check.**doit**(*root='ixvel'*, *outputfile='out.txt'*)

Create a summary of a Python run, which will enough information that one can assess whether the run was successful

Description:

Notes:

History:

run_check.**how_many_dimensions**(*filename*)

Check whether a windsave file is one or two dimenaions

run_check.**make_html**(*root*, *converge_plot*, *te_plot*, *tr_plot*, *spec_tot_plot*, *spec_plot*, *nspectra=3*, *complete_message=['test']*, *errors=['test', 'test2']*)

Make an html file that collates all the results

run_check.**plot_converged**(*root*, *converged*, *converging*, *t_r*, *t_e*, *hc*)

Make a plot of the convergence statistics in the diag directroy

run_check.**py_error**(*root*)

Run py_error.py and capture the output to the screen

Note:
py_error could be refactored so that it did not need to be run from the command line, but this is the simplest way to capture the outputs at present

run_check.**read_diag**(*root*)

Get convergence and possibly other information from the diag file

run_check.**steer**(*argv*)

Process the command line

run_check.**windsave2table**(*root*)

Run windsave2table

Normally this will just run windsave2table, but if it turns out that that fails the routine will try to run the same version (not commit) of windsave2table that python was run with. This will only work, if the correct version exists in one's bin file

run_check.**xwindsave2table**(*root*)

Run windsave2table with the same version number (not commit) as the .spec files indicate was written)

This is a backup method, and is not guaranteed to work. Two obvious reasons it could fail would be if one does not have the specified compiled version of windsave2bable in one's path, or if the the structure of the windsavefile changed mid-version.

3.16.3 Utility, I/O and Imports

py_read_output

Synopsis:
This program enables one to read outputs from the Python radiative transfer code. Where possible, we use the astropy.io module to read outputs. There are also a number of routines for processing and reshaping various data formats

see https://github.com/agnwinds/python/wiki/Useful-python-commands-for-reading-and-processing-outputs for usage

Usage:

Arguments:

py_read_output.**read_convergence**(*root*)

check convergence in a diag file

py_read_output.**read_emissivity**(*root*)

Read macro atom emissivities from a root diag file. Returns two arrays, kpkt_emiss and matom_emiss.

py_read_output.**read_pf**(*root*)

reads a Python .pf file and returns a dictionary

Parameters

root
[file or str] File, filename to read.

new:
True means the Created column exists in the file

Returns

pf_dict
Dictionary object containing parameters in pf file

py_read_output.**read_pywind**(*filename*, *return_inwind=False*, *mode='2d'*, *complete=True*)

read a py_wind output file using np array reshaping and manipulation

Parameters

filename
[file or str] File, filename to read, e.g. root.ne.dat

return_inwind: Bool
return the array which tells you whether you are partly, fully or not inwind.

mode: string
can be used to control different coord systems

Returns

x, z, value: masked arrays
value is the quantity you are concerned with, e.g. ne

py_read_output.**read_pywind_summary**(*filename*, *return_inwind=False*, *mode='2d'*)

read a py_wind output file using np array reshaping and manipulation

Parameters

filename
[file or str] File, filename to read, e.g. root.ne.dat

return_inwind: Bool
return the array which tells you whether you are partly, fully or not inwind.

mode: string
can be used to control different coord systems

Returns

d: astropy.Table.table.table object
value is the quantity you are concerned with, e.g. ne

py_read_output.**read_spectrum**(*filename*)

Load data from a spectrum output file from the radiative transfer code Python

Parameters:
filename : file or str

File, filename, or generator to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed. Note that generators should return byte strings for Python 3k.

Returns
Success: spectrum returns a Table of class astropy.table.table.Table

Failure returns 1

py_read_output.**read_spectrum_to_class**(*filename*, *new=True*)

reads a Python .spec file and places in specclass array, which is returned

Parameters

filename
[file or str] File, filename to read.

new:
True means the Created column exists in the file

Returns

Success: spectrum returns a spectrum class cls.specclass

Failure returns 1

py_read_output.**setpars**()

set some standard parameters for plotting

py_read_output.**thinshell_read**(*root*)

Read py_wind output filename for thin shell models with one cell

py_read_output.**write_pf**(*root*, *pf_dict*)

writes a Python .pf file from a dictionary

Parameters

root
[file or str] File, filename to write.

pf_dict:
dictionary to write

Returns

pf_dict
Dictionary object containing parameters in pf file

py_plot_util

Synopsis:

various utilities for processing Python outputs and plotting spectra and wind properties

Usage:

Arguments:

py_plot_util.**get_flux_at_wavelength**(*lambda_array*, *flux_array*, *w*)

Find the flux at wavelength w

Parameters:

lambda_array: array-like
array of wavelengths in angstroms. 1d

flux_array: array-like
array of fluxes same shape as lambda_array

w: float
wavelength in angstroms to find

Returns:

f: float
flux at point w

py_plot_util.**get_pywind_summary**(*fname*, *vers=''*, *den_or_frac=0*)

run version vers of py_wind on file fname.wind_save and generate the complete wind summary as output

produce the output fname.complete to read

if den_or_frac is 1, return fractions, otherwise densities

py_plot_util.**parse_rcparams**(*fname='params.rc'*)

parse the file params.rc and set values in matplotlib.rcparams

file should be of format

font.family : serif mathtext.fontset : custom

py_plot_util.**read_pywind_smart**(*filename*, *return_inwind=False*)

read a py_wind file using np array reshaping and manipulation

DEPRECATED

py_plot_util.**run_py_wind**(*fname*, *vers=''*, *cmds=None*, *ilv=None*, *py_wind_cmd='py_wind'*, *return_output=False*)

run version vers of py_wind on file fname.wind_save

py_plot_util.**smooth**(*x*, *window_len=20*, *window='hanning'*)

smooth data x by a factor with window of length window_len

py_plot_util.**wind_to_masked**(*d*, *value_string*, *return_inwind=False*, *mode='2d'*, *ignore_partial=True*)

turn a table, one of whose colnames is value_string, into a masked array based on values of inwind

Parameters:

d: astropy.table.table.Table object
data, probably read from .complete wind data

value_string: str
the variable you want in the array, e.g. "ne"

return_inwind: Bool
return the array which tells you whether you are partly, fully or not inwind.

Returns:

x, z, value: Floats
value is the quantity you are concerned with, e.g. ne

import_cyl

Synopsis:

Read the master file produced by windsave2table for a model created in cylindrical coordinates and produce a file which can be imported into Python and run

Command line usage (if any):

`import_cyl.py rootname` where rootname is the rootname of the mastertable or windsave file

Description:

This operates on the mastertable produced by windsavetable

Primary routines:

doit

Notes:

import_cyl.**doit**(*root='cv'*, *outputfile=''*)

Read a master.txt file for models in cylindrical coordinates and produce a file which can be read in to python

Description:

Notes:

History:

import_cyl.**read_file**(*filename*, *char=''*)

Read a file and split it into words, eliminating comments

char is an optional parameter used as the delimiter for splitting lines into words. Otherwise white space is assumed.

import_cyl.**read_table**(*filename='foo.txt'*, *format=''*)

Read a file using astropy.io.ascii and return this

Description:

Notes:

History:

3.16.4 py4py

py4py is a module written in *Python* for reading, processing and visualising the input and output files of the *c* code **Python**.

Installation instructions can be found in the associated README.md

A reverberation mapping example using a Jupyter Notebook can be found under *Reverberation Mapping*

py4py

Functions designed for plotting output files

py4py.py4py.**load_grid**(*filename*)

Loads a pair of grid files from a root name.

Use as:
x_r10, z_r10 = load_grid('r10/')

py4py.py4py.**plot_dat**(*table*, *grid_x*, *grid_z*, *title*, *label*, *volume=True*)

Plots a given .dat file

Use as:
plot_dat(table_h1_r01, x_r01, z_r01, 'H-I, radius 1x', 'Log ion fraction', volume=False)

py4py.py4py.**plot_dat_many**(*tables*, *grids_x*, *grids_z*, *xlims*, *zlims*, *titles*, *title*, *label*, *shared_y=False*, *shared_cbar=False*, *volume=True*, *log=True*)

Plot many dat files on a single plot.

Use:

plot_dat_many([table_h1_r01, table_h1_r10, table_h1_r30],
[x_r01, x_r10, x_r30], [z_r01, z_r10, z_r30], xlims=[(14.5, 17.5), (15.5, 17.5), (16, 17.5)], zlims=[(13, 17), (13, 17), (13, 17)], titles=['1x Radius', '10x Radius', '30x Radius'], title='H-I ion fraction', label='Log ion fraction', shared_y=True, volume=False, shared_cbar=True)

py4py.py4py.**plot_spec**(*col*, *spectra*, *names*, *log_x=False*, *log_y=False*, *scale_to=None*, *lim_x=False*)

Plots an array of python spectra imported as Tables.

Use as:
plot_spec('A40P0.50', [spec_r01, spec_r10], ['1x', '10x'])

py4py.reverb

Reverberation Mapping module

This contains the type used to create and manipulate reverberation maps from Python output files.

Example:

For an existing delay output file called 'qso.delay_dump', to generate a TF plot for the C4 line for a specific spectrum, with axis of velocity offset vs days, you would do:

```
qso_conn = open_database('qso')
tf_c4_1 = TransferFunction(
    qso_conn, continuum=1e43, wave_bins=100, delay_bins=100, filename='qso_c4_
↪spectrum_1'
)
tf_c4_1.spectrum(1).line(443).run()
tf_c4_1.plot(velocity=True, days=True)
```

Given database queries can take a long time, it is advisable to pickle a TF that has been run so you can access it later on. Note, however: Once a TF has been restored from a pickle, you can no longer change the filters and re-run.

with open('qso_c4_spectrum_1', 'wb') as file:
pickle.dump(tf_c4_1, file)

class py4py.reverb.**Origin**(***kwargs*)

The SQLalchemy table for the photon origins. Unused. Could be removed but will break backward compatibility. Information required for this is not stored in the output files.

Todo: Implement or remove this table

class py4py.reverb.**Photon**(***kwargs*)

SQLalchemy class for a photon. Why are all the properties capitalised? Changing them to lowercase as would make sense breaks backwards compatibility.

Todo: Change to lower case.

class py4py.reverb.**Spectrum**(***kwargs*)

The SQLalchemy table for the spectra. Unused. Could be removed but will break backward compatibility. Information required for this is not stored in the output files.

Todo: Implement or remove this table.

class py4py.reverb.**TransferFunction**(*database: Connection*, *filename: str*, *continuum: float*, *wave_bins: int = None*, *delay_bins: int = None*, *template:* TransferFunction *= None*, *template_different_line: bool = False*, *template_different_spectrum: bool = False*)

Used to create, store and query emissivity and response functions

cont_scatters(*scat_min: int*, *scat_max: Optional[int] = None*) → *TransferFunction*

Constrain the TF to only photons that have scattered min-max times via a continuum scattering process (e.g. electron scattering).

Args:
scat_min (int): Minimum number of continuum scatters scat_max (Optional[int]): Maximum number of continuum scatters, if desired

Returns:
TransferFunction: Self, so filters can be stacked

count(*delay: Optional[float] = None*, *wave: Optional[float] = None*, *delay_index: Optional[int] = None*) → Union[int, ndarray]

Returns the photon count in either one specific wavelength/delay bin, or all wavelength bins for a given delay.

Args:

delay (Optional[float]): Delay to return value for. Must provide this or delay_index. delay_index (Optional[int]): Delay index to return value for. Must provide this or delay. wave (Optional[float]): Wavelength to return value for

Returns:

Union[int, np.ndarray]: Either the count in one specific bin, or if wave is not specified
the counts in each wavelength bin at this delay

Todo:

Allow for only wavelength to be provided?

delay(*response: bool = False*, *threshold: float = 0*, *bounds: float = None*, *days: bool = False*)

Calculates the centroid delay for the current data

Args:

response (bool):
Whether or not to calculate the delay from the response

threshold (float):
Exclude all bins with value < threshold

bounds (float):
Return the fractional bounds (i.e. bounds=0.25, the function will return [0.5, 0.25, 0.75]). Not implemented.

days (bool):
Whether to return the delay in days or seconds

Returns:

Union[float, Tuple[float, float, float]]:
Centroid delay, and lower and upper fractional bounds if bounds keyword provided

Todo:

Implement fractional bounds. Should just be able to call the centroid_delay function!

delay_bins() → ndarray

Returns the range of delays covered by this TF.

Returns:

np.ndarray: Array of the bin boundaries.

delay_dynamic_range(*delay_dynamic_range: float*) → *TransferFunction*

If set, the TF will generate delay bins to cover this dynamic range of responses, i.e. (1 - 10^-ddr) of the delays. So a ddr of 1 will generate photons with delays up to 1 - (1/10) = the 90th percentile of delays. ddr=2 will give up to the 99th percentile, 3=99.9th percentile, etc.

Arguably this is a bit of an ambiguous name

Args:

delay_dynamic_range (float): The dynamic range to be used when

Returns:

TransferFunction: Self, so filters can be stacked

delay_peak(*response: bool = False*, *days: bool = False*) → float

Calculates the peak delay for the transfer or response function, i.e. the delay at which the response is strongest.

Args:
response (bool): Whether or not to calculate the peak transfer or response function. days (bool): Whether to return the value in seconds or days.

Returns:
float: The peak delay.

delays(*delay_min: float*, *delay_max: float*, *days: bool = True*) → *TransferFunction*

The delay range that should be considered when producing the TF.

Args:
delay_min (float): Minimum delay time (in seconds or days) delay_max (float): Maximum delay time (in seconds or days) days (bool): Whether or not the delay range has been provided in days

Returns:
TransferFunction: Self, so filters can be stacked

emissivity(*delay: Optional[float] = None*, *wave: Optional[float] = None*, *delay_index: Optional[int] = None*) → Union[float, ndarray]

Returns the emissivity in either one specific wavelength/delay bin, or all wavelength bins for a given delay.

Args:
delay (Optional[float]): Delay to return value for. Must provide this or delay_index. delay_index (Optional[int]): Delay index to return value for. Must provide this or delay. wave (Optional[float]): Wavelength to return value for.

Returns:

Union[int, np.ndarray]: Either the emissivity in one specific bin, or if wave is not specified
the counts in each wavelengthin bin at this delay

Todo:
Allow for only wavelength to be provided?

filter(**args*)

Apply a SQLalchemy filter directly to the content.

Args:
args: The list of filter arguments

Returns:
TransferFunction: Self, so filters can be stacked

fwhm(*response: bool = False*, *velocity: bool = True*)

Calculates the full width half maximum of the delay-summed transfer function, roughly analogous to the line profile. Possibly meaningless for the response function?

Args:
response (bool): Whether to calculate the FWHM of the transfer or response function velocity (bool): Whether to return the FWHM in wavelength or velocity-space

Returns:

float: Full width at half maximum for the function.
If the function is a doublet, this will not work properly.

Todo:
Catch doublets.

line(*number: int*, *wavelength: float*) → *TransferFunction*

Constrain the TF to only photons last interacting with a given line

This includes being emitted in the specified line, or scattered off it

Args:
number (int): Python line number. Will vary based on data file! wavelength (float): Wavelength of the line in angstroms

Returns:
TransferFunction: Self, so filters can be stacked

lines(*line_list: List[int]*) → *TransferFunction*

Constrain the TF to only photons with a specific internal line number. This list number will be specific to the python atomic data file!

Args:
line_list (List[int]): List of lines

Returns:
TransferFunction: Self, so filters can be stacked

plot(*log: bool = False*, *normalised: bool = False*, *rescaled: bool = False*, *velocity: bool = False*, *name: str = None*, *days: bool = True*, *response_map=False*, *keplerian: dict = None*, *dynamic_range: int = None*, *rms: bool = False*, *show: bool = False*, *max_delay: Optional[float] = None*, *format: str = '.eps'*) → *TransferFunction*

Takes the data gathered by calling 'run' and outputs a plot

Args:

log (bool):
Whether the plot should be linear or logarithmic.

normalised (bool):
Whether or not to rescale the plot such that the total emissivity = 1.

rescaled (bool):
Whether or not to rescale the plot such that the maximum emissivity = 1.

velocity (bool):
Whether the plot X-axis should be velocity (true) or wavelength (false).

name (Optional[str]):
The file will be output to 'tf_filename.eps'. May add the 'name' component to modify it to 'tf_filename_name.eps'. Useful for adding e.g. 'c4' or 'log'.

days (bool):
Whether the plot Y-axis should be in days (true) or seconds (false).

response_map (bool):
Whether to plot the transfer function map or the response function.

keplerian (Optional[dict]):
A dictionary describing the profile of a keplerian disk, the bounds of which will be overlaid on the plot. Arguments include angle (float) - Angle of disk to the observer, mass (float) - Mass of the central object in M_sol, radius (Tuple(float, float)) - Inner and outer disk radii, include_minimum_velocity - Whether or not to include the outer disk velocity profile (default no).

dynamic_range (Optional[int]):
If the plot is logarithmic, the dynamic range the colour bar should show. If not provided, will attempt to use the base dynamic range property, otherwise will default to showing 99.9% of all emissivity.

max_delay (Optional[float]):
The optional maximum delay to plot out to.

rms (bool):
Whether or not the line profile panel should show the root mean squared line profile.

show (bool):
Whether or not to display the plot to screen.

format (str):
The output file format. .eps by default.

Returns:
TransferFunction: Self, for chaining outputs

res_scatters(*scat_min: int*, *scat_max: Optional[int] = None*) → *TransferFunction*

Constrain the TF to only photons that have scattered min-max times via a resonant scattering process (e.g. line scattering).

Args:
scat_min (int): Minimum number of resonant scatters scat_max (Optional[int]): Maximum number of resonant scatters, if desired

Returns:
TransferFunction: Self, so filters can be stacked

response(*delay: Optional[float] = None*, *wave: Optional[float] = None*, *delay_index: Optional[int] = None*) → Union[float, ndarray]

Returns the responsivity in either one specific wavelength/delay bin, or all wavelength bins for a given delay.

Args:
delay (Optional[float]): Delay to return value for. Must provide this or delay_index. delay_index (Optional[int]): Delay index to return value for. Must provide this or delay. wave (Optional[float]): Wavelength to return value for.

Returns:

Union[int, np.ndarray]: Either the responsivity in one specific bin, or if wave is not specified
the counts in each wavelength bin at this delay

Todo:
Allow for only wavelength to be provided?

response_map_by_tf(*low_state:* TransferFunction, *high_state:* TransferFunction, *cf_low: float = 1.0*, *cf_high: float = 1.0*) → *TransferFunction*

Creates a response function for this transfer function by subtracting two transfer functions bracketing it. Requires two other completed transfer functions, bracketing this one in luminosity, all with matching wavelength/velocity and delay bins.

Correction factors are there to account for things like runs that have been terminated early, e.g. if you request 100 spectrum cycles and stop (or Python dies) after 80, the total photon luminosity will only be 80/100. A correction factor allows you to bump this up. Arguably correction factors should be applied during the 'run()' method.

Args:
low_state (TransferFunction): A full, processed transfer function for a lower-luminosity system. high_state (TransferFunction): A full, processed transfer function for a higher-luminosity system. cf_low (float): Correction factor for low state. Multiplier to the whole transfer function. cf_high (float): Correction factor for high state. Multiplier to the whole transfer function.

Returns:
TransferFunction: Self, so plotting can be chained on.

`response_total`() → float

Returns the total response.

Returns:
float: Total response.

`run`(*scaling_factor: float = 1.0*, *limit: int = None*, *verbose: bool = False*) → *TransferFunction*

Performs a query on the photon DB and bins it.

A TF must be run *after* all filters are applied and before any attempts to retrieve or process data from it. This can be a time-consuming call, on the order of 1 minute per GB of input file.

Args:

scaling_factor (float):
1/Number of cycles in the spectra file

limit (int):
Number of photons to limit the TF to, for testing. Recommend testing filters on a small number of photons to begin with.

verbose (bool):
Whether to output exactly what the query is.

Returns:
TransferFunction: Self, for chaining commands

`spectrum`(*number: int*) → *TransferFunction*

Constrain the TF to photons from a specific observer

Args:
number (int): Observer number from Python run

Returns:
TransferFunction: Self, so filters can be stacked

`transfer_function_1d`(*response: bool = False*, *days: bool = True*) → ndarray

Collapses the 2-d transfer/response function into a 1-d response function, and returns the bin midpoints and values in each bin for plotting.

Args:
response (bool): Whether or not to return the response function data days (bool): Whether the bin midpoints should be in seconds or days

Returns:

np.ndarray: A [bins, 2]-d array containing the midpoints of the delay bins,
and the value of the 1-d transfer or response function in each bin.

`velocities`(*velocity: float*) → *TransferFunction*

Constrain the TF to only photons with a range of Doppler shifts

Args:

velocity (float): Maximum doppler shift velocity in m/s. Applies
to both positive and negative Doppler shift

Returns:
TransferFunction: Self, so filters can be stacked

wavelength_bins(*wave_range: ndarray*) → *TransferFunction*

Constrain the TF to only photons with a range of wavelengths, and to a specific set of bins

Args:
wave_range (np.ndarray): Array of bins to use

Returns:
TransferFunction: Self, so filters can be stacked

wavelengths(*wave_min: float*, *wave_max: float*) → *TransferFunction*

Constrain the TF to only photons with a range of wavelengths

Args:
wave_min (float): Minimum wavelength in angstroms wave_max (float): Maximum wavelength in angstroms

Returns:
TransferFunction: Self, so filters can be stacked

py4py.reverb.**calculate_delay**(*angle: float*, *phase: float*, *radius: float*, *days: bool = True*) → float

Delay relative to continuum for emission from a point on the disk.

Calculate delay for emission from a point on a keplerian disk, defined by its radius and disk angle, to an observer at a specified angle.

Draw plane at r_rad_min out. Find x projection of disk position. Calculate distance travelled to that plane from the current disk position Delay relative to continuum is thus (distance from centre to plane) + distance from centre to point

Args:
angle (float): Observer angle to disk normal, in radians phase (float): Rotational angle of point on disk, in radians. 0 = in line to observer radius (float): Radius of the point on the disk, in m days (bool): Whether the timescale should be seconds or days

Returns:
float: Delay relative to continuum

py4py.reverb.**open_database**(*file_root: str*, *user: str = None*, *password: str = None*, *batch_size: int = 25000*)

Open or create a SQL database

Will open a SQLite DB if one already exists, otherwise will create one from file. Note, though, that if the process is interrupted the code cannot intelligently resume- you must delete the half-written DB!

Args:

file_root (string):
Root of the filename (no '.db' or '.delay_dump')

user (string):
Username. Here in case I change to PostgreSQL

password (string):
Password. Here in case I change to PostgreSQL

batch_size (int):
Number of photons to stage before committing. If too low, file creation is slow. If too high, get out-of-memory errors.

Returns:
sqlalchemy.engine.Connection: Connection to the database opened

PYTHON MODULE INDEX

INDEX

R

S

T

V

W

X