RH 1.5D Documentation

Release r79

Tiago M. D. Pereira

Nov 12, 2020

Contents

Contents:

CHAPTER 1

Introduction

1.1 What is RH 1.5D

RH 1.5D is a modified version of the RH radiative transfer code that runs through a 3D/2D/1D atmosphere column-by-column, in the same fashion of rhf1d. It was developed as a way to efficiently run RH column-by-column (1.5D) over large atmospheres, simplifying the output and being able to run in supercomputers. It is MPI-parallel and scales well to at least 10,000 processes.

While initially developed as another geometry on the RH tree, the requirements of the parallel version required changes to the RH core tree. Thus, it is more than a *wrapper* over RH and is distributed with a modified version of RH (see below for differences from RH distributed by Han Uitenbroek).

1.2 Acknowledging RH 1.5D

If you use RH 1.5D for your work, we would appreciate if you would acknowledge it appropriately in a publication, presentation, poster, or talk. For a publication, this is best done by citing the RH 1.5D (Pereira & Uitenbroek 2015) and RH (Uitenbroek 2001) papers. In addition, if the journal allows it please include a link to its Github repository.

1.3 What this manual covers

Because there is much in common with RH, this manual should be seen as an incremental documentation of the 1.5D parallel side. This manual focuses on what is different from RH. Users should refer to the RH documentation by Han Uitenbroek for more detailed information on RH.

1.4 Comparison with RH

RH 1.5D inherits most of the code base from RH, but some features are new. The code is organised in the same was as the RH source tree, with the routines specific to the 1.5D version residing on a subdirectory `rh15d` of the `rh` source tree. In this way, it works similarly to the subdirectories in `rh` for different geometries. The compilation and linking proceeds as for the other geometries: first the general `librh.a` library should be compiled, and then the code in `rh15d` will be compiled and linked to it. The run directory is very similar to that of a given geometry in RH: most of the `*.input` files are used in the same way.

The lists below show a comparison between RH 1.5D and RH for a 1D plane-parallel geometry:

1.4.1 Commonalities between RH 1.5D and RH

- Core RH library
- Structure and location of `*.input` files (some new options available)
- Wavetable, Atom, Molecule, line list, and any other input files except the atmospheres
- Directory-level compilation and general structure of run directory (new subdirectories needed)

1.4.2 What is new in RH 1.5D

- MPI-parallelism with dynamic load balancing and efficient I/O
- Different format of atmosphere files
- Different formats of output files
- Select which quantities should be in output
- New options for `keyword.input` and `atoms.input`
- Hybrid angle-dependent PRD mode
- PRD-switching
- Option for using escape probability approximation as initial solution
- Exclude from the calculations the higher parts of the atmosphere, above a user-defined temperature threshold
- Depth scale optimisation
- Option for cubic Hermite interpolation of source function in formal solver
- Option for cubic Bézier and DELO Bézier interpolation of source function in formal solvers, both for polarised and unpolarised light
- Support for more types of collisional excitations
- Easy re-run of non-converged columns
- Option for keeping background opacities in memory and not in disk
- New analysis suite in Python

1.4.3 What is not supported in RH 1.5D

- Currently only a fraction of the RH output is written to disk (to save space), but more output can be added
- The old IDL analysis suite does not currently support the new output format
- `solveray` is no longer used
- `backgrcontr` no longer works with the new output
- Any other geometry aside from 1.5D
- Continuing old run by reading populations and output files
- Full Stokes NLTE iterations and background polarisation (might work with little effort, but has not been tested)
- Thread parallelisation

CHAPTER 2

Installation

2.1 Getting the code

The code is available on a git repository, hosted on github: https://github.com/ita-solar/rh. If you don't have git installed and just want to get started, the easiest way is to download a zip file with the latest revision: https://github.com/ita-solar/rh/archive/master.zip. If you have git installed and would like to be up-to-date with the repository, you can do a git clone:

```
git clone https://github.com/ita-solar/rh.git
```

or using SSH:

```
git clone git@github.com:ita-solar/rh.git
```

Whether you unpack the zip file or do one of the above it will create a directory called `rh` in your current path. This directory will have the following subdirectories:

Directory	Contents
rh	Main RH source
rh/Atmos	Used to keep atmosphere files
rh/Atoms_example	Used to keep atom files, line and wavelength lists (example)
rh/idl	Old RH IDL routines, not used
rh/Molecules	Used to keep molecule files
rh/python	Utility Python programs
rh/rh15d.	Source files for RH 1.5D
rh/rhf1d	Source files for 1D geometry, **deprecated, do not use**
rh/rhsphere	Source files for spherical geometry, **deprecated, do not use**
rh/tools	Associate C programs for RH, not tested.

Warning: The source code directories for other geometries (rhf1d, rhsphere) are still in the code tree, but they are deprecated and will be removed soon. With the latest changes related to rh15d, they are not guaranteed to work or even run. Do not use.

2.2 Dependencies

2.2.1 HDF5

RH 1.5D makes use of the HDF5 library to read the atmosphere files and write the output. It is not possible to run the code without this library. RH 1.5D requires HDF5 version 1.8.1 or newer (including versions 1.10.x).

Note: RH 1.5D previously made use of the netCDF4 library for its output (which in turn also required HDF5). The latest changes mean RH 1.5D needs only HDF5. Because netCDF4 files are also HDF5 files, the output is still readable in the same way as before and input files in netCDF version 4 format can still be read in the same way by RH 1.5D. If you used input atmospheres in netCDF version 3 format, then these will have to be converted to HDF5. It is recommended that new atmosphere files be created in HDF5 only.

Because HDF5 is commonly used in high-performance computing, many supercomputers already have them available. In Fram, they can be loaded as (also loading the intel compilers):

```
module load HDF5/1.8.19-intel-2018a intel/2018a
```

in Pleiades:

```
module load hdf5/1.8.18_mpt
```

in Vilje:

```
module load intelcomp/18.0.1 mpt/2.14 hdf5/1.8.19
```

in Hexagon:

```
module load cray-hdf5-parallel
```

and at ITA's Linux system:

```
module load hdf5/Intel/1.8.19 Intel_parallel_studio/2018/3.051
```

2.2.2 MPI

RH 1.5D is parallelised via MPI, therefore an MPI library is necessary even if running with only one CPU. These are readily available in supercomputers and clusters, but not always in individual workstations. In such cases, users will have to manually install both MPI and HDF5 libraries (HDF5 should be compiled with the MPI library so that the parallel I/O module works).

If using Linux or MacOS, HDF5 is available in a variety of ways. The safest bet is to download and compile HDF5 from the source, enabling parallel builds in the `./configure` script, e.g.:

```
./configure (...) --enable-parallel
```

An easier way, but not guaranteed to work every time, is to install both MPI and HDF5 with parallel support is via the Anaconda Python distribution. Once you install Anaconda (version 3.6 or above), you can install the `hdf5-parallel` package, which installs both MPI and HDF5 and has been tested with RH 1.5D:

```
conda install -c spectraldns hdf5-parallel
```

Warning: If you obtain pre-compiled binaries or packages for HDF5, you need to make sure they have parallel support enabled. Most available packages **do not have parallel support**. This includes typical packages from Linux distributions and the `hdf5` Anaconda package (but not the above `hdf5-parallel` package).

2.3 Compilation

Compilation of RH 1.5D consists of two steps:

1. Compilation of the geometry-independent main libraries (`librh.a` and `librh_f90.a`)
2. Compilation of the `rh15d_mpi` tree and main binaries

RH 1.5D has been compiled in a variety of architectures and compilers, including gcc, the Intel compilers, and clang. As for MPI implementations, it has been tested with SGI's mpt, OpenMPI, mpich, mvapich, and Intel's MPI.

2.3.1 Makefile configuration

RH 1.5D does not automatically look for the compilers and libraries. You need to tell RH which compilers to use and where to find the HDF5 library by editing the file `rh/Makefile.config`. This file is also used to set up any additional compiler or linker flags, if appropriate. **Changes to any other Makefiles are not necessary.** It is also no longer necessary to set the environment variables `OS` and `CPU`, as in previous versions.

For `HDF5_DIR`, please enter the base directory for the library (not the directory with the lib* files), so that both library and include files are used. In Fram and Hexagon this is already stored in the `HDF5_DIR` environment variable, so you can comment that line in `Makefile.config`. If your version of HDF5 was not built as a shared binary, you need to link HDF5 and other used libraries directly (you will need to set at least `-lz` in `LDFLAGS`).

There are two steps in the compilation: main libraries and rh15d binaries. To speed up compilation, you can use parallel builds (e.g. `make -j8`) in all steps of the compilation.

2.3.2 Main libraries

The common RH files are put in a library under the base directory. After editing `Makefile.config`, build the main libraries with `make` on the `rh` directory. If successful, the compilation will produce the two library files `librh.a` and `librh_f90.a`.

2.3.3 Program binaries

The `rh15d` contains the source files for the 1.5D version. After compiling the main library, go to that directory and compile the binaries with `make`. The following executables will be created:

File	Description
rh15d_ray_pool	Main RH 1.5D binary, uses a job pool (see *Binaries and execution*)
rh15d_ray	Alternative RH 1.5D binary. **Deprecated.** This program runs much slower than rh15d_ray_pool and is kept for backwards compatibility only. Will be removed in a future revision.
rh15d_lteray	Special binary for running in LTE

2.4 Run directory

Once compiled, you can copy or link the binaries to a run directory. This directory will contain all the necessary input files, and it should contain two subdirectories called `output` and `scratch`.

Warning: If the subdirectories `output` and `scratch` do not exist in the directory where the code is run, the code will crash with an obscure error message.

CHAPTER 3

Input files

3.1 Configuration files

The configuration of an RH 1.5D is made primarily through several text files that reside in the run directory. The main file is `keyword.input`. All the other files and their locations are specified in `keyword.input`. The source tree contains a sample `rh/rh15d/run/` directory with the following typically used configuration files:

File	Description
atoms.input	Lists the atom files to be used
keyword.input	Main configuration file
kurucz.input	Contains list of line lists to be used
molecules.input	Lists the molecule files to be used
ray.input	Selects output mu and wavelengths for detailed output

The `kurucz.input` and the `molecules.input` files are identical under RH, so we refer to the RH manual for more information about them. Most of the other files behave very similarly in RH and RH 1.5D, with a few differences.

The `atoms.input` file is identical in RH, but it can also have a new starting solution, `ESCAPE_PROBABILITY`.

The `keyword.input` file functions in very much the same manner under RH and RH 1.5D. The main difference is that there are new options for the 1.5D version, and some options should not be used.

The new `keyword.input` options for the 1.5D version are:

Name	Default value	Description
SNAPSHOT	0	Snapshot index from the atmosphere file.
X_START	0	Starting column in the x direction. If < 0, will be set to 0.
X_END	-1	Ending column in the x direction. If <= 0, will be set to NX in the atmosphere file.
X_STEP	1	How many columns to sample in the y direction. If < 1, will be set to 1.
Y_START	0	Starting column in the y direction. If < 0, will be set to 0.
Y_END	-1	Ending column in the y direction. If <= 0, will be set to NY in the atmosphere file.
Y_STEP	1	How many columns to sample in the y direction. If < 1, will be set to 1.
15D_WRITE_POPS	FALSE	If TRUE, will write the level populations (including LTE) for each active atom to output_aux.hdf5.
15D_WRITE_RRATES	FALSE	If TRUE, will write the radiative rates (lines and continua) for each active atom to output_aux.hdf5.
15D_WRITE_TAU1	FALSE	If TRUE, will write the height of tau=1 to output_ray.hdf5, for all wavelengths (this takes up as much space as the intensity).
15D_RERUN	FALSE	If TRUE, will rerun for non-converged columns.
15D_DEPTH_ZCUT	TRUE	If TRUE, will perform a cut in z for points above a threshold temperature
15D_TMAX_CUT	-1	Threshold temperature (in K) over which the above depth cut ids made. If < 0, no temperature cut will be made.
15D_DEPTH_REFINE	FALSE	If TRUE, will perform an optimisation of the depth scale, based on optical depth, density and temperature gradients.
BACKGR_IN_MEM	FALSE	If TRUE, will keep background opacity coefficients in memory instead of scratch files on disk.
BARKLEM_DATA_DIR	../../Atoms	Directory where the Barklem_*data.dat data files are saved.
COLLRAD_SWITCH	0.0	Defines if collisional radiative switching is on. If < 0, switching parameter is constant (and equal to COLLRAD_SWITCH_INI). If = 0, no collisional radiative switching. If > 0, collisional radiative switching decreases by COLLRAD_SWITCH per log decade, starting with COLLRAD_SWITCH_INI.
COLLRAD_SWITCH_INIT	1.0	Initial increment for collisional-radiative switching
LIMIT_MEMORY	FALSE	If TRUE, will not keep several large arrays in memory but rather save them to scratch files. Not recommended unless memory usage is critical.
N_PESC_ITER	3	Number of escape probability iterations, if any atoms have it as initial solution.
PRD_SWITCH	0.0	If > 0, the PRD effects will be added gradually, converging to the full PRD solution in 1/sqrt(PRD_SWITCH) iterations.
PRDH_LIMIT_MEM	FALSE	If TRUE and using PRD_ANGLE_APPROX, will not keep in memory quantities necessary to calculate the current PRD weights, but rather calculate them again. Will affect the performance, so should be used only when necessary.
S_INTERPOLATION	LINEAR	Type of source function interpolation to use in formal solver. Can be LINEAR, BEZIER, BEZIER3 or CUBIC_HERMITE.

The X_START, X_END, and X_STEP keywords (and the equivalent for the y direction) define which columns of the atmosphere file are going to be run. They can be used to calculate only a specific region. RH 1.5D chooses the columns to calculate using the (start, end, step) parameters as in the range() function in Python: the result is [start, start + step, start + 2 * step, ...]. The last element is the largest start + i * step less than end. This means that the numbers given by X_END and Y_END are **not inclusive** (e.g. if nx = 50 and X_END = 49, the column with the index 49 will not be calculated). One must set X_END = nx to calculate all the columns.

The following options have a different meaning under RH 1.5D:

Name	Default value	Description
PRD_ANGLE_DEP	PRD_ANGLE_INDEP	This keyword is no longer boolean. To accommodate for new options, it now takes the values PRD_ANGLE_INDEP for angle-independent PRD, PRD_ANGLE_DEP for angle-dependent PRD, and PRD_ANGLE_APPROX for the approximate angle-dependent scheme of Leenaarts et al. (2012)[2] .
BACKGROUND_FILE		This keyword is no longer the name of the background file, but the prefix of the background files. There will be one file per process, and the filenames are this prefix plus _i.dat, where i is the process number.
STOKES_INPUT		This option is not used in RH 1.5D because the magnetic fields are now written to the atmosphere file. However, it **must** be set to any string if one is using any STOKES_MODE other than NO_STOKES (RH won't read B otherwise).

And the following options are valid for RH but may not work with RH 1.5D:

Name	Default value	Description
LIMIT_MEMORY	FALSE	This option has not been tested and may not work well with RH 1.5D.
PRINT_CPU	FALSE	This option does not work with RH 1.5D and **should always be** FALSE.
N_THREADS	0	Thread parallelism will not work with RH 1.5D. This option should always be 0 or 1.

The ray.input has the same structure in RH1D and RH 1.5D. In RH it is used as input for the solveray program, but in RH 1.5D it is used for the main program. It should contain the following:

```
1.00
Nsource
```

The first line is the mu angle for the output ray, and it should always be 1.00. The second line is Nsource, the number of wavelengths for which detailed output (typically source function, opacity, and emissivities) will be written. If Nsource > 0, it should be followed in the same line by the indices of the wavelengths (e.g. 0 2 10 20).

3.2 Atom and molecule files

The atom and molecule files have the same format as in RH. In the rh/Atoms and rh/Molecules directories there are a few sample files. They are read by the procedures in readatom.c and readmolecule.c. The atom files have the following basic structure:

[1] de la Cruz Rodríguez, J.; Piskunov, N. 2013, ApJ, 764, 33, ADS link.
[2] Leenaarts, J., Pereira, T. M. D., & Uitenbroek, H. 2012, A&A, 543, A109, ADS link.

Input	Format
`ID`	**(A2)**. Two-character atom identifier.
`Nlevel Nline Ncont Nfixed`	**(4I)**. Number of levels, lines, continua, and fixed radiation temperature transitions.
`level_entries`	Nlevel * **(2F, A20, I)**
`line entries`	Nline * **(2I, F, A, I, A, 2F, A, 6F)**
`continuum_entries`	Ncont * **(I, I, F, I, A, F)**
`fixed_entries`	Ncont * **(2I, 2F, A)**

3.3 Atmosphere files

The atmosphere files for RH 1.5D are a significant departure from RH. They are written in the flexible and self-describing HDF5 format. They can be written with any version, except the 1.10.x development branch.

The atmosphere files contain all the atmospheric variables necessary for RH 1.5D, and they may contain one or more simulation snapshots. The basic dimensions of the file are:

`nt`	Number of snapshots.
`nx`	Number of x points
`ny`	Number of y points.
`nz`	Number of depth points.
`nhydr`	Number of hydrogen levels.

While strictly 3D atmosphere files, 2D and 1D snapshots can also be used provided that one or both of `nx` and `ny` are equal to 1.

Note: The atmosphere variables must be written to the file in a particular way. They should be written in a *height grid* (meaning the top of the atmosphere has a larger value of `z`), and must *start from the top* (meaning that the first height index of the arrays must be the **TOP** of the atmosphere). Failure to follow these two rules can either lead to RH aborting a run, or worse, getting wrong results without a clear error message.

The atmosphere file can contain the following variables:

Name	Dimensions	Units	Notes
B_x	(nt, nx, ny, nz)	T	Magnetic field x component. **Optional**
B_y	(nt, nx, ny, nz)	T	Magnetic field y component. **Optional**
B_z	(nt, nx, ny, nz)	T	Magnetic field z component. **Optional**
electron_density	(nt, nx, ny, nz)	m^{-3}	**Optional**.
hydrogen_populations	(nt, nhydr, nx, ny, nz)	m^{-3}	nhydr must correspond to the number of levels in the hydrogen atom used. If nhydr=1, this variable should contain the total number of hydrogen atoms (in all levels), and LTE populations will be calculated.
snapshot_number	(nt)	None	The snapshot number is an array of integers to identify each snapshot in the output files.
temperature	(nt, nx, ny, nz)	K	
velocity_z	(nt, nx, ny, nz)	m s^{-1}	Vertical component of velocity. **Positive velocity is upflow.**
velocity_turbulent	(nt, nx, ny, nz)	m s^{-1}	Turbulent velocity (microturbulence).
z	(nt, nx, ny, nz) or (nt, nz)	m	Height grid. Can vary with column and snapshot. First nz index is **top of the atmosphere** (closest to observer).

Any other variable in the file will not be used. In addition, the atmosphere file **must** have a global attribute called has_B. This attribute should be 1 when the magnetic field variables are present, and 0 otherwise. Also recommended, but optional, is a global attribute called description with a brief description of the atmosphere file (e.g. how and from they were generated).

Note: Variables in the atmosphere files can be compressed (zlib or szip), but compression is not recommended for performance reasons.

As HDF5 files, the contents of the atmosphere files can be examined with the h5dump utility. To see a summary of what's inside a given file, one can do:

```
h5dump -H atmosfile
```

Here is the output of the above for a sample file:

```
HDF5 "example.hdf5" {
GROUP "/" {
   ATTRIBUTE "boundary_bottom" {
      DATATYPE  H5T_STD_I64LE
      DATASPACE  SCALAR
   }
   ATTRIBUTE "boundary_top" {
      DATATYPE  H5T_STD_I64LE
      DATASPACE  SCALAR
   }
   ATTRIBUTE "description" {
      DATATYPE  H5T_STRING {
         STRSIZE H5T_VARIABLE;
         STRPAD H5T_STR_NULLTERM;
         CSET H5T_CSET_UTF8;
         CTYPE H5T_C_S1;
      }
      DATASPACE  SCALAR
   }
   ATTRIBUTE "has_B" {
      DATATYPE  H5T_STD_I64LE
      DATASPACE  SCALAR
   }
   ATTRIBUTE "nhydr" {
      DATATYPE  H5T_STD_I64LE
      DATASPACE  SCALAR
   }
   ATTRIBUTE "nx" {
      DATATYPE  H5T_STD_I64LE
      DATASPACE  SCALAR
   }
   ATTRIBUTE "ny" {
      DATATYPE  H5T_STD_I64LE
      DATASPACE  SCALAR
   }
   ATTRIBUTE "nz" {
      DATATYPE  H5T_STD_I64LE
      DATASPACE  SCALAR
   }
   DATASET "electron_density" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 1, 512, 512, 425 ) / ( H5S_UNLIMITED, 512, 512, 425 ) }
   }
   DATASET "hydrogen_populations" {
      DATATYPE  H5T_IEEE_F32LE
      DATASPACE  SIMPLE { ( 1, 6, 512, 512, 425 ) / ( H5S_UNLIMITED, 6, 512, 512,␣
↪425 ) }
   }
   DATASET "snapshot_number" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   }
   DATASET "temperature" {
      DATATYPE  H5T_IEEE_F32LE
      DATASPACE  SIMPLE { ( 1, 512, 512, 425 ) / ( H5S_UNLIMITED, 512, 512, 425 ) }
   }
   DATASET "velocity_z" {
      DATATYPE  H5T_IEEE_F32LE
      DATASPACE  SIMPLE { ( 1, 512, 512, 425 ) / ( H5S_UNLIMITED, 512, 512, 425 ) }
   }
   DATASET "x" {
      DATATYPE  H5T_IEEE_F32LE
```

(continues on next page)

(continued from previous page)

```
      DATASPACE  SIMPLE { ( 512 ) / ( 512 ) }
   }
   DATASET "y" {
      DATATYPE  H5T_IEEE_F32LE
      DATASPACE  SIMPLE { ( 512 ) / ( 512 ) }
   }
   DATASET "z" {
      DATATYPE  H5T_IEEE_F32LE
      DATASPACE  SIMPLE { ( 1, 425 ) / ( H5S_UNLIMITED, 425 ) }
   }
}
}
```

All the floating point variables can be either double or single precision.

3.4 Line lists and wavelength files

Other auxiliary files that can be used are line lists files and wavelength files.

The line list files are used to include additional lines not included in the different atoms. These lines will be treated in LTE. The line lists are specified in the `kurucz.input` file (one per line), and have the Kurucz line list format (link).

Just adding new transitions doesn't mean that they will be included in the synthetic spectra. The extra lines will only be included in the existing wavelength grid, which depends on the active atoms used. The calculation of additional wavelengths can be forced by using a wavelength file. This file is specified in `keyword.input` using the keyword `WAVETABLE`. The format is a binary XDR file. Its contents are, in order: the number of new wavelengths (1 XDR int), vacuum wavelength values (XDR doubles).

CHAPTER 4

Running the code

4.1 Binaries and execution

Compilation should produce three executables: `rh15d_ray_pool`, `rh15d_ray`, and `rh15d_lteray`. The latter is a special case for running only in LTE. The other two are the main programs. They represent two different modes of job distribution: *normal* and *pool*.

In the *pool* mode there is a process that works as *overlord*: its function is to distribute the work to other processes. The other processes (*drones*) ask the *overlord* for a work unit. When they finish their unit, they go back and ask for more, until all tasks are completed. Because of race conditions and because different columns will run at different speeds, it is not possible to know which columns a given process will run beforehand. Due to the *overlord*, `rh15d_ray_pool` needs to run with two or more processes. The advantage of the *pool* mode is that the dynamic load allocation ensures the most efficient use of the resources. With the normal mode it may happen that some processors will work on columns that take longer to converge (especially as they are adjacent), and in the end the execution will have to wait for the process that takes longer. In some cases (especially with PRD) the *pool* mode can be 2-3 times faster than the *normal* mode. When one runs with a large number of processes (> 2000) and each column takes little time to calculate, the *pool* mode can suffer from communication bottlenecks and may be slower because a single *overlord* cannot distribute the tasks fast enough. The only disadvantage of the pool mode (so far) is that not all output is currently supported with this mode.

Warning: The *normal* mode is deprecated and will be removed in a later revision. Use only for single processor runs or if you know what you're doing!

In the *normal* mode the jobs (all the atmosphere columns for which one wants to calculate) are divided by the number of processes at the start of the execution. There is no communication between processes, and each process knows from the start all the columns it is going to run. These columns are adjacent. If the number of columns is not a multiple of the number of processes, there will be some processes with larger workloads. There is no minimum number of processes to run, and `rh15d_ray` can also be run in a single process. Regions of an atmosphere can take a lot longer to run than others, and the processes that work on those will take longer to finish. In the *normal* mode this means that the slowest process will set the overall running time, and therefore in practice it can take more than 10x longer than the *pool* mode (and is therefore not recommended).

As an MPI program, the binaries should be launched with the appropriate command. Some examples:

```
mpirun -np N ./rh15d_ray_pool
mpiexec ./rh15d_ray_pool    # use in Pleiades
aprun -B ./rh15d_ray        # use in Hexagon or other Cray
```

4.2 The run directory

Warning: Before running, make sure you have the sub-directories `scratch` and `output` in the run directory.

The run directory contains the configuration files, the binaries, and the `scratch` and `output` directories. As the names imply, temporary files will be placed under `scratch` and the final output files in `output`. No files under `scratch` will be used after the run is finished (they are not read for re-runs).

The `scratch` directory contains different types of files. Most of them are binary files write by RH 1.5D to save memory. Example files are the `background_p*.dat` with the background opacities, files with PRD weights, and the `rh_p*.log` log files. Each process creates one of those files, and they will have the suffix `_pN.*`, where `N` is the process number. The log files have the same format as in RH. The writing of each process's log file is buffered by line. Because these are updated often, when running with many processes this can be a drag on some systems. Therefore, it is possible to run full buffering (meaning log files are only written when the main program finishes). This option is not exposed in the configuration files, so one needs to change the file `parallel.c` in the following part:

```
/* _IOFBF for full buffering, _IOLBF for line buffering */
setvbuf(mpi.logfile, NULL, _IOLBF, BUFSIZ_MPILOG);
```

One should replace `_IOLBF` by `_IOFBF` to change from line buffering to full buffering.

The `output` directory will contain the three output files: `output_aux.hdf5`, `output_indata.hdf5`, and `output_ray.hdf5`. See *Output file structure* for more details on the structure of these files. If doing a re-run, these files must already exist; they will be updated with the new results. Otherwise, if these files are already in `output` before the execution, they will be overwritten. At the start of the execution, the output files are written with a special a fill value. This means that the disk space for the full output must be available at the start of the run, and no CPU time will be wasted if at the end of the run there is not enough disk space. The files are usually written every time a process finishes work on a given column. The fill value arrays are overwritten with the data. One advantage of this method is that even if the system crashes or the program stops, it is possible to recover the results already written (and a re-run can be performed for just the missing columns).

All the processes write asynchronously to all the output files. In some cases this can cause contention in the filesystem, with many processes trying to access the same data at the same time. In the worst case scenario, the contention can create bottlenecks which practically stop the execution. Therefore, it is highly recommended that the users tune their filesystem for the typical loads of RH. Many supercomputers make use of Lustre, a parallel filesystem. With Lustre, resources such as files can be divided in different stripes that can be placed in several different machines (OSTs). For running RH with more than 500 processes, one should use as many OSTs as available in the system, and select the lustre stripe size to the typical amount of data written to a file per simulation column. The stripe can set with the *lfs setstripe* command:

```
lfs setstripe -s stripe_size -c stripe_count -o stripe_offset directory|filename
```

It can be run per file (e.g. `output_ray.hdf5`), or for the whole `output` directory. Using a stripe count of `-1` will ensure that the maximum number of OSTs is used. For the typical files RH 1.5D produces, it is usually ok to apply the same Lustre settings to the whole `output` directory, and the following settings seem to reasonable:

```
lfs setstripe -s 4m -c -1 output/
```

Similarly, the `scratch` directory can also benefit from Lustre striping. Because most files there are small, it is recommended to use a stripe count of 1 for `scratch`.

Warning: You need a parallel filesystem if you are running RH 1.5D across more than one host. Most supercomputers have parallel filesystems, but if you are running in a smaller cluster this may not be the case. RH 1.5D will always run, but the HDF5 writes will not work and the results will be unreadable. NFS is **not** a parallel file system.

4.3 Logs and messages

In addition to the logs per process saved to `scratch`, a much smaller log will be printed in `stdout`. This log is a smaller summary of what each process is doing. Here is an example of typical messages:

```
Process   1: --- START task   1, (xi,yi) = (  0,156)
Process 232: --- START task   1, (xi,yi) = (  0,159)
Process  36: --- START task   1, (xi,yi) = (  0,162)
Process  12: --- START task   1, (xi,yi) = (  0,171)
(...)
Process  12: *** END   task   1 iter, iterations = 121, CONVERGED
Process   3: *** END   task   1 iter, iterations = 200, NO Convergence
Process   4: *** SKIP  task   1 (crashed after 81 iterations)
Process   3: --- START task   2, (xi,yi) = ( 23, 64)
Process  12: --- START task   2, (xi,yi) = ( 23, 65)
Process   4: --- START task   2, (xi,yi) = ( 23, 65)
(...)
*** Job ending. Total 262144 1-D columns: 262142 converged, 1 did not converge, 1␣
↪crashed.
*** RH finished gracefully.
```

In this example one can see the three possible outputs for a single-column calculation: convergence, non-convergence (meaning the target `ITER_LIMIT` was not met in `N_MAX_ITER` iterations), or a crash (many reasons). If there are singular matrices or other causes for a column to crash, RH 1.5D will skip that column and proceed to the next work unit. Such cases can be re-run with different parameters. In some cases (e.g. inexistent files) it is not possible to prevent a crash, and RH 1.5D will finish non-gracefully.

4.4 Reruns and lack of convergence

Dynamic atmospheres often have large gradients in temperature, density, velocity, etc. that cause problems when solving the non-LTE problem. This may lead to some (or all) columns not converging, and is dependent on the input atmosphere, model atoms, and run options. In RH terms, non-converged or "crashed", represent the same problem. In some cases, the iterations diverge strongly (crash), while in others they fail to reach the target limit for convergence in the allocated maximum number of iterations (non-convergence).

The output for atmosphere columns that did not converged or crashed is not saved to disk (a fill value is used instead). The recommended procedure in these cases is to rerun RH with different input options (e.g. less agressive acceleration, a larger number of maximum iterations). A special rerun mode is available to save time calculating again the columns that have already converged. When `15D_RERUN = TRUE` in `keyword.input`, RH will read the output and run again only for the columns that did not converge.

The rerun mode requires all the previous output to be under `output/`. The user has the possibility of changing options in `keyword.input` for the rerun. Not all options can be changed, because this could lead to non-sensical results (e.g. changing the input atmosphere or atom files). Only the following options can be changed in `keyword.input`:

15D_RERUN
15D_DEPTH_CUT
15D_TMAX_CUT
15D_DEPTH_REFINE
N_PESC_ITER
COLLRAD_SWITCH
COLLRAD_SWITCH_INIT
PRD_SWITCH
NRAYS
N_MAX_SCATTER
N_MAX_ITER
ITER_LIMIT
NG_DELAY
NG_ORDER
NG_PERIOD
S_INTERPOLATION
PRD_N_MAX_ITER
PRD_ITER_LIMIT
B_STRENGTH_CHAR

All the other options are locked to the values used for the firs run. Likewise, it is not possible to change `atoms.input`, the atom files, or the line list files. When RH 1.5D is first run, nearly all input options (except the input atmosphere and molecule files) are saved into the output, and in a rerun these are read from the output and not from the original files. This also means that a rerun can be performed even if the original files are no longer available.

4.5 Helper script

There is a Python script called `runtools.py` designed to make it easier to run RH 1.5D for large projects. It resides in `rh/python/runtools.py`. It requires Python with the numpy and h5py (or netCDF4) modules. It was made to run a given RH 1.5D setup over many simulation snapshots, spanning several atmosphere files. It supports a progressive rerun of a given problem, and allows the of use different `keyword.input` parameters for different columns, tackling columns harder to converge.

The first part of `runtools.py` should be modified for a users's need. It typically contains:

```
atmos_dir = '/mydata_dir'
seq_file = 'RH_SEQUENCE'
outsuff = 'output/output_ray_mysim_CaII_PRD_s%03i.hdf5'
mpicmd = 'mpiexec'
bin = './rh15d_ray_pool'
defkey = 'keyword.save'
log = 'rh_running.log'
tm = 40
rerun = True
rerun_opt = [ {'NG_DELAY': 60, 'NG_PERIOD': 40, '15D_DEPTH_REFINE': 'FALSE',
               '15D_ZCUT': 'TRUE', 'N_MAX_ITER': 250, 'PRD_SWITCH': 0.002 },
              {'NG_DELAY': 120, 'NG_PERIOD': 100, '15D_DEPTH_REFINE': 'TRUE',
               'PRD_SWITCH': 0.001 } ]
```

The different options are:

Name	Type	Description
atmos_dir	string	Directory where the atmosphere files are kept.
seq_file	string	Location of sequence file. This file contains the names of the atmosphere files to be used (one file per line). The script will then run RH 1.5D for every snapshot in every file listed.
outsuff	string	Template to write the output_ray.ncdf files. The %03i format will be replaced with the snapshot number.
mpicmd	string	System-dependent command to launch MPI. The script knows that for aprun the -B option should be used. This option also activates system specific routines (e.g. how to kill the run in pleiades).
bin	string	RH 1.5D binary to use.
defkey	string	Default template for keyword.input. Because the of the rerun options, keyword.input is overwritten for every rerun. This file is used as a template it (i.e., most of its options will be unchanged, unless specified in rerun_opt).
log	string	File where to save the main log. Will be overwritten for each new snapshot.
tm	int	Timeout (in minutes) to kill execution of code, if there is no message written to main log. Used to prevent code from hanging if there are system issues. After killed, program is relaunched. If tm = 0, program will never be killed.
rerun	bool	If True, will re-run the program (with different settings) to achieve convergence if any columns failed. Number of reruns is given by size of rerun_opt.
rerun_opt	list	Options for re-run. This is a list made of dictionaries. Each dictionary contains the keywords to update keyword.input. Only the keywords that differ from the defkey file are necessary.

Note: Only the first line of the sequence time is read at a time. The script reads the first line, deletes it from the file, and closes the file. It then reads the first line again and continues running, until there are no more lines in the file. This behaviour enables the file to be worked by multiple scripts at the same time, and allows one to dynamically change the task list at any time of the run.

Note: The script also includes a tenaciously persistent wait and relaunch feature designed to avoid corruption if there are system crashes or problems. Besides the tm timeout, if there is any problem with the execution, the code will wait for some periods and try and relaunch the code. For example, if one of the atmosphere files does not exist, runtools.py will try three times and then proceed to the next file.

CHAPTER 5

Analysis of output

5.1 Output file structure

The output is written to three files: `output_aux.hdf5`, `output_indata.hdf5`, and `output_ray.hdf5`. This is a big departure from RH, which contained several more output files. In particular, RH 1.5D will not write all the information that was written by RH, due to the sheer size it would take for large 3D simulations. The files are written in the machine-independent, self-describing HDF5 format. The contents of the files are organised in groups, variables, and attributes. Groups and variables can be imagined as directories and files in a filesystem. Inside groups, different variables and dimensions can be organised. The content of the output files can vary: some runs will have more detailed information and therefore more variables written to the files.

HDF5 is an open, platform-independent format, and therefore interfaces to many programming languages are available. The main interface libraries are available in C, C++, Fortran, and Java. But there are also interfaces for Python (h5py), Julia, IDL (from version 6.2), MATLAB , Octave, Perl, and R.

The RH 1.5D output format is standard HDF5 but it is also compatible with NetCDF 4 readers: in most cases one needs to specify only the variable or group name to read the data. The HDF5 and NetCDF libraries provide useful command line tools, which can be used to gather information about the RH 1.5D files or extract data. Additionally, there is a more complete set of tools written in Python to read and analyse these files.

Warning: Because of the limitations of different languages, not all interfaces support all HDF5 features. Some libraries (e.g. IDL or plain h5py in Python) will not detect missing data in arrays (written with the fill value). In such cases, reading variables with missing data (see *Output file structure*), the data are read with no warning or indication of those that have special fill values.

The structure of the three output files is given below.

Note: When a column fails to converge, output for that column is not written. This means that the variables that depend on `(nx, ny)` will have some values missing. HDF5 marks these values as *missing data* and uses a fill value (of 9.9692e+36). When the `15D_DEPTH_ZCUT` option is used, not all heights will be used in the calculation. The code does not read the skipped parts of the atmosphere. When writing such variables of `nz`, only the points that were used are written to the file, and the rest will be marked as missing data (typically the z cut height varies with the column).

5.1.1 output_aux.hdf5

This file contains the level populations and radiative rates. For each active atom or molecule, it contains different groups called `atom_XX` or `molecule_XX`, where `XX` is the identifier for the species (e.g. `MG`, `CO`).

Note: The atmosphere dimensions on many of the output files are not necessarily the same as in the atmosphere file. They depend on the number of columns calculated, which are a function of `X/Y_START/END/STEP`.

It has the following global attributes:

`atmosID`	Identifier for the atmosphere file.
`rev_id`	Revision identifier.
`nx`	Number of points in x dimension
`ny`	Number of points in y dimension
`nz`	Number of points in height dimension

Inside each of the atom/molecule groups, the following dimensions can exist:

Name	Description
`x`	Horizontal x dimension.
`y`	Horizontal y dimension.
`height`	Vertical dimension.
`level`	Number of atomic levels.
`line`	Number of atomic transitions
`continuum`	Number of bound-free transitions.
`vibration_level`	Number of molecule vibration levels.
`molecular_line`	Number of molecular lines.
`rotational_state`	Number of rotational states.

The atom groups can contain the following optional variables:

Name	Dimensions	Description
`populations`	`(level, x, y, height)`	Atomic populations.
`populations_LTE`	`(level, x, y, height)`	Atomic LTE populations.
`Rij_line`	`(line, x, y, height)`	Radiative rates out of the line.
`Rji_line`	`(line, x, y, height)`	Radiative rates into the line.
`Rij_continuum`	`(continuum, x, y, height)`	Radiative rates out of the bf transition.
`Rji_continuum`	`(continuum, x, y, height)`	Radiative rates into the bf transition.

The molecule groups can contain the following optional variables:

Name	Dimensions	Description
`populations`	`(vibration_level, x, y, height)`	Molecular populations.
`populations_LTE`	`(vibration_level, x, y, height)`	Molecular LTE populations.

All units are SI.

Note: In older versions it was possible to specify the keyword `15D_WRITE_EXTRA` and get additional output written to `output_aux.hdf5` (e.g. a new `opacity` group and more rates). While the procedures are still in `writeAux_p.c`, the functionality is deprecated because other changes in the code were not compatible with this way of writing the output. It is possible that this functionality will return at a later version.

5.1.2 output_indata.hdf5

This file contains data and metadata related to the run. It contains three groups: `input` (mostly settings from `keyword.input`), `atmos` (atmospheric variables), and `mpi` (several variables relating to the run).

It has the following global attributes:

`atmosID`	Identifier for the atmosphere file.
`rev_id`	Revision identifier.
`nx`	Number of points in x dimension
`ny`	Number of points in y dimension
`nz`	Number of points in height dimension

The `input` group contains all the input files (except atmosphere and molecular data), and a few attributes that are options from `keyword.input`. It contains the following string variables:

Variable	Description
`atom_groups`	Array with names of atom groups. Other is the same as atom order in `atoms.input`
`atoms_file_contents`	Contents of `atoms.input` saved into a string
`keyword_file_contents`	Contents of `keyword.input` saved into a string
`kurucz_file contents`	Contents of `kurucz.input` saved into a string, only if used

The `input` group also has other groups inside. If Kurucz line lists are used, it contains groups called `Kurucz_line_file0`, ..., `Kurucz_line_fileN`, where N-1 is the total number of line list files. The other groups are all atom files (`PASSIVE` and `ACTIVE`), and they take the names of `atom_XX`, where `XX` is the element name (for a list of these, see the variable `atom_groups` above). Inside all of these groups (Kurucz and atom) there is one variable, called `file_contents`, which contains the file saved intro a string and an attribute, called `file_name`, which contains the file name and path. These input options and files are read instead of the original files when doing a rerun.

The `atmos` groups contains the dimensions `x`, `y`, `height`, `element` and `ray`. It also contains the following variables:

Name	Dimensions	Units	Description
`temperature`	`(x, y, height)`	K	Temperatures
`velocity_z`	`(x, y, height)`	m s^{-1}	Vertical velocities
`electron_density`	`(x, y, height)`	m s^{-3}	Electron densities
`height_scale`	`(x, y, height)`	m	Height scale used. Can be different for every column when depth refine is used.
`element_weight`	`(element)`	a.m.u.	Atomic weights
`element_abundance`	`(element)`		Element abundances relative to hydrogen.
`muz`	`(ray)`		mu values for each ray.
`muz`	`(ray)`		mu weights for each ray.
`x`	`(x)`	m	Spatial coordinates along x axis.
`y`	`(y)`	m	Spatial coordinates along y axis.

Note: When `15D_DEPTH_REFINE` is used, each column will have a different (optimised) height scale, but they all have the same number of depth points (`nz`). In these cases, it is very important to save the `height` variable because otherwise one does not know how to relate the height relations of quantities from different columns.

The `atmos` group also contains the following attributes:

moving	Unsigned int, 1 if velocity fields present.
stokes	Unsigned int, 1 if stokes output present.

The mpi group contains the dimensions x, y, and iteration (maximum number of iterations).

Warning: iteration is currently hardcoded in the code to a maximum of 1500. If you try to run more than 1500 iterations, there will be an error writing to the output.

The mpi group also contains several variables:

Name	Dimensions	Description
xnum	(x)	Indices of x positions calculated.
xnum	(x)	Indices of x positions calculated.
task_map	(x, y)	Maps which process ran which column.
task_map_number	(x, y)	Maps which task number each column was.
iterations	(x, y)	Number of iterations used for each column.
convergence	(x, y)	Indicates if each column converged or not. Possible values are 1 (converged), 0 (non converged), or −1 (crashed).
delta_max	(x, y)	Final value for delta_max when iteration finished.
delta_max_history	(x, y, iteration)	Evolution of delta_max
z_cut	(x, y)	Height index of the temperature cut.

The mpi group also contains the following attributes: x_start, x_end, x_step, y_start, y_end, and y_step, all of which are options from keyword.input.

5.1.3 output_ray.hdf5

This file contains the synthetic spectra and can also contain extra information such as opacities and the source function. It contains only the root group. Its dimensions are x, y, wavelength, and eventually wavelength_selected and height. The latter two are only present when ray.input specifies more than 0 wavelengths for detailed output, and it matches Nsource, the number of those wavelengths entered in ray.input.

It can contain the following variables:

Name Description	Dimensions	U
wavelength Wavelength scale.	(wavelength)	n
intensity Synthetic disk-centre intensity (Stokes I).	(x, y, wavelength)	W
stokes_Q Stokes Q. **Optional.**	(x, y, wavelength)	W
stokes_U Stokes U. **Optional.**	(x, y, wavelength)	W
stokes_V Stokes V. **Optional.**	(x, y, wavelength)	W
tau_one_height Height where optical depth reaches unity, for each column. **Optional.**	(x, y, wavelength)	m
wavelength_selected Wavelength scale for the detailed output variables below. **Optional.**	(wavelength_selected)	
wavelength_indices Indices of wavelengths selected for variables below. **Optional.**	(wavelength_selected)	
chi Total opacity (line and continuum). **Optional.**	(x, y, height, wavelength_selected)	m
source_function Total opacity (line and continuum). **Optional.**	(x, y, height, wavelength_selected)	W
Jlambda Angle-averaged radiation field. **Optional.**	(x, y, height, wavelength_selected)	W
scattering Scattering term multiplied by Jlambda. **Optional.**	(x, y, height, wavelength_selected)	

The wavelength is in nm, air or vacuum units, depending if VACUUM_TO_AIR is TRUE or FALSE (in keyword.input). chi is in $m^{-1}$and tau_one_height in m.

Despite internally being calculated in double precision, all the output (except the wavelength scale) is written in single precision to save disk space.

The full Stokes vector is only written when in keyword.input STOKES_MODE is not NO_STOKES and the STOKES_INPUT is set.

The chi, source_function, and Jlambda variables depend on the 3D grid and on wavelength. Therefore, for even moderate grid sizes they can take huge amounts of space. If nx = ny = nz = 512 and wavelength_selected = 200, each of these variables will need 100Gb of disk space. For a simulation with a cubic grid of 1024^3 points and saving the full output for 1000 wavelength points, output_ray.hdf5 will occupy a whopping 12Tb per snapshot of disk space. To avoid such problems, these large arrays are only written when ray.input contains Nsource > 0, and for the wavelengths selected.

The output_ray.hdf5 file contains the following global attributes:

atmosID	Identifier for the atmosphere file
snapshot_number	Number of simulation snapshot (from atmosphere file)
rev_id	Revision identifier
nx	Number of points in x dimension
ny	Number of points in y dimension
nz	Number of points in height dimension
nwave	Number of wavelength points
wavelength_selected	Number of wavelength points selected for detailed output
creation_time	Local time when file was created

5.2 Command line tools

Two useful command line tools that come with HDF5 are h5dump and h5repack.

`h5dump` can be used with the `-H` option to look at the header of a file: see the dimensions, variables, groups. It can also be used to print a text version of any variable in an HDF5 file (e.g. this can be redirected to a text file). When printing a variable (dataset in HDF5) one uses the option `-d variable`, and the resulting output is the same as in the `-H` mode, with the variable printed at the end. The NetCDF ncdump program offers an even clearer look into the file (e.g. used with the `-h` option to print out the header).

The `h5repack` program can be used to copy and modify the parameters of HDF5 files. It can convert the files between different format versions, compress variables, etc. Of particular importance is the option for rechunking a file. Chunking in HDF5 files can be used to improve performance by changing the disk structures to improve different read patterns. It is analogous to fully or partially transposing the variables along certain dimensions.

See also:

h5dump guide Detailed information about `h5dump`.

h5repack guide Detailed information about `h5repack`.

Chunking in HDF5 Description on the advantages of chunking.

5.3 Reading output in Python

The helita package has a complete python interface to read the output, input, and visualise files from RH 1.5D. The `helita tools are described in detail in section *helita interface*.

If `helita is not available, the easiest and fastest way to read the RH 1.5D output (or input) files in Python is via the xarray package. `xarray` can load the output files as a dataset directly, but in the case of the `output_aux.hdf5` and `output_indata.hdf5` one needs to specify which group to read (see above).

Here is a quick example on how to read some output from RH 1.5D with `xarray`:

```
>>> import xarray
>>> ray = xarray.open_dataset("output_ray.hdf5")
>>> ray
<xarray.Dataset>
Dimensions:               (height: 82, wavelength: 902, wavelength_selected: 10, x:␣
↪1, y: 1)
Coordinates:
  * wavelength            (wavelength) float64 28.0 31.4 32.8 33.7 34.3 35.3 ...
  * wavelength_selected   (wavelength_selected) float64 85.1 276.4 278.5
  * x                     (x) float64 0.0
  * y                     (y) float64 0.0
Dimensions without coordinates: height
Data variables:
    Jlambda               (x, y, height, wavelength_selected) float64 ...
    chi                   (x, y, height, wavelength_selected) float64 ...
    intensity             (x, y, wavelength) float64 ...
    scattering            (x, y, height, wavelength_selected) float64 ...
    source_function       (x, y, height, wavelength_selected) float64 ...
    wavelength_indices    (wavelength_selected) int32 ...
Attributes:
    atmosID:               FALC_82_5x5.hdf5 (Wed Jan 10 15:29:28 2018)
    snapshot_number:       0
    rev_id:                001d537  Tiago Pereira  2018-01-10 12:34:07 +0100
    nx:                    1
    ny:                    1
    nz:                    82
    nwave:                 902
```

(continues on next page)

(continued from previous page)

```
    wavelength_selected:  3
    creation_time:        2018-01-10T16:16:42+0100
>>> aux = xarray.open_dataset("output_aux.hdf5", group="atom_MG")
>>> aux
<xarray.Dataset>
Dimensions:          (continuum: 10, height: 82, level: 11, line: 15, x: 1, y: 1)
Coordinates:
  * x                (x) float64 0.0
  * y                (y) float64 0.0
Dimensions without coordinates: continuum, height, level, line
Data variables:
    Rij_continuum    (continuum, x, y, height) float64 ...
    Rij_line         (line, x, y, height) float64 ...
    Rji_continuum    (continuum, x, y, height) float64 ...
    Rji_line         (line, x, y, height) float64 ...
    populations      (level, x, y, height) float64 ...
    populations_LTE  (level, x, y, height) float64 ...
Attributes:
    nlevel:      11
    nline:       15
    ncontinuum:  10
```

5.4 Reading output in IDL

There are no specific IDL routines for reading the output from RH 1.5D. However, there is a utility function that can be used to variables from HDF5/netCDF4 files, under the `idl/` directory in a file named `read_ncdf_var.pro`. The function `read_ncdf_var()` can be used to read variables from an HDF5 or netCDF4 file, e.g.:

```
IDL> data = read_ncdf_var("output_ray.hdf5", "intensity")
IDL> help, data
DATA            FLOAT     = Array[902, 512, 512]
IDL> pops = read_ncdf_var("output_aux.hdf5", "populations", groupname="atom_CA")
IDL> help, pops
POPS            FLOAT     = Array[400, 512, 512, 5]
```

Note: The IDL analysis suite of RH **does not work with RH 1.5D**.

CHAPTER 6

helita interface

The helita Python package contains several routines to interface with RH 1.5D. Installation instructions are available in its website.

6.1 Reading and writing input files

6.1.1 Writing atmosphere files

The rh15d module in helita.sim contains a function to write an input atmosphere in RH 1.5D format, assuming the user already has the required data to write at hand. Its function definition is:

```
def make_xarray_atmos(outfile, T, vz, z, nH=None, x=None, y=None, Bz=None, By=None,
                      Bx=None, rho=None, ne=None, vx=None, vy=None, vturb=None,
                      desc=None, snap=None, boundary=None, append=False):
    """
    Creates HDF5 input file for RH 1.5D using xarray.

    Parameters
    ----------
    outfile : string
        Name of destination. If file exists it will be wiped.
    T : n-D array
        Temperature in K. Its shape will determine the output
        dimensions. Shape is generally (nt, nx, ny, nz), but any
        dimensions except nz can be omitted. Therefore the array can
        be 1D, 2D, or 3D, 4D but ultimately will always be saved as 4D.
    vz : n-D array
        Line of sight velocity in m/s. Same shape as T.
    z : n-D array
        Height in m. Can have same shape as T (different height scale
        for each column) or be only 1D (same height for all columns).
    nH : n-D array, optional
        Hydrogen populations in m^-3. Shape is (nt, nhydr, nx, ny, nz),
        where nt, nx, ny can be omitted but must be consistent with
        the shape of T. nhydr can be 1 (total number of protons) or
        more (level populations). If nH is not given, rho must be given!
    ne : n-D array, optional
```

(continues on next page)

(continued from previous page)

```
        Electron density in m^-3. Same shape as T.
    rho : n-D array, optional
        Density in kg m^-3. Same shape as T. Only used if nH is not given.
    vx : n-D array, optional
        x velocity in m/s. Same shape as T. Not in use by RH 1.5D.
    vy : n-D array, optional
        y velocity in m/s. Same shape as T. Not in use by RH 1.5D.
    vturb : n-D array, optional
        Turbulent velocity (Microturbulence) in km/s. Not usually needed
        for MHD models, and should only be used when a depth dependent
        microturbulence is needed (constant microturbulence can be added
        in RH).
    Bx : n-D array, optional
        Magnetic field in x dimension, in Tesla. Same shape as T.
    By : n-D array, optional
        Magnetic field in y dimension, in Tesla. Same shape as T.
    Bz : n-D array, optional
        Magnetic field in z dimension, in Tesla. Same shape as T.
    x : 1-D array, optional
        Grid distances in m. Same shape as first index of T.
    y : 1-D array, optional
        Grid distances in m. Same shape as second index of T.
    x : 1-D array, optional
        Grid distances in m. Same shape as first index of T.
    snap : array-like, optional
        Snapshot number(s).
    desc : string, optional
        Description of file
    boundary : Tuple, optional
        Tuple with [bottom, top] boundary conditions. Options are:
        0: Zero, 1: Thermalised, 2: Reflective.
    append : boolean, optional
        If True, will append to existing file (if any).
    """
```

Note that while in this routine the writing of the hydrogen populations is optional (they can be derived from the mass density, if available), RH 1.5D does not support this yet.

Note: The variables passed to `make_xarray_atmos` must be consistent with the height scale. The first height index must be the top of the atmosphere (closest to observer), and the height scale must be strictly decreasing.

6.1.2 Reading atmosphere files

Once written, the input atmosphere files can be read in Python with `xarray`, and do not require `helita`. For example:

```
>>> import xarray
>>> atmos = xarray.open_dataset('my_atmos.hdf5')
>>> atmos
<xarray.Dataset>
Dimensions:                (depth: 82, nhydr: 6, snapshot_number: 1, x: 5, y: 5)
Coordinates:
  * x                      (x) int64 0 1 2 3 4
  * y                      (y) int64 0 1 2 3 4
    z                      (snapshot_number, depth) float32 ...
  * snapshot_number        (snapshot_number) int32 0
Dimensions without coordinates: depth, nhydr
Data variables:
```

(continues on next page)

(continued from previous page)

```
    temperature           (snapshot_number, x, y, depth) float32 ...
    velocity_z            (snapshot_number, x, y, depth) float32 ...
    electron_density      (snapshot_number, x, y, depth) float64 ...
    hydrogen_populations  (snapshot_number, nhydr, x, y, depth) float32 ...
    velocity_turbulent    (snapshot_number, x, y, depth) float32 ...
Attributes:
    comment:          Created with make_xarray_atmos on 2018-01-25 15:28:10.4...
    boundary_top:     0
    boundary_bottom:  1
    has_B:            0
    description:      FAL C model with 82 depth points replicated to 5x5 colu...
    nx:               5
    ny:               5
    nz:               82
    nt:               1
```

The amount of detail loaded by `xarray` will depend how the atmosphere was written. Older atmosphere files may not have as much verbose attributes or labeled coordinates (especially if written by plain HDF5 with no attaching of dimension scales), but they are still valid. Older netCDF atmospheres should work fine with `xarray`.

It is also possible to modify the data with `xarray`, and saving and updated atmosphere is done via the `to_netcdf()` method:

```
>>> atmos.to_netcdf("newfile.hdf5", format='NETCDF4')
```

Be sure to use `format='NETCDF4'` so that the file is internally HDF5!

6.1.3 Writing wavelength files

Another utility function in `rh15d.py` is `make_wave_file`. This creates an RH wavelength file (to be used with the option `WAVETABLE` in `keyword.input`) that contains additional wavelengths to be calculated. The function's usage is documented in its function call:

```
def make_wave_file(outfile, start=None, end=None, step=None, new_wave=None,
                   ewave=None, air=True):
    """
    Writes RH wave file (in xdr format). All wavelengths should be in nm.

    Parameters
    ----------
    start: number
        Starting wavelength.
    end: number
        Ending wavelength (non-inclusive)
    step: number
        Wavelength separation
    outfile: string
        Name of file to write.
    ewave: 1-D array, optional
        Array of existing wavelengths. Program will make discard points
        to make sure no step is enforced using these points too.
    air: boolean, optional
        If true, will at the end convert the wavelengths into vacuum
        wavelengths.
    """
```

You can either supply an array with the wavelengths, or give a range of wavelengths and a fixed spacing, e.g.:

```
>>> from helita.sim import rh15d
>>> rr = rh15d.Rh15dout()
```

(continues on next page)

(continued from previous page)

```
# this will write wavelenghts from 650 to 650 nm, 0.01 nm spacing:
>>> rh15d.make_wave_file('my.wave', 650, 660, 0.01)
# this will write an existing array "my_waves", if it exists
>>> rh15d.make_wave_file('my.wave', ewave=my_waves)
```

6.2 Reading output files

The main class to read the output is called `Rh15dout`. It uses `xarray` under the hood and populates an object with all the different datasets. It can be initiated in the following way:

```
>>> from helita.sim import rh15d
>>> rr = rh15d.Rh15dout()
--- Read ./output_aux.hdf5 file.
--- Read ./output_indata.hdf5 file.
--- Read ./output_ray.hdf5 file.
```

By default, it will look for the three files in the directory specified as main argument (defaults to current directory). Additionally, the method `read_group(infile)` can be used to manually load the `output_aux.hdf5` or `output_indata.hdf5` and the method and `read_ray(infile)` can be used to manually load the `output_ray.hdf5` file. The variables themselves are not read into memory, but are rather a memmap object (file pointer; only read when needed) that `xarray` opens.

After loading the files, the `Rh15dout` instance loads each file as an `xarray` dataset with the base name of each group (e.g. `ray`, `atmos`, `atom_CA`, `mpi`).The `ray` attribute contains the same dataset as shown in the `xarray` example above.

The attributes of each file are still accessible under the attributes of each object, e.g.:

```
>>> rr.ray.creation_time
'2018-01-10T16:16:42+0100'
>>> rr.atmos.nrays
5
>>> rr.mpi.nprocesses
2048
```

With `xarray` it is easy to quickly inspect and plot different quantities. For example, to plot the intensity at `(x, y) = (0, 0)`:

```
>>> rr.ray.intensity[0, 0].plot()
```

Or the intensity at a fixed wavelength:

```
>>> rr.ray.intensity.sel(wavelength=279.55, method='nearest').plot()
```

(This only shows a 2D image if you calculated the intensity from a 3D model, otherwise an histogram or line plot is shown.)

6.3 Visualisation and notebooks

`helita` includes a visualisation module, `helita.sim.rh15d_vis`, with widgets that are meant to be used inside the Jupyter notebook. To use these, you will need to install not only `helita` but also the Matplotlib Jupyter Extension and the IPython widgets for Jupyter. If you have Anaconda, both can be installed with conda:

```
conda install -c conda-forge ipywidgets ipympl widgetsnbextension
jupyter nbextension enable --py widgetsnbextension
```

You can also install them with `pip` (check their pages for details).

Currently we have the following Jupyter notebooks for visualisation of RH 1.5D output:

- Basic output
- Visualisation widgets

To use the above notebooks, you need to have run RH 1.5D and have the output files ready!

You can also explore the input atmosphere files with the Jupyter widget `rh15d_vis.InputAtmosphere` in `helita`:

```
>>> from helita.sim import rh15d_vis
>>> rh15d_vis.InputAtmosphere('my_atmos.hdf5');
```

CHAPTER 7

Known bugs and limitations

RH 1.5D is always evolving, and there are likely to be bugs and limitations. Please send all bug reports to tiago.pereira-at-astro.uio.no and they will be dealt with as time permits.

7.1 Current issues

- Check the github RH issues page for an updated list.
- If the `scratch` or `output` directories are not present, the code will crash. The error message is not very clear.
- In `keyword.input`, if one sets a `SNAPSHOT` value to be more than what is in the atmosphere file, the code will stop with an error message: `Index exceeds dimension bound`. This error should be made more clear.
- The atom files must not end with a blank line, otherwise `gencol` will fail and the program stops.
- Line buffered or full buffered log options still require the user to change the source code.
- Depth refinement fails in some cases due to problems caused by cubic interpolation artefacts.
- Using more than 4000 cores and writing full output may cause I/O slowdowns and Lustre contention in some systems.

7.2 Planned features

- Support for multiple snapshots in the output files.
- *pool* mode be more flexible, with the possibility of several *overlord* nodes, useful for running with more than 4000 processes.
- More flexible control of what output is written.