Sedona6 User Guide

Jul 16, 2020

Contents:

CHAPTER 1

Installing Sedona

1.1 Getting the Code

The code is available for download on github here. Or you can clone from the command line using:

```
git clone git@github.com:dnkasen/sedona6.git
```

1.2 Installing Dependencies

[sedona] requires a C++ compiler for instalation. In addition. the following dependencies must be installed:

- HDF5 (file formating)
- GSL (gnu scientific library)
- lua (scripting language used for parameter files)

[sedona] is parallelized using a hybrid of MPI and openMP. To run in parallel, you must also have installed an MPI and/or an openMP distribution.

These packages may already been installed on clusters and computing systems. If not, they can be conveniently installed using a package manager – see below for details.

If your dependencies are not installed in a standard location (e.g., usr/local/) then you will need to set the `GSL_DIR`, `HDF5_DIR`, and `LUA_DIR` environment variables to specify the path to the instalations. In the bash shell, for example, add to your bash.profile:

```
export  GSL_DIR=/base/path/of/gsl/
export HDF5_DIR=/base/path/of/hdf5/
export  LUA_DIR=/base/path/of/lua/
```

where the `/base/path/of/xxx` should be replaced with the full base path where the package is installed.

A python distribution is not required to run [sedona] itself, however it can be useful for working with input and output files (in particular the h5py package allows for easy creation and reading of hdf5 files). The [sedona] distribution comes with a python package called *sedonalib* that contains useful tools for generating, reading, and plotting files. The anaconda python distribution makes it relatively easy to install the needed python packages.

Installing Dependencies on a Mac

On a mac, the *homebrew* package manager provides a convenient way to install dependences. Type `brew help` to check if homebrew it is already installed on your machine. If the command is not found, it can be installed by issuing the command given on the homebrew installation page

Once installed, the necessary [sedona] dependencies can be installed with the following commands:

```
brew install lua
brew install gsl
brew install hdf5
```

If you wish to run mpi parallel jobs, you should also install mpich:

```
brew install mpich
```

If you lack a C++ compiler, you can install the gcc compiler through `brew install gcc`. Alternatively, mac Mac provides a C++ compiler through its Xcode tools.

Installing Dependencies on a Linux System

On Debian, Ubunto and some other linux systems, the *apt* package manager provides a convenient way to install dependences. If *apt* is installed on your machine, you can update it using:

```
sudo apt-get update
```

Root privileges are required. You can then install the necessary dependencies using:

```
sudo apt-get install gsl-bin
sudo apt-get -s install lua5.2
sudo apt-get install libhdf5-dev
```

If you wish to run mpi and openMP parallel jobs, you should also install:

```
sudo apt-get install  mpich
sudo apt install libomp-dev
```

Installing Dependencies on a Cluster

Many clusters and supercomputer centers will have some of the dependencies already installed. Use the module load

1.3 Compiling the Code

To compile [sedona], first change to the `src/` directory in the [sedona] distribution, and type:

```
chmod +x ./install.sh
```

to ensure that the compile script is executable. [sedona] comes with a set of makefiles to facilitate compilation on different machines. To see a list of machines that have makefiles, type:

```
./install.sh help
```

If you see a machine name appropriate to your system, you can attempt compiling the code by typing:

```
./install.sh <MACHINE>
```

where `<MACHINE>` is the name of the machine.

If you don't see a relevant Makefile for your machine, or if compilation fails, you should modify a Makefile directly. Open the `makefiles/Makefile.general` file and make sure that the `CXX` variable is set to your C++ compiler. You can also change the `CXX_FLAGS` variables to set the compiler flags of your choosing. You can also check that all dependent packages are installed and that the GSL_DIR, HDF5_DIR and LUA_DIR environment variables are set, as described in *Installing Dependencies*

Once compilation is successful, the executable file `sedona6.ex` will appear in the `src/` directory. Copy this to the directory where you would like to run the code.

CHAPTER 2

Running Sedona

Before running [sedona] it is helpful to set the environment variable `SEDONA_HOME` to point to the base directory of sedona. In the bash shell, for example, add this line to your `.bash_profile` or `.bashrc`:

```
export SEDONA_HOME=/Users/kasen/sedona6/
```

replacing `/Users/kasen/sedona6/` with the base directory of your sedona

To run [sedona] on a single processor, copy the compiled `sedona6.ex` executable into the directory where you plan to run. Make sure the required input files exist (see . . .) and run the code with the command:

```
./sedona6.ex my_parameter_file.lua
```

where `my_parameter_file.lua` is the name of the input parameter file containing runtime parameters. If you run `./sedona6.ex` with no subsequent argument, the code will assume the parameter file is name *param.lua*

For calculations on multiple processing cores, [sedona] uses a mixture of MPI and openMP parallelism. You should refer to the instructions on the system you are using to determine the appropriate commands to execute a parallel calculation. If using mpich, for example, the command to run with MPI parallelism is:

```
mpirun -n 2 ./sedona6.ex my_parameter_file.lua
```

where the number after `-n` is the number of mpi ranks to use (in this case 2).

To run using openMP threading, often you set the number of threads as an environment variable. In the bash shell, for example:

```
export OMP_NUM_THREADS=4
```

where the value of `OMP_NUM_THREADS` is the number of threads to use (in this case 4).

A key distinction between the two parallelism models is that openMP threads share the same memory whereas MPI ranks do not. Thus, each MPI rank used in [sedona] allocates a replica of the entire simulation grid and independently transports a subset of the Monte Carlo photon particles, with the results being communicated among all processors at the end of each transport step. In contrast, all openMP threads (on a given MPI rank) share and operate on the same simulation grid.

The `examples/` directory if the [sedona] installation provides example setups for a range of different science problems. This is a good place to start becoming familiar with calculations; below we describe a few instructive examples. If you are acustomed to runnning Jupyter notebooks, the directory `examples/jupyter_notebooks` provides some example notebooks.

2.1 Example #1: Spherical Lightbulb Test

Example problem located in directory:

```
$SEDONA_HOME/examples/spherical_lightbulb/1D
```

The *spherical lightbulb* is a quick test problem consisting of a spherical surface (the "core") that uniformly emits blackbody radiation into an optically thin medium. The radial dependence of the radiation field in steady state can be calculated analytically

$$T_{\rm rad}(r) = T_{\rm c}\left[\frac{1}{2}\left(1-\sqrt{1-R_{\rm c}^2/r^2}\right)\right]^{1/4}$$

where $R_{\rm c}$ is the radius and $T_{\rm c}$ the blackbody temperature of the emitting spherical core. The observed spectrum should be a blackbody of temperature $T_{\rm c}$.

To run the problem, change to the `examples/spherical_lightbulb/1D` directory and copy the executable *sedona6.ex* there. You will see a parameter file called *param.lua* which is used to set the runtime parameters. The comments in this file explain the meaning of the parameters being used in the run (see parameter_file for a full description of these files).

Run the code as described in the section above. The calculation should only take a few seconds. The code outputs two spectrum files (*spectrum_1.dat* and *spectrum_1.h5*) which both contain the same observed spectrum, just in different formats (ascii and hdf5 respectively). See section ?? for a more complete description of output spectrum files.

The code also outputs plot files (such as *plt_00001.dat* and *plt_00001.h5*) which contain information on the properties on the model grid. The files labeled *plt_00000* represent the initial conditions of the model, while those labeled *plt_00001* represent the state of the model after the first iteration (i.e., after the transport has run). The ascii format only contains a few of the basic radial variables, such as density, temperature, velocity. The hdf5 format also stores the detailed radiation properties in each zone (e.g., the frequency dependent spectrum and opacity, found within the `zonedata/` group). See section ?? for a more complete description of the plot files.

The file *integrated_quantities.dat* provides some of the model properties integrated over the entire grid, for the initial conditions and the state after each iteration.

Once you have successful run the code, you can try changing runtime parameters, such as the properties of the core, the number of photon packets emitted, or the opacity. The model density is set so that using a grey opacity = 1 gives an radial optical depth of 1.

2.2 Example #2: Type Ia Supernova Spectrum

Example problem located in directory:

```
$SEDONA_HOME/examples/supernova/TypeIa/spectrum
```

This example problem calculates a snapshot spectrum of a simple 1D Type Ia supernova. The input model file is located at `examples/supernova/TypeIa/models/toy_SNIa_1D.mod` and was generated by the `make_toy_SNIa_model.py` script in the same directory. The model consists of supernova ejecta moving in homologous expansion (i.e., velocity proportional to radius) with a power-law density profile. The mass of the ejecta is

1 $M_\odot$ and the kinetic energy is 10^{51} ergs. The compositional structure is composed of 3 layers: the inner $0.6 M_\odot$ of ejecta is pure radioactive ^{56}Ni the surrounding $0.3 M_\odot$ is a mixture of intermediate mass elements (Si, S, Ca) and the outer $0.1 M_\odot$ is a 50-50 mixture of carbon and oxygen. The radioactive ^{56}Ni will be the energy source of radiation to be observed in the final spectrum.

The directory has several different parameter files, each of which demonstrates a slightly different way of calculating the spectrum. Consider first the parameter file `param_d20_lte_exp.lua`. In this setup, the time after explosion at which we will calculate the spectrum is set to 20 days (which is set in units of seconds)

```
-- time of spectrum calculation
tstep_time_start = 20*3600.0*24.0
```

The parameter file instructs the code to run a steady state calculation at this time with 4 interations:

```
transport_steady_iterate  = 4
```

In a steady state calculation, the Monte Carlo photon packets are transported through a medium held stationary. This should be a reaonable approximation when the timescale for photons to diffuse through the ejecta is much faster than the time it takes for the ejecta to expand significantly in size. The velocity of the expanding medium is still considered to account for Doppler shift effects.

This steady state calculation requires multiple iterations in order to calculate the radial temperature structure, $T(r)$, of the ejecta. The temperature is here calculated under the assumption of *radiative equilibrium*, which means that in each zone the heating by absorption of photons is balanced by the cooling by the emission of photons. Radiative equilibrium is set by:

```
transport_radiative_equilibrium  = 1
```

On the first iteration, the code uses the $T(r)$ given in the model file, which is merely a guess at the true temperature structure. The code calculates the opacities and emissivities using this $T(r)$, and then transports photons packets through the domain, which determines the rate of radiative heating in each zone. After the transport step, the code solves for the temperature in each zone that required for radiative cooling to balance the heating rate. The new solved for $T(r)$ will differ from the initial guess, so this procedure must be iterated until convergence (i.e., until the temperature structure stops changing from one iteration to the next). For this problem, around 4 iterations are found to be adequate.

The number of particles propagated in each iteration is set by

```
-- radioactive particle emission
particles_n_emit_radioactive = 1e4
particles_last_iter_pump     = 10
```

The first parameter instructs the code to emit 10,000 photon packets from a radioacitve sources, in this case the ^{56}Ni present in the model ejecta. The packets will be emitted randomly throughout the zones that contain ^{56}Ni. The packets are originally created as gamma-rays produced from the radioactive decay, but in the process of the transport will be absorbed and re-emitted as optical photons.

While 10,000 photons are sufficient to calculate the temperature structure in this 1D model, more photons are desirable to get decent signal to noise in the output spectrum. The parameter `particles_last_iter_pump = 10` instructs the code to increase (pump-up) the number of packets by a factor of 10 (i.e., to 100,000) on the last iteration, when the temperature structure is presumed to be converged.

Other parameters in the parameter file define the opacity to be used in the calculation

```
-- opacity information
opacity_grey_opacity         = 0
opacity_electron_scattering  = 1
opacity_fuzz_expansion       = 0
```

(continues on next page)

(continued from previous page)

```
opacity_line_expansion       = 1
opacity_bound_bound          = 0
opacity_epsilon              = 1
```

This parameter file instructs the code to use electron scatter and line opacity in the expansion opacity formalism. The line opacity is chosen to be purely absorptive by setting opacity_epsilon = 1.0, where epsilon is the ratio of absorptive opacity to total (i.e., absorptive plus scattering) opacity.

Run the code using:

```
./sedona6.ex param_d20_lte_exp.lua
```

The code produces multiple "plt_00000x.dat" files, one for each iteration. These carry information about the physical properties on the grid after each iteration. You can examine them to ensure that the solution is converged (i.e., the temperature structure is no longer changing from one iteration to the next). The "plt_00000x.h5" carry even more information, in hdf5 format.

The code also produces multiple spectrum files; the one from the last iteration ("spectrum_4.dat") is the final result you are looking for (the output spectra from early iterations can be examined to double check that the solution is converged). The "spectrum_x.h5" files contain the same information but in hdf5 format.

The "spectrum_x.dat" files each have a one line header that describe the dimensions of the spectral output, with the format:

```
#  n_times    n_frequency   n_mu   n_phi
```

where n_times is the number of time steps (here 1 for a steady state calculation), n_frequency is the number of frequency points in the spectrum, n_mu is the number of viewing angle bins in the theta direction (where $\mu = \cos(\theta)$) and n_phi the number of viewing angle bins in the phi direction (both set to 1 here since the problem is spherically symmetric).

After the header, the spectrum files have 3 columns in the format:

```
frequency(Hz)   L_nu(ergs/s/Hz)    packet_count
```

The L_nu column gives the specific luminosity of the observed spectrum at that frequency. To transform this to a flux observed on earth (units: ergs/s/cm^2/Hz), simply divide L_nu by $4\pi D^2$ where D is the distance from the supernova to earth.

The other parameter files in the directory demonstrate slightly different ways in which you can calculate the spectrum of this model:

- **param_d20_lte_fuzz.lua:** This calculates the spectrum using line data from an alternative file, specified in the `data_fuzzline_file` parameter. The lines are treated in the expansion opacity formalism, so the results should be similar to the first example given, with differences being only due to the atomic data used.
- **param_d20_lte_bb.lua:** This calculates the spectrum using a more accurate treatment of lines (i.e., bound-bound transitions) by considering them independently instead of binning them together in the expansion opacity formalism. A higher freqeuency resolution is required to resolve the line profiles, and the calculation takes longer due to the higher opacities in the resolved line centers.
- **param_d20_lte_core.lua:** This calculates the spectrum using an absorbing spherical inner boundary surface (i.e., the photosphere, or *core*) at a radius set by the `core_radius` parameter. Photon packets are emitted as a blackbody from the surface of this core (rather than from the radioactive zones throughout the domain). This calculation is faster than the others since the optically thick inner regions of ejecta below the core are not included in the transport. Using a core is artifical, but the approximation can give reasonable results when the core is placed at high enough optically.

2.3 Example #3: Type Ia Supernova Light Curve

Example problem located in directory:

```
$SEDONA_HOME/examples/supernova/TypeIa/lightcurve
```

This problem uses the same Type Ia supernova model discussed in Example #2, but instead does a time-dependent calculation that evolves the ejecta from 2 days to 60 days after explosion. The final spectral output will be in the "spectrum_final.dat" file (as well as the "spectrum_final.h5" final) which now consists if a series of spectra, one for each time point. The "spectrum_final.dat" file now has four columns with data for:

```
time(seconds)  frequency(Hz)   L_nu(ergs/s/Hz)    packet_count
```

The light curve in any band can be calculated by integrating the spectra over the appropriate filter transmission function. Scripts for doing this are available in the *sedonalib* python package.

CHAPTER 3

Input Files

Sedona requires the user to supply two files that describe the properties of the desired run.

- *The Parameter File* is a text file (in the lua scripting language) that let's you set the desired runtime parameters of the job. By default, this file is assumed to be named **param.lua**
- *The Model File* specifies the grid geometry and the initial conditions (e.g., density, velocity, temperature) to be used in your calculation.

In addition, the user must point to two additional files

- The **defaults file** is a text file that specifies the default values of all runtime parameters. Any parameter not specifed in the *The Parameter File* will assume the default value given in this file. A standard defaults file is included in the Sedona distribution at **defaults/sedona_defaults.lua**. You can modify this file, or create different default files for different projects.
- The *Atomic Data File* provides the detailed properties (e.g., level energies, line transitions) of the atoms used for calculating opacities and emissivities. Existing atomic data files are provided in the **data/** directory of the Sedona release.

The names of the model file, the defaults file and the atomic data file are must be specified within the parameter file used for the run. The name of the parameter file itself is specified as a command line argument when running the code:

```
./sedona6.ex my_parameter_file.lua
```

If no argument is given after `./sedona6.ex` the code will assume the file is named **param.lua** and is in the current directory.

Input parameter and model files for several example setups for different sorts of science runs are provided in the `examples/` directory of the [sedona] distribution.

3.1 The Parameter File

All runtime parameters are set in the parameter file (assumed to be named **param.lua** unless another name is supplied on the command line argument). A complete list of [sedona] runtime parameters is compiled in the appendix. The pracitical usage of individual parameters described in the relevant sections throughout this documentation.

The parameter files uses the Lua scripting language. Each parameter is set on a different line. Scalar parameters are set as, e.g.,:

```
tstep_time_start   = 100.0
```

String parameters (such as filenames) are set using quotes, e.g.,:

```
model_file = "my_model.hdf5"
```

Vectors parameters are set using { } brackets, e.g.,:

```
transport_nu_grid   = {1e14,1e15,1e13}
```

Comments can be made in the lua parameter files by putting a double dash before a line, e.g.,

```
-- set uniform frequency grid with bounds (nu_start, nu_stop, nu_delta)
transport_nu_grid   = {1e14,1e15,1e13}
```

The lua scripting allows you to define helper variables and use math expressions within the parameter file itself. For example, we can define a variable called "days" and use it to more conveniently set runtime parameters

```
days = 86400 -- helper variable, seconds in a day

-- set start and stop time of calculation
tstep_time_start = 2*days
tstep_time_stop  = 100*days
```

You can also access environment variables within the parameter file using the `os.getenv()` command. For example, to set a local variable `sedona_path` to the environment variable `SEDONA_HOME`, use:

```
sedona_path    = os.getenv('SEDONA_HOME')
```

String concantention is done using double periods. For example, you can use the `sedona_path` variable defined above to specify the full path of a file

```
atomic_data_file = sedona_path.."/data/atomic_data.hdf5"
```

The parameter file **must** set the name of a default file, which is a file that uses the same lua scripting to define the default values of all runtime parameters. A standard defaults file is included in the [sedona] distribution and can be pointed in the parameter file by:

```
defaults_file     = sedona_path.."/defaults/sedona_defaults.lua"
```

You can create multiple defaults files and use them for different projects, and point to which one they wish to use in the parameter file of any given run.

While the lua scripting language is fairly convenient, the **sedonalib** python package also provides a python class that lets the user easily generate parameter files within a python script or jupyter notebook. See.. for usage.

3.2 The Model File

The model file defines the grid geometry, resolution and initial conditions to be used in the calculation. The grid geometries currently implemented in [sedona] are[1]

Table 1: Available Grid Geometries

Geometry name	Description
grid_1D_sphere	1D spherical coordinates (r)
grid_2D_cyln	2D cylndrical coordinates (r-z)
grid_3D_cart	3D cartesian coordinates (x,y,z)
grid_3D_sphere	3D spherical coordinates (r, θ, ϕ)

In addition to the geometry, the model file also specifies the following properties

- The number of zones each dimension
- The spatial size of zones in each dimension
- The list of atomic species to be used in the calculation (defined by their atomic number Z and atomic weight A)
- The time (in seconds) at which the model is defined (most relevant for homologously expanding models)

The model file also includes arrays that specify, for each zone, the values of the:

- density
- temperature
- velocity
- mass fraction of each atomic species being used
- radiation energy density (optional)

where all quantities are in cgs units.

The structure of the model file differs slightly depending on the geometry being used; the format for each case is described in the sections below. For 1-dimensional calculations, the model file can either be in ascii or in hdf5. For 2D and 3D calculations, the model file must be in hdf5 format.

3.2.1 1D spherical model (ascii format)

1D spherical models the zones are cocentric shells. The

In the ascii format, the 1D_sphere model files begin with the following 3 line header:

```
**geometry**  **type**

**n_zones**     **r0_in  time  n_species**

**:math:`Z.A_1`  Z_2.A_2  Z_3.A_3 ...**
```

where in the first line,

- **geometry** = grid geometry, one of the names in Table~.
- **type** = a subtype of the geometry

[1] Additional geometries can be defined by modifying the [sedona] source code. The grid structure is abstracted into a separate class that handles all geometrical information. Defining a new geometry thus requires writing a new derived C++ class that provides the requisite geometrical routines.

where **n_zones** is the number of radial zones (i.e., shells), **r0_in** is the inner radius of the innermost shell, **time** is the time at which the model is defined, and **n_species** is the number of atomic species to be used.

For example, this header

```
1D_SPHERE standard
100  0.0  86400  4
1.1. 2.4 6.12 8.16
```

states that the model has 100 radial zones, the inner radius of the innermost zone is $r_{0,in} = 0$, and the time of the model is 86400 seconds (i.e., one day). There are 4 atomic species used in the model, which are specifie d on the third line in the Z.A format to be: 1.1 = hydrogen-1 (^{1}H), 2.4 = helium-4 (^{4}He), 6.12 (carbon-12, ^{12}C) and 8.16 (oxygen-16, ^{16}O)

After the header follows a set of columns with n_zones rows that specify the properties of each zone

r_out v_out density temperature X_1 X_2 X_3 . . . *

where **r_out** is the outer radius of a zone (i.e., shell), **v_out** is the velocity at the outer radius of the zone,

3.2.2 1D spherical model (hdf5 format)

Has the following hdf5 datasets

Table 2: 1D_sphere model hdf5 data sets

dataset	dimensions	description
r_out	double[n_zones]	outer radius of a zone (i.e., shell)
rho	double[n_zones]	density of the gas in each zone
temp	double[n_zones]	temperature of the gas in each zone

3.2.3 2D cylndrical

term (up to a line of text) Definition of the term, which must be indented

and can even consist of multiple paragraphs

next term Description.

3.3 Atomic Data File

Atomic datafiles hold detailed information about atoms, and can be found in the **data/** folder.

Essential atomic data is compiled into a single hdf5 file. Additional atomic line data (e.g., Kurucz line lists) can optionally be accessed using “fuzzline” files

Table 3: Atomic Data Files

parameter	values	definition
data_atomic_file	<string>	name of the atomic data file
data_fuzzline_file	<string>	name of fuzzline file to include extra “fuzz” lines

CHAPTER 4

Output Files

4.1 Spectrum Files

Files with the name **spectrum_?.h5** carry the emergent radiation of the model. Output is always in frequency space, with the output luminosity in units of ergs/sec/Hz.

Note that the time spacing and frequency spacing of the output spectrum need not be identical to that used in the transport calculation. However, it is sensible to use time and frequency resolutions that are comparable to or coarser than that used in the transport run.

To set the frequency grid of the output spectrum, set the runtime parameter:

```
spectrum_nu_grid   = {start,stop,delta}
```

where gives a uniform frequency grid between values **start** and **stop** with spacing **delta**. To use a logarithmically spaced frequency grid, add an extra entry of 1:

```
spectrum_nu_grid   = {start,stop,delta,1}
```

where now the spacing between points is dnu = nu*delta. To perform a single frequency (i.e., grey) calculation, simply set start >= stop, e.g.,:

```
spectrum_nu_grid   = {1,1,1}
```

The spacing of the time bins is set in a similar way, e.g.,:

```
spectrum_time_grid   = {start,stop,delta}
```

which gives a uniformly spaced time gridbetween values **start** and **stop** with spacing **delta**.

Parameters controlling spectrum output

Table 1: Output Spectrum Parameters

parameter	values	definition
spectrum_name	<string>	name of the output spectrum files, if output_write_radiation is enabled
spectrum_time_grid	<float vector>	time grid for the spectrum file
spectrum_nu_grid	<float vector>	frequency grid for the spectrum file
spectrum_n_mu	<integer>	number of evenly spaced mu (viewing angles in theta (polar coord.) direction mu = cos theta)
spectrum_n_phi	<integer>	number of evenly spaced phi (viewing angles in phi (polar coord.) direction)
gamma_name	<string>	name of the output gamma-ray spectrum, if radioactivity is being used
gamma_nu_grid	<float vector>	grid for output gamma-rays; dimensions here are MeV

4.2 Plt Files

Plt files contain data describing the physical properties (e.g., temperature, density) of the model and different time steps or iterations. Exhaustive information is contained in the **plt_?????.h5** files, in hdf5 form. For 1D calculations, some information is also output (for convenience) in ascii **plt_?????.dat** files.

Several runtime parameters control when and what is written to plt files. To control when plt files are output, set e.g.,

```
output_write_plt_file_time = 1000.0
```

which will write a plt file every 1000.0 seconds of simulation time. To write plt files out with logarithmic spacing use e,g.,:

```
output_write_plt_log_space = 0.5
```

In which case, if a plt file is written at time t0, the next plt file will be written at time t1 = t0*(1 + 0.5). This parameter will override the **output_write_plt_file_time** parameter

Table 2: plt File Output Parameters

parameter	values	definition
output_write_plt_file_time	<float>	interval of simulation time (in seconds) before writing next plt file
output_write_plt_log_space	<float>	using logarithmic spacing for plt file output. If equal to 0, use equal spacing set by output_write_plt_file_time. If > 0 will override write_plt_file_time
output_write_radiation	0 = no | 1 = yes	Write out frequency dependent radiation properites (e.g., opacity, emissivity, Jnu) for every zone
output_write_atomic_levels	0 = no | 1 = yes	Write out detailed level populations for every zone
output_write_mass_fractions	0 = no | 1 = yes	Write out the composition (mass fractions) for every zone

4.3 Checkpoint Files

Checkpoint files provide a complete dump of the state of the program, and are used to restart a calculation. By default, checkpointing will not happen.

Several parameters are available to control checkpointing. For example, setting:

```
run_do_checkpoint = 1
run_chk_timestep_interval = 10
```

will write a checkpoint file every 10 time steps. Meanwhile setting:

```
run_do_checkpoint = 1
run_chk_simtime_interval = 1000.0
```

will write a checkpoint file every 1000.0 seconds of time elapsed in the simulation. To make sure that a checkpoint is written before a job is scheduled to die, use:

```
run_do_checkpoint = 1
run_chk_walltime_max = 12*60.0*60.0
run_chk_walltime_max_buffer = 1.1
```

will make sure that a checkpoint is written when the code thinks the next time step will not complete within the set max walltime of 12 hours. This estimated by determining if the time left before 12 hours is less than the time it took to compute the last time step, multiplied by a buffer (here = 1.1) for safety.

Table 3: Checkpoint Parameters

parameter	values	definition
run_do_restart	0 = no | 1 = yes	Whether or not to restart from a checkpoint file. If 0, starts a fresh run. Otherwise, restarts from run_restart_file.
run_restart_file	<string>	Name of file to restart from (e.g., chk.h5)
run_do_checkpoint	0 = no | 1 = yes	Whether or not to writeout checkpoint files. Note, that one of the interval parameters below must also be specified to write checkpoints
run_checkpoint_name_base	<string>	Filename prefix for checkpoint files
run_chk_timestep_interval	<int>	If 0, don't checkpoint based on simulation iteration number. Otherwise, checkpoint every $run_chk_timestep_interval timesteps.
run_chk_walltime_interval	<float>	If 0, don't checkpoint based on wallclock time. Otherwise, checkpoint $run_chk_walltime_interval after the last checkpoint in wallclock time. Measured in seconds
run_chk_simtime_interval	<float>	If 0, don't checkpoint based on simulation time. Otherwise, checkpoint $run_chk_simtime_interval after the last checkpoint in simulation time. Measured in seconds,
run_chk_walltime_max	<float>	If 0, don't checkpoint based on when the simulation will end. Otherwise, checkpoint when the simulation thinks it might not finish before $run_chk_walltime_max of wall-clock time has elapsed since the start of the run. Checkpoints based on this condition happen when ${run_chk_walltime_max_buffer} * (walltime duration of last timestep) + (current walltime) >= ${run_chk_walltime_max}. Measured in seconds, default is 0. This time should probably be the wallclock limit on your run.
run_chk_walltime_max_buffer	<float>	See above. Default is 1.1. Setting this to 0 will also turn off checkpointing based on run_chk_walltime_max
run_chk_number_start	<int>	Number with which to start checkpoint file numbering.
run_do_checkpoint_test	0 = no | 1 = yes	**Whether to save out a checkpoint file immediately after reading in a restart file. If you choose to run** restart file and this initial checkpoint file (named {$run_checkpoint_name_base}_init.h5) should return empty.

CHAPTER 5

Basic Code Execution

5.1 Time Dependent Calculations

Calculations in **sedona** can be run either as time evolving or steady-state models. This is controlled by time-stepping parameters

Table 1: Time Stepping Parameters

parameter	values	definition
tstep_max_steps	<integer>	Maximum number of time steps to take before exiting
tstep_time_start	<real>	Start time (in seconds)
tstep_time_stop	<real>	Stop time (in seconds)
tstep_max_dt	<real>	Maximum value of a time step (in seconds)
tstep_min_dt	<real>	Minimum value of a time step (in seconds)
tstep_max_delta	<real>	Maximum fractional size of a timestep, restricts dt to the specified value multiplied by the current time

Times are always in seconds.

5.2 Steady State Calculations

CHAPTER 6

Radiation Transport

6.1 Controlling Transport

Table 1: Radiation Transport Parameters

parameter	values	definition
transport_module	“monte_carlo”	What method to use for transport. Currently only monte carlo is implemented.
transport_nu_grid	<float vector>	Define frequency grid used for transport and opacities
transport_radiative_equilibrium	0 = no | 1 = yes	Whether to solve for radiative equilibrium
transport_steady_iterate	<integer>	Do a steady-state calculation with this number of iterations
transport_boundary_in_reflect	0 = no | 1 = yes	
transport_boundary_out_reflect	0 = no | 1 = yes	
transport_store_Jnu	0 = no | 1 = yes	
transport_use_ddmc	0 = no | 1 = yes	Whether to use discrete diffusion monte carlo
transport_ddmc_tau_threshold	<float>	At what optical depth ddmc takes over
transport_fleck_alpha	<float>	fleck alpha parameter (needs to be between 0.5 and 1 for ddmc)
transport_solve_Tgas_with_updated_opacities	0 = no | 1 = yes	whether to solve for Tgas after updating opacities
transport_fix_Tgas_during_transport	0 = no | 1 = yes	whether to fix Tgas
transport_set_Tgas_to_Trad	0 = no | 1 = yes	whether to set Tgas to Trad instead of solving for it

Note: The parameter transport_steady_iterate .. ? should really be under tstep, since it controls time evolution not transport per se.

6.2 Interaction Processes

6.2.1 Electron Scattering

6.2.2 Compton Scattering

6.2.3 Resonant Line Scattering

CHAPTER 7

Radiation Sources

7.1 Radioactivity

Table 1: Radioactivity Parameters

parameter	values	definition
maximum_half_life_seconds	<real>	the largest acceptable half life (in seconds). Radioactive nuclei with half-lives larger than this will not undergo decay
parent_isotopes_to_use	<string> of isotopes in the form "Z1.A1 Z2.A2 … Zn.An"	select the earliest nuclei to consider for each decay chain you would like to include. Nuclei that are earlier in each decay chain will not undergo decay
force_rprocess_heating	0 = no | 1 = yes	force the simulation to include r-process heating
partial_rprocess_heating	?	?

7.2 Thermal Emission

7.3 Radiating Core

Table 2: Radiating Core Parameters

parameter	values	definition
core_n_emit	<integer>	Number of particles to emit from core per time step (or iteration)
core_radius	<real>	Radius (in cm) of emitting spherical core
core_luminosity	<real> or <function>	Luminosity (in erg/s) emitted from core
core_temperature	<real>	Blackbody spectrum of core emission, if using blackbody emission
core_photon_frequency	<real>	Frequency of photons emitted from core, if using monochromatic emission
core_timescale	<real>	?
core_spectrum_file	<string>	filename of file to read to set spectrum of core emission
core_fix_luminosity	0 = no | 1 = yes	In steady state calculations, will rescale to fix output luminosity
particles_max_total	<float>	maximum number of particles (photons) allowed on the grid at the same time
particles_n_emit_radioactive	<integer>	number of particles emitted through radioactivity per timestep
particles_n_emit_thermal	<integer>	number of thermal particles emitted per timestep
particles_n_initialize	<integer>	number of particles used to initialize the simulation
particles_n_emit_pointsources	<integer>	
particles_pointsource_file	<string>	
particles_last_iter_pump		
multiply_particles_n_emit_by_dt_over_dtmax	0 = no | 1 = yes	
force_rprocess_heating	0 = no | 1 = yes	

7.4 Multiple Point Sources

CHAPTER 8

Opacity

There are several options for calculating and controlling the opacity of the gas in the calculation.

Though we generally use the word “opacity”, the code actually calculates and stores an extinction coefficient. The two are related by

$$\alpha = \kappa\rho$$

where α is the extinction coefficient (units cm^{-1}), κ is the opacity (units $\mathrm{cm}^2\ \mathrm{g}^{-1}$), and ρ is the mass density

8.1 Grey Opacity

The grey opacity flags allow the user to specify a simple, wavelength-independent opacity. *Note that setting grey opacity will override all other forms of opacity described below*.

To set a uniform grey opacity at all points in space, set the parameter:

```
opacity_grey_opacity = 0.1
```

where, in this example, the code will set the value $\kappa_g = 0.1\ \mathrm{cm}^2\ \mathrm{g}^{-1}$ to all zones in the model. All other sources of opacity discussed below (e.g., free-free, bound-free) will be ignored.

The user can also set a spatially varying grey opacity, provided that a dataset named grey_opacity giving the value of kappa in every zone has been set in an input hdf5 model file. Then setting:

```
opacity_zone_specific_grey_opacity    = 1
```

will use the kappa defined in the model file

8.2 Continuum opacities

8.2.1 Electron Scattering Opacity

To turn on electron-scattering opacity:

```
opacity_electron_scattering = 1
```

This is calculated as

$$\alpha_{\text{es}} = \sigma_t n_e$$

where n_e is the free electron density

Options for include Klein-Nishina corrections, Comptonization etc…

8.2.2 Bound-Free Opacity

8.2.3 Free-Free Opacity

8.3 Line Opacity

There are various options for treating lines. In general, only one of these approaches should be used to treat lines, unless one can be certain that line opacity is not be multiply counted.

8.3.1 Resolved Bound-Bound Opacity

This approach is selecting by setting the runtime parameter:

```
opacity_bound_bound = 1
```

This approach treats the lines generally as Voigt profiles. The frequency good must be fine enough that there are multiple grid points across to resolve the line profile.

The widths of lines can be artificially broadened using the runtime parameter:

```
line_velocity_width = <real>
```

where <real> is a velocity (in cm/s) by which the lines should be Gaussian broadened.

8.3.2 Line Expansion Opacity

This approach is selected by setting the runtime parameter:

```
opacity_line_expansion = 1
```

This approach bins lines into frequency bins, assuming a homologous flow.

This approach implies the Sobolev approximation. For each line, the code calculates the Sobolev optical depth

$$\tau_{\text{sob}} = \frac{\pi e^2}{m_e c} n_l f_{lu} \lambda_0 t_{\text{exp}}$$

where…

The expansion opacity is then calculated by binning lines

$$\alpha_{\rm exp} = \frac{1}{ct_{\rm exp}} \frac{\lambda_0}{\Delta\lambda} \sum_i (1 - e^{-\tau_i})$$

Some linelists (such as the Kurucz lists) are in a format that do not include as much detail about the atomic levels. These can be included as:

```
opacity_fuzz_expansion = 1
```

The physics here is identical to that of line expansion opacity, it is just that the lines are read from an independent file.

8.3.3 Resonant Line Scattering

This is how we would treat scattering in strong individual lines like Lyman alpha.

8.4 User Defined Opacity

You can write your own function

8.5 LTE and NLTE settings

8.6 Opacity Parameters

Table 1: Opacity parameters

parameter	values	definition
opacity_grey_opacity	<real>	value of grey opacity to use (in cm^2/g). Will override all other opacity settings
opacity_zone_specific_grey_opacity	0 = no | 1 = yes	Use a zone-specific grey opacity dataset that is set in an hdf5 input model file and named grey_opacity
opacity_user_defined	0 = no | 1 = yes	Calculate opacities by calling the function
opacity_epsilon	<float>	The fraction of
opacity_atom_zero_epsilon	<int>	
opacity_electron_scattering	0 = no | 1 = yes	include electron scattering opacity
opacity_line_expansion	0 = no | 1 = yes	include binned line expansion opacity
opacity_fuzz_expansion	0 = no | 1 = yes	include binned line expansion opacity, taken from a fuzz file
opacity_bound_free	0 = no | 1 = yes	include bound-free (photoionization) opacity
opacity_free_free	0 = no | 1 = yes	include free-free opacity
opacity_bound_bound	0 = no | 1 = yes	include bound-bound (resolved line) opacity
opacity_use_nlte	0 = no | 1 = yes	include nlte opacity
opacity_atoms_in_nlte	<int vector>	A vector of atomic numbers of the species to be treated in NLTE
opacity_use_collisions_nlte	0 = no | 1 = yes	only matters if use_nlte == 1, include collisions for nlte calculations
opacity_no_ground_recomb	0 = no | 1 = yes	Suppress all recombination transitions to the ground state in the NLTE level population solve
opacity_minimum_extinction	<float>	Minimum value of the extinction coefficient (units 1/cm) in any zone
opacity_maximum_opacity	<float>	Minimum value of the extinction coefficient (units 1/cm) in any zone
opacity_no_scattering	0 = no | 1 = yes	if = 1, will not include any kind of scattering opacity
dont_decay_composition		
opacity_compton_scatter_photons		
line_velocity_width	<float>	velocity in cm/s used to doppler broaden the (bound bound?) lines
line_x_extent		

CHAPTER 9

Hydrodynamics

9.1 Homologous Expansion

9.2 Lagrangian Hydrodynamics

Table 1: Hydrodynamics Parameters

parameter	values	definition
hydro_module	<string>	choose hydro module to evolve rho and T, options are homologous, none, 1D_lagrangian
hydro_gamma_index	<float>	gamma used in the eos
hydro_mean_particle_mass	<float>	mean particle mass mu, mass = mu * proton mass
hydro_cfl	<float>	cfl parameter used in hydro simulation to control time steps
hydro_v_piston	<float>	
hydro_viscosity_parameter		
hydro_central_point_mass		
hydro_use_gravity		Whether to include gravity for the hydro simulation
hydro_use_transport	0 = no | 1 = yes	Whether to use radiation transport (?)
hydro_accrete_radius		
hydro_bomb_radius	<float>	Radius of the bomb in cm
hydro_bomb_energy	<float>	Energy of the bomb in erg
hydro_boundary_outflow	0 = no | 1 = yes	
hydro_boundary_rigid_outer_wall	0 = no | 1 = yes	

CHAPTER 10

Plotting and Data Visualization

10.1 Generating Synthetic Light Curves

Sedona default outputs a file called *spectrum.h5* that gives the time series of the light curve $L_{\{\nu\}}(\{t\})$ at frequencies $\{\nu\}$ and output times $\{t\}$.

The bolometric luminosity is simply given as

$$L_{bol}(\{t\}) = \int_{\{\nu\}} L_{\{\nu\}}(\{t\})\, d\nu$$

Similarly, the absolute bolometric magnitude is given as

$$M_{bol} = -2.5 \log_{10} L_{bol} + 88.697425$$

In order to get light curves in certain filters, you have to convolve it with a given transmission curve.

If $T_b(\nu)$ is the transmission for a given filter band at frequency ν, then the luminosity convolved with the filter is expressed as

$$\mathcal{L}_\nu(b) = \frac{\int T_b(\nu) L_\nu \, d\ln\nu}{\int T_b(\nu)\, d\ln\nu}$$

The formula to convert this to an AB magnitude is

$$M_{AB}(b) = -2.5 \log_{10}\left(\frac{\mathcal{L}_\nu(b)}{4\pi d^2}\right) - 48.600$$

where $d = 10$ pc is the standardized distance to convert to a flux.

Note that $T_b(\nu)$ is here expressed as an energy-counting response.

A script that generates synthetic light curves from the Sedona *spectrum.h5* file is provided in the directory *tools/lightcurve_tools*. The python program *lcfilt.py* takes as input a *spectrum.h5* file and a list of filters/bands and converts the raw spectrum output to light curves.

To run, simply call:

```
python lcfilt.py -s <spectrum.h5> -b <band1,band2,...>
```

where *spectrum.h5* points to the Sedona spectrum file, and <*band1,band2,. . . > is a comma-separated list of filters that you wish to make light curves in. For example, if I wanted to generate synthetic light curves of a file */path/to/supernova_spectra.h5* in the LSST bands, then one would call:

```
python lcfilt.py -s /path/to/supernova_spectra.h5 -b LSST_u,LSST_g,LSST_r,LSST_i,LSST_
↪z,LSST_y
```

The light curve table is outputted in the file *lightcurve.out* and gives the time, bolometric luminosity, bolometric magnitude, and absolute magnitudes in the specified bands, using the AB magnitude system.

A full list of filters can be accessed by calling:

```
python lcfilt.py --bands
```

or by examining the file *FILTER_LIST*, which also contains references for the filters.

To add a filter not provided, add an entry to the end of *FILTER_LIST* and append the transmission curve (in Angstroms and relative response) to the **end** of *allfilters.dat*.

CHAPTER 11

All Runtime Parameters

sedona uses cgs units everywhere

Table 1: Time Stepping Parameters

parameter	values	definition
tstep_max_steps	<integer>	Maximum number of time steps to take before exiting
tstep_time_start	<real>	Start time (in seconds)
tstep_time_stop	<real>	Stop time (in seconds)
tstep_max_dt	<real>	Maximum value of a time step (in seconds)
tstep_min_dt	<real>	Minimum value of a time step (in seconds)
tstep_max_delta	<real>	Maximum fractional size of a timestep, restricts dt to the specified value multiplied by the current time

Table 2: Hydrodynamics Parameters

parameter	values	definition
hydro_module	<string>	choose hydro module to evolve rho and T, options are homologous, none, 1D_lagrangian
hydro_gamma_index	<float>	gamma used in the eos
hydro_mean_particle_mass	<float>	mean particle mass mu, mass = mu * proton mass
hydro_cfl	<float>	cfl parameter used in hydro simulation to control time steps
hydro_v_piston	<float>	
hydro_viscosity_parameter		
hydro_central_point_mass		
hydro_use_gravity		Whether to include gravity for the hydro simulation
hydro_use_transport	0 = no | 1 = yes	Whether to use radiation transport (?)
hydro_accrete_radius		
hydro_bomb_radius	<float>	Radius of the bomb in cm
hydro_bomb_energy	<float>	Energy of the bomb in erg
hydro_boundary_outflow	0 = no | 1 = yes	
hydro_boundary_rigid_outer_wall	0 = no | 1 = yes	

Table 3: Model File Parameters

parameter	values	definition
grid_type	"grid_1D_sphere", "grid_2D_cyln", "grid_3D_cart"	grid geometry; must match input model
model_file	<string>	Name of model file

Table 4: Atomic Data Files

parameter	values	definition
data_atomic_file	<string>	name of the atomic data file
data_fuzzline_file	<string>	name of fuzzline file to include extra "fuzz" lines

Table 5: Opacity parameters

parameter	values	definition
opacity_grey_opacity	<real>	value of grey opacity to use (in cm^2/g). Will override all other opacity settings
opacity_zone_specific_grey_opacity	0 = no | 1 = yes	Use a zone dependent grey opacity dataset set in an hdf5 input model file
opacity_user_defined	0 = no | 1 = yes	Calculate opacities by calling the function
opacity_epsilon	<float>	The fraction of
opacity_atom_zero_epsilon	<int>	
opacity_electron_scattering	0 = no | 1 = yes	include electron scattering opacity
opacity_line_expansion	0 = no | 1 = yes	include binned line expansion opacity
opacity_fuzz_expansion	0 = no | 1 = yes	include binned line expansion opacity, taken from a fuzz file
opacity_bound_free	0 = no | 1 = yes	include bound-free (photoionization) opacity
opacity_free_free	0 = no | 1 = yes	include free-free opacity
opacity_bound_bound	0 = no | 1 = yes	include bound-bound (resolved line) opacity
opacity_use_nlte	0 = no | 1 = yes	include nlte opacity
opacity_atoms_in_nlte	<int vector>	A vector of atomic numbers of the species to be treated in NLTE
opacity_use_collisions_nlte	0 = no | 1 = yes	only matters if use_nlte == 1, include collisions for nlte calculations
opacity_no_ground_recomb	0 = no | 1 = yes	Suppress all recombination transitions to the ground state in the NLTE level population solve
opacity_minimum_extinction	<float>	Minimum value of the extinction coefficient (units 1/cm) in any zone
opacity_maximum_opacity	<float>	Minimum value of the extinction coefficient (units 1/cm) in any zone
opacity_no_scattering	0 = no | 1 = yes	if = 1, will not include any kind of scattering opacity
dont_decay_composition		
opacity_compton_scatter_photons		
line_velocity_width	<float>	velocity in cm/s used to doppler broaden the (bound bound?) lines
line_x_extent		

Table 6: Output Spectrum Parameters

parameter	values	definition
spectrum_name	<string>	name of the output spectrum files, if output_write_radiation is enabled
spectrum_time_grid	<float vector>	time grid for the spectrum file
spectrum_nu_grid	<float vector>	frequency grid for the spectrum file
spectrum_n_mu	<integer>	number of evenly spaced mu (viewing angles in theta (polar coord.) direction mu = cos theta)
spectrum_n_phi	<integer>	number of evenly spaced phi (viewing angles in phi (polar coord.) direction)
gamma_name	<string>	name of the output gamma-ray spectrum, if radioactivity is being used
gamma_nu_grid	<float vector>	grid for output gamma-rays; dimensions here are MeV

Table 7: plt File Output Parameters

parameter	values	definition
output_write_plt_file_time	<float>	interval of simulation time (in seconds) before writing next plt file
output_write_plt_log_space	<float>	using logarithmic spacing for plt file output. If equal to 0, use equal spacing set by output_write_plt_file_time. If > 0 will override write_plt_file_time
output_write_radiation	0 = no | 1 = yes	Write out frequency dependent radiation properites (e.g., opacity, emissivity, Jnu) for every zone
output_write_atomic_levels	0 = no | 1 = yes	Write out detailed level populations for every zone
output_write_mass_fractions	0 = no | 1 = yes	Write out the composition (mass fractions) for every zone

Table 8: Checkpoint Parameters

parameter	values	definition
run_do_restart	0 = no | 1 = yes	Whether or not to restart from a checkpoint file. If 0, starts a fresh run. Otherwise, restarts from run_restart_file.
run_restart_file	<string>	Name of file to restart from (e.g., chk.h5)
run_do_checkpoint	0 = no | 1 = yes	Whether or not to writeout checkpoint files. Note, that one of the interval parameters below must also be specified to write checkpoints
run_checkpoint_name_base	<string>	Filename prefix for checkpoint files
run_chk_timestep_interval	<int>	If 0, don't checkpoint based on simulation iteration number. Otherwise, checkpoint every $run_chk_timestep_interval timesteps.
run_chk_walltime_interval	<float>	If 0, don't checkpoint based on wallclock time. Otherwise, checkpoint $run_chk_walltime_interval after the last checkpoint in wallclock time. Measured in seconds
run_chk_simtime_interval	<float>	If 0, don't checkpoint based on simulation time. Otherwise, checkpoint $run_chk_simtime_interval after the last checkpoint in simulation time. Measured in seconds,
run_chk_walltime_max	<float>	If 0, don't checkpoint based on when the simulation will end. Otherwise, checkpoint when the simulation thinks it might not finish before $run_chk_walltime_max of wall-clock time has elapsed since the start of the run. Checkpoints based on this condition happen when ${run_chk_walltime_max_buffer} * (walltime duration of last timestep) + (current walltime) >= ${run_chk_walltime_max}. Measured in seconds, default is 0. This time should probably be the wallclock limit on your run.
run_chk_walltime_max_buffer	<float>	See above. Default is 1.1. Setting this to 0 will also turn off checkpointing based on run_chk_walltime_max
run_chk_number_start	<int>	Number with which to start checkpoint file numbering.
run_do_checkpoint_test	0 = no | 1 = yes	**Whether to save out a checkpoint file immediately after reading in a restart file. If you choose to run** restart file and this initial checkpoint file (named {$run_checkpoint_name_base}_init.h5) should return empty.

Table 9: Radiation Transport Parameters

parameter	values	definition
transport_module	"monte_carlo"	What method to use for transport. Currently only monte carlo is implemented.
transport_nu_grid	<float vector>	Define frequency grid used for transport and opacities
transport_radiative_equilibrium	0 = no | 1 = yes	Whether to solve for radiative equilibrium
transport_steady_iterate	<integer>	Do a steady-state calculation with this number of iterations
transport_boundary_in_reflect	0 = no | 1 = yes	
transport_boundary_out_reflect	0 = no | 1 = yes	
transport_store_Jnu	0 = no | 1 = yes	
transport_use_ddmc	0 = no | 1 = yes	Whether to use discrete diffusion monte carlo
transport_ddmc_tau_threshold	<float>	At what optical depth ddmc takes over
transport_fleck_alpha	<float>	fleck alpha parameter (needs to be between 0.5 and 1 for ddmc)
transport_solve_Tgas_with_updated_opacities	0 = no | 1 = yes	whether to solve for Tgas after updating opacities
transport_fix_Tgas_during_transport	0 = no | 1 = yes	whether to fix Tgas
transport_set_Tgas_to_Trad	0 = no | 1 = yes	whether to set Tgas to Trad instead of solving for it

Table 10: Radiating Core Parameters

parameter	values	definition
core_n_emit	<integer>	Number of particles to emit from core per time step (or iteration)
core_radius	<real>	Radius (in cm) of emitting spherical core
core_luminosity	<real> or <function>	Luminosity (in erg/s) emitted from core
core_temperature	<real>	Blackbody spectrum of core emission, if using blackbody emission
core_photon_frequency	<real>	Frequency of photons emitted from core, if using monochromatic emission
core_timescale	<real>	?
core_spectrum_file	<string>	filename of file to read to set spectrum of core emission
core_fix_luminosity	0 = no | 1 = yes	In steady state calculations, will rescale to fix output luminosity
particles_max_total	<float>	maximum number of particles (photons) allowed on the grid at the same time
particles_n_emit_radioactive	<integer>	number of particles emitted through radioactivity per timestep
particles_n_emit_thermal	<integer>	number of thermal particles emitted per timestep
particles_n_initialize	<integer>	number of particles used to initialize the simulation
particles_n_emit_pointsources	<integer>	
particles_pointsource_file	<string>	
particles_last_iter_pump		
multiply_particles_n_emit_by_dt_over_dtmax	0 = no | 1 = yes	
force_rprocess_heating	0 = no | 1 = yes	

CHAPTER 12

Python Tools

12.1 Overview

The python tools is a small library of tools and classes useful for handling sedona in- and output. So far it consists of the following classes:

- `ModelFile` class (h5 files `MODEL.h5` and ASCII files `MODEL.mod`)
- `SliceFile` class (1d Castro output of the form `NAME.slice`)
- `PltFile` class (h5 files `pltXXXXX.h5`)
- `SpectrumFile` class (h5 files `SPECTRUM.h5`, produced by sedona if `output_write_radiation = 1` is set in the `param.lua`.)
- `PlotterClass` (Visualising DataFiles of all kind either in a set of plots, one combined plot or an animation of the time evolution)

There are also a number of scripts

- `lightcurve_tools.py` (An useful tool to create lightcurves from spectrum.h5 files. See section on *lightcurve_tools*. for more information eg. how to use it)
- `test_\*.py` (test scripts, all available in the `python_tools/tests/` folder. They show the basic utilisation of the Datafile/Plotter classes with the example data found in the `python_tools/data/ directory`).

To use any of the `test_*.py` files just type:

```
python test_REST_OF_NAME.py
```

in the `python_tools/tests/` folder.

12.2 Installation

To be able to use the tools, make sure, that you have downloaded the python_tools directory to your machine and add the following lines to the start of your python script:

```
import sys
path_to_tools = '../Classes/'
sys.path.append(path_to_tools)

import DataFile as DF
import Plotter as P
```

`path_to_tools` can either be a relative path to the python_tools/Classes/ directory or its absolute path (ie. `/usr/path/to/the/folder/python_tools/Classes/`).

12.3 File Handling

Generally speaking, Spectrum, Model, Plt and Slice files all have something in common: They consist of a large set of data table and a set of column names - they are some sort of data file. Examples on how to use them can be found in the python_tools/tests/ directory. There we can see, how to use the classes:

```
import DataFile as DF

MyFile = DF.ModelFile(NAME_OF_THE_FILE, PATH_TO_THE_FILE, autoload = False)
MyFile.load_data()
```

The `load_data()` is not necessary, if `autoload` is set to `True`, which is its default. `MyFile` is now a DataFile object (also named DFO). It comes with a list of capabilities, for example, once we have a file initialised, we can take a quick look at it:

```
MyFile.plot_data_1D()
```

This will open up a standard matplotib window. The console and the labels on the plot tell, what is plotted, if the arguments in plot_data_1D are empty, the program will look for the default keys:

Table 1: Default `x` and `y` keys

File Type	`default_x_key`	`default_y_key`
ModelFile	`'r'`	`'rho'`
SliceFile	`'x'`	`'density'`
PltFile	`'r'`	`'rho'`
SpectrumFile	`'nu'`	`Lnu_averaged'`, which is the automatically angle and time averaged Lnu

The following functions are defined for all data file types:

- `set_keyword`: Set a value in the data dictionary.

```
MyFile.set_keyword('numbers',              np.array([1,2,3,4,5]))
MyFile.set_keyword('other numbers',        np.array([5,4,3,2,1]))
MyFile.set_keyword('2D array of numbers', np.array([  [1,2,3,4,5],
                                                      [6,7,8,9,10],
                                                      [11,12,13,14,15],
```

(continues on next page)

(continued from previous page)

```
                                                    [16,17,18,19,20],
                                                    [21,22,23,24,25]] ))
```

Table 2: `set_keyword`

Argument	Type	Description
`key`	arbitrary, string recommended	key, at which the value is written
`value`	arbitrary, numpy array recommended	dictionary entry

- `get_value`: Return the set value in the data dictionary.:

```
numbers = Myfile.get_value('numbers')
```

Table 3: `get_value`

Argument	Type	Description
`key`	arbitrary, string recommended	key, at which the value is written

- `plot_data_1D`: Creates a 1D matplotlib plot of two 1D arrays `MyFile.data[x]` vs. `MyFile.data[y]`

```
MyFile.plot_data_1D(x = 'numbers', y = 'other numbers')
```

- `plot_data_2D`: Visualises a 2D array vs. two 1D arrays, ie. `MyFile.data[x]`, `MyFile.data[y]` vs. `MyFile.data[z]`

```
MyFile.plot_data_2D(x = 'numbers', y = 'other numbers', z = '2D array of numbers')
```

Table 4: `plot_data_1D and plot_data_2D`

Argument	Type	Description
`x`, `y`, `z` (for 2D only)	arbitrary, string recommended	key of the `x`/`y`/`z` data in the data dictionary, if no set it will default to the `default_x/y/z_key`
`plot_type`	string either `'std'`, `'logx'`, `'logy'` or `'loglog'`	Describes the axis scaling, standard (linear on both axes), semilogarithmic plot with logarithmic x/y axis or loglog plot
`plot_title`, `plot_xlabel`, `plot_ylabel`	string	Set these by hand to manually write title and labels of the plot. They **do not** affect the data plotted!
`plot_fmt`	string	format string used by `matplotib.pyplot.plot()`, eg. `'co'` for cyan dots or `'r--'` for a dashed red line.
`plot_save`	boolean	whether to save the plot or not. Default is `False`
`plot_save_type`	string	What file the plot is to be saved to, eg. `'pdf'` or `'JPG'`. Default is `'png'`
`interactive`	boolean	Whether or not to open up an interface, where two sets of buttons allow to change x and y. See *Plotting* for more information.

But there are also file type specific functions, which are

12.4 Tools

In the python_tools/tools directory there is a script called `lightcurve_tools.py`. It can be used to create the light curve in a given band. A list of all available bands can be found in the `FILTER_LIST` file in `python_tools/data/`. They can also be accessed by typing

```
python lightcurve_tools.py --bands
```

To produce a light curve just use:

```
python lightcurve_tools.py -s <path_to_spectrum_file> -b <space separated list of␣
↪bands>
```

The resulting lighcurve can be found in the file `./lightcurve.out`. It has 3 columns called *Time (Days)*, *Lbol (erg/s)*, *Mbol* and one for every band For example the `lightcurve.out` created by running

```
python lightcurve_tools.py -s ../data/spectrum_1D_TypeIa.h5 -b U B V
```

in the `python_tools/tools/` directory produces a `lightcurve.out` (in `python_tools/tools/`) with 6 data columns for time, bolometric luminosity, bolometric magnitude and the magnitude in the U, B and V band. All magnitudes are given in the AB Magnitude System.

Warning: Invalid elements are assigned a value of 0, the script also floors the magnitudes to 0 if <0

12.5 Plotting

There are also possibilities to compare single or multiple data files. The important basic examples can be found in `/tests/test_plotter.py` and `tests/test_datafilelasses.py`. The different types of plots available are:

For a single DataFile

- plot X vs. Y, where X and Y are two arrays found in the data
- plot X vs. Y, where two row of buttons let you switch between all available X and Y arrays
- plot X and Y vs. Z

For multiple DataFiles

- making subplots for a set of files (plotting their respective default X vs. default Y)
- combining a set of files into one single plot (again, plotting their respective default X vs. default Y)
- timeseries of a set of files, ie. plotting time dependent data into a short repeating movie. This works for single DataFiles, that have a time axis (ie SpectrumFiles) or a set of files, that have a time associated to them (so far only SliceFiles have been implemented)

A single DataFile plot can be opened without explicitly calling the plotter, for example by:

```
path = "python_tools/data/"
name = "plt19400.slice"

S = DF.SliceFile(name, path)
S.plot_data_1D()
```

The function `DATAFILE.plot_data_1D()` takes the arguments described in the *FileHandling* section. If a 2D plot is needed, use `DATAFILE.plot_data_2D()`, which can be used to visualise a 2D array of shape (A x B), using two 1D arrays of length A and B as x and y-axis. For an interactive plotting interface (for 1D plots), set `interactive = True` for the `DATAFILE.plot_data_1D()` function, ie.:

```
S.plot_data_1D(interactive = True)
```

The same plot will be opened, if

```
Plot = P.Plotter(DataFileObjects = [S])
Plot.start_1D_UI(mode = "interactiveplot")
```

is called. The first line initialises the Plotter object called Plot, the second starts the interface. **Note, that P.Plotter takes a list of DFOs to initialise!** The reason is, that normally one would want to compare a set of DFOs. Generally speaking, the Plotter class comes (aside from some convenience methods) with a one-for-all-function **‘‘start_1D_UI()‘‘**. Its most important argument is `mode`. `mode` can be one of the following options:

- `mode = 'interativeplot'`: Opens the same plot, as if `DATAFILE.plot_data_1D(interactive)` were called. Window consists of one plot and two columns of (radio) buttons, that can be used to select the data on the x and y axis.
- `mode = 'subplots'`: Open a window with N = (number of DFOs in the DataFileObjects list) subplots, one for every DFO. Plots default x vs. default y for every file. There is a button to en- and disable log scaling for the axes.
- `mode = 'onewindow'`: Open a window with one plot. As for `mode = 'subplots'` default x vs. default y is plotted for every file, but now in one combined plot. There is a button to en- and disable log scaling for the axes. **Warning:** The Plotter doesn't check the dimensions and units of the data, it assumes, that it makes sense, that all the data can be put into one plot window!
- `mode = 'timeseries'`: Open a window with one plot, a progress bar above it, and two buttons for the log scaling. If the Plotter object has been initialised with `DataFileObjects = [ONEFILE]`, then this file needs to have a time axis. So far, only `SPECTRUM.h5` files do. If the `DataFileObjects` list consists of more than one element, then it uses `DFO.find_time()` to see, if a time can be found. Once the data is loaded, the movie is created using the given fps number and film_duration and interpolating between snapshots created from the DFO(s).

CHAPTER 13

Adding Documentation

This documentation was constructed from .rst files in the **docs/sphinx_doc/** directory

For basics of using the reStructuredTex (.rst) markup language, see

http://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html

To compile the documentation into html or latex you must install Sphinx from

http://www.sphinx-doc.org/en/master/usage/installation.html

e.g., on a mac you can try to install using:

```
brew install sphinx-doc
```

Once installed, make the webpage version from the sphinx_doc/ directory using:

```
make html
```

The resulting webpage files appear in the _build/html/ directory. You can view them using:

```
open _build/html/index.html
```

To make a latex file use:

```
make latex
```

which makes a complete latex file in _build/latex/. To make the latex file and compile it into a pdf using pdflatex you can just use:

```
make latexpdf
```

CHAPTER 14

Indices and tables

- genindex
- modindex
- search