RH 1.5D Documentation

Release r79

Tiago M. D. Pereira

Apr 24, 2017

Contents

Contents:

CHAPTER 1

Introduction

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 4096 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 complete modified version of RH (see below for differences from RH distributed by Han Uitenbroek).

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.

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:

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)

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 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

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

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 (only for contributors):

> 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_mpi	Source files for RH 1.5D
rh/rhf1d	Source files for 1D geometry
rh/rhsc2d	Source files for 2D geometry, not used
rh/rhsc3d	Source files for 3D geometry, not used
rh/rhsphere	Source files for spherical geometry, not used
rh/tools	Associate C programs for RH, not tested.

Dependencies

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 (versions from branch 1.10.x do not currently work).

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 Hexagon, they can be loaded as:

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

in Pleiades:

```
module load hdf5/1.8.7/gcc/mpt
```

and at ITA's Linux system:

```
module load hdf5/Intel/openmpi/1.8.18
```

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.

Main libraries

First, one needs to set the environment variables `OS` and `CPU`:

```
export CPU=`uname -m`
export OS=`uname -s`
```

Note: All the shell commands given in this manual are for bash, so you'll have to modify them if using another shell.

The main Makefile will then look for an architecture-dependent Makefile in `rh/makefile.$CPU.$OS`. If a Makefile for your system combination does not exist, you'll have to create a new Makefile and adapt it to your configuration. You need to make sure that the architecture-dependent Makefile reflects your system's configuration (i.e., compiler names and flags).

After setting the correct compiler, just 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`.

Program binaries

Go to the `rh/rh15d_mpi/` directory and update the Makefile with your compiler and flags. You will need to set `CC` to the MPI alias (e.g. `mpicc`). The path to the HDF5 library needs to be explicitly set in `HDF5_DIR`. In Hexagon this is already stored in the `HDF5_DIR` environment variable.

If your version of HDF5 was not built as a shared binary, you need to link HDF5 and other used libraries directly. Set the LDFLAGS accordingly, and update the LIBS variable to contain all the other libraries. For Pleiades, make sure your Makefile contains the following:

```
OTHER_LIBRARY_DIR = /path/to/library
HDF5_DIR = /path/to/hdf5
LDFLAGS = -L../ -L$(OTHER_LIBRARY_DIR)/lib/  -L$(HDF5_DIR)/lib/
LIBS = -lrh -lrh_f90 $(ARCHLIBS) -lhdf5_hl -lhdf5 -lz -lm
```

Once the Makefile is updated, compilation is achieved 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

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.

The run directory can be located anywhere, but it **must** have a directory called Atoms two levels below (i.e. ../../Atoms/) with the Barklem_*data.dat files. This is because these relative paths are hardcoded in barklem.c. The input files in the run directory must obviously point to existing path names.

CHAPTER 3

Input files

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	Description
`SNAPSHOT`	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`	How many columns to sample in the y direction. If < 1, will be set to 1.
`Y_START`	Starting column in the y direction. If < 0, will be set to 0.
`Y_END`	Ending column in the y direction. If <= 0, will be set to `NY` in the atmosphere file.
`Y_STEP`	How many columns to sample in the y direction. If < 1, will be set to 1.
`15D_WRITE_POPS`	If `TRUE`, will write the level populations (including LTE) for each active atom to `output_aux.hdf5`.
`15D_WRITE_RRATES`	If `TRUE`, will write the radiative rates (lines and continua) for each active atom to `output_aux.hdf5`.
`15D_WRITE_TAU1`	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`	If `TRUE`, will rerun for non-converged columns.
`15D_DEPTH_ZCUT`	If `TRUE`, will perform a cut in z for points above a threshold temperature
`15D_TMAX_CUT`	Threshold temperature (in K) over which the above depth cut ids made. If < 0, no temperature cut will be made.
`15D_DEPTH_REFINE`	If `TRUE`, will perform an optimisation of the depth scale, based on optical depth, density and temperature gradients.
`BACKGR_IN_MEM`	If `TRUE`, will keep background opacity coefficients in memory instead of scratch files on disk.
`COLLRAD_SWITCH`	Defines if collisional radiative switching is on. If < 0, switching parameter is constant (and equal to `COLLRAD_SWITCH_INI`). I
`COLLRAD_SWITCH_INIT`	Initial increment for collisional-radiative switching
`LIMIT_MEMORY`	If `TRUE`, will not keep several large arrays in memory but rather save them to scratch files. Not recommended unless memory usa
`N_PESC_ITER`	Number of escape probability iterations, if any atoms have it as initial solution.
`PRD_SWITCH`	If > 0, the PRD effects will be added gradually, converging to the full PRD solution in `1/sqrt(PRD_SWITCH)` iterations.
`PRDH_LIMIT_MEM`	If `TRUE` and using `PRD_ANGLE_APPROX`, will not keep in memory quantities necessary to calculate the current PRD weights, b
`S_INTERPOLATION`	Type of source function interpolation to use in formal solver. Can be `LINEAR`, `BEZIER`, `BEZIER3` or `CUBIC_HERMITE`.
`S_INTERPOLATION_STOKES`	Type of source function interpolation to use in formal solver for polarised cases. Can be `DELO_PARABOLIC` or `DELO_BEZIER`
`VTURB_MULTIPLIER`	Atmospheric `vturb` will be multiplied by this value
`VTURB_ADD`	Value to be added to atmospheric `vturb`

[1] de la Cruz Rodríguez, J.; Piskunov, N. 2013, ApJ, 764, 33, ADS link.

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`).

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:

[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)**

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.

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 x 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.
`z`	`(nt, nz)`	m	Height grid. Can be different for each snapshot, hence the *nt* dependence.

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
      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.

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

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.

The 'normal' mode is deprecated and will be removed in a later revision. Use only 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
```

The run directory

Warning: Before running, make sure you have the sub-directories `scratch` and `output` in the run directory **and** that two levels below there is an `Atoms` directory (`../../Atoms/`).

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` 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. The way that HDF5 work means that these files are created and filled with special (masked) values at the start of the execution. 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 masked values 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`.

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.

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 re-run 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

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.

The structure of the three 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).

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`).

The dimensions of the root group are `nx`, `ny`, and `nz`.

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`.

There are two global attributes:

`atmosID`	Identifier for the atmosphere file.
`rev_id`	Revision identifier.

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

Name	Description
`nlevel`	Number of atomic levels.
`nline`	Number of atomic transitions
`ncontinuum`	Number of bound-free transitions.
`nlevel_vibr`	Number of molecule vibration levels.
`nline_molecule`	Number of molecular lines.
`nJ`	Number of rotational states.

The atom groups contain the following variables:

The molecule groups contain the following variables:

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.

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).

The dimensions of the root group are `nx`, `ny`, `nz`, and `strlen` (used as maximum length for string type variables).

There are two global attributes:

`atmosID`	Identifier for the atmosphere file.
`rev_id`	Revision identifier.

The `input` group contains only attributes, all options from `keyword.input`.

The `atmos` groups contains the dimensions `nhydr`, `elements` and `nrays`. It also contains the following variables:

Name	Dimensions	Units	Description
`temperature`	`(nx, ny, nz)`	K	Temperatures
`velocity_z`	`(nx, ny, nz)`	m s^{-1}	Vertical velocities
`height`	`(nx, ny, nz)`	m	Height scale used. Can be different for every column when depth refine is used.
`element_weight`	`(nelements)`	a.m.u.	Atomic weights
`element_abundance`	`(nelements)`		Log of element abundances relative to hydrogen (A(H) = 12).
`element_id`	`(nelements, strlen)`		Element identifiers.
`muz`	`(nrays)`		mu values for each ray.
`muz`	`(nrays)`		mu weights for each ray.
`x`	`(nx)`	m	Spatial coordinates along x axis.
`y`	`(ny)`	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 `nprocesses` (number of processes) and `niterations` (maximum value of iterations).

> **Warning:** `niterations` 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`	`(nx)`	Indices of x positions calculated.
`xnum`	`(nx)`	Indices of x positions calculated.
`task_map`	`(nx, ny)`	Maps which process ran which column.
`task_map_number`	`(nx, ny)`	Maps which task number each column was.
`iterations`	`(nx, ny)`	Number of iterations used for each column.
`convergence`	`(nx, ny)`	Indicates if each column converged or not. Possible values are `1` (converged), `0` (non converged), or `-1` (crashed).
`delta_max`	`(nx, ny)`	Final value for `delta_max` when iteration finished.
`delta_max_history`	`(nx, ny, niterations)`	Evolution of `delta_max`
`ntasks`	`(nprocesses)`	Number of tasks assigned to each process. Does **not** work in *pool* mode.
`hostname`	`(nprocesses, strlen)`	Hostname of each process.
`starting_time`	`(nprocesses, strlen)`	Time when each process started the calculations.
`finish_time`	`(nprocesses, strlen)`	Time when each process finished the calculations.
`z_cut`	`(nx, ny)`	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`.

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 `nx`, `ny`, `nwave`, and eventually `wavelength_selected`. The latter is 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	Dimensions	Description
`wavelength`	`(nwave)`	Wavelength scale.
`intensity`	`(nx, ny, nwave)`	Synthetic disk-centre intensity (Stokes I).
`stokes_Q`	`(nx, ny, nwave)`	Stokes Q. **Optional.**
`stokes_U`	`(nx, ny, nwave)`	Stokes U. **Optional.**
`stokes_V`	`(nx, ny, nwave)`	Stokes V. **Optional.**
`tau_one_height`	`(nx, ny, nwave)`	Height where optical depth reaches unity, for each column. **Optional.**
`wavelength_indices`	`(wavelength_selected)`	Indices of wavelengths selected for variables below. **Optional.**
`chi`	`(nx, ny, nz, wavelength_selected)`	Total opacity (line and continuum). **Optional.**
`source_function`	`(nx, ny, nz, wavelength_selected)`	Total opacity (line and continuum). **Optional.**
`Jlambda`	`(nx, ny, nz, wavelength_selected)`	Angle-averaged radiation field. **Optional.**

All units are in SI. The `intensity`, Stokes vector, `source_function`, `Jlambda` are all in J s^{-1}m^{-2}Hz^{-1}sr^{-1}. 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.

Note: As noted above, arrays of `nz` will have the first values missing if `15D_DEPTH_ZCUT` is used. The variables `chi`, `source_function`, and `Jlambda` are an exception to this rule. For performance reasons these missing values are not marked with a fill value, but instead they are filled with zeros.

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
`creation_time`	Local time when file was created

Reading the output 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 files can be read with standard HDF5 or NetCDF 4 readers: in most cases one needs to specify only the variable or group name. The HDF5 library provides 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. Interfaces in other languages are also planned.

Warning: Because of the limitations of different languages, not all interfaces support all HDF5 features. IDL in particular does not support masked arrays. This means that when reading variables with missing data (see *Output file structure*), IDL will happily read all the data with no warning or indication of those that have special fill values.

Command line tools

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

As shown above, `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 one uses the option `-v variable`, and the resulting output is the same as in the `-H` mode, with the variable printed at the end.

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.

Python interface

Python routines to read the input, output, and more are available at `rh/python/rh15d.py`. They require the numpy and h5py (or netCDF4) modules.

Reading output files

The main class to read the output is called `Rh15dout`. It can be initiated in the following way:

```
>>> from rh15d import *
>>> 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 methods `read_aux(infile)`, `read_indata(infile)`, and `read_ray(infile)` can be used to manually load a file. The variables themselves are not read into memory, but are rather a memmap object (file pointer; only read when needed).

After loading the files, the `Rh15dout` instance maps them into different classes. The `ray` class contains all the variables that were saved in the `output_ray.hdf5` as attributes:

```
>>> [a for a in dir(rr.ray) if a[0] != '_']
['intensity',
 'params',
 'stokes_Q',
 'stokes_U',
 'stokes_V',
 'chi',
 'source_function',
 'scattering',
 'Jlambda',
 'tau_one_height',
 'wavelength',
 'wavelength_indices']
>>> rr.ray.intensity
<HDF5 dataset "intensity": shape (256, 256, 1626), type "<f4">
```

Several of the options from `output_indata.hdf5`, along with many of the dimensions used, are gathered into a method called `params`, saved as a dictionary. The `mpi` and `atmos` groups of `output_indata.hdf5`, with all their variables are like `ray`, added into classes with the same name that are methods of the `Rh15dout` instance. For example:

```
>>> rr.mpi.convergence
<HDF5 dataset "convergence": shape (256, 256), type "<i8">
>>> rr.atmos.temperature
<HDF5 dataset "temperature": shape (256, 256, 400), type "<f4">
```

Likewise, the groups of `output_aux.hdf5` are also added as classes that are methods of the main instance:

```
>>> [a for a in dir(rr.atom_CA) if a[0] != '_']
['Rij_continuum',
 'Rij_line',
 'Rji_continuum',
 'Rji_line',
 'params',
 'populations',
 'populations_LTE']
```

All the groups and variables are therefore automatically added, so they will adapt to any changes in the output files.

Reading input files

The `HDF5Atmos` class can be used to read the input atmosphere files. It can be initiated in the following way:

```
>>> from rh15d import *
>>> atm = rh15d.HDF5Atmos('my_atmos.hdf5')
```

Inspection of the result reveals the variables, a dictionary called `param` with basic data and the properties of the variables:

```
>>> [a for a in dir(atm) if a[0] != '_']
['B_x',
 'B_y',
 'B_z',
 'close',
 'closed',
 'electron_density',
 'file',
 'hydrogen_populations',
 'params',
 'read',
 'snapshot_number',
 'temperature',
 'velocity_z',
 'write_multi',
 'write_multi_3d',
 'x',
 'y',
 'z']
>>> atm.params      # Contains basic properties
```

```
{'boundary_bottom': 1,
 'boundary_top': 0,
 'description': 'Created with make_hdf5_atmos.on 2017-04-03 17:32:06.831811',
 'has_B': 1,
 'nhydr': 6,
 'nt' : 4,
 'nx': 512,
 'ny': 512,
 'nz': 400}
>>> atm.velocity_z
<HDF5 dataset "velocity_z": shape (4, 512, 512, 400), type "<f4">
```

This interface is read-only, no modifications to the atmosphere files are possible.

Writing input files

In `rh15d.py` there is also a function to write the input atmosphere, assuming the user already has the required arrays at hand. Its function definition is:

```
def make_hdf5_atmos(outfile, T, vz, nH, z, x=None, y=None, Bz=None, By=None,
                    Bx=None, rho=None, ne=None, vx=None, vy=None, desc=None,
                    snap=None, boundary=[1, 0], comp=None, complev=None,
                    append=False):
    """
    Creates HDF5 input file for RH 1.5D.

    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 (can be 1D, 2D, or 3D).
    vz : n-D array
        Line of sight velocity in m/s. Same shape as T.
    nH : n-D array
        Hydrogen populations in m^-3. Shape is [nhydr, shape.T] where
        nydr can be 1 (total number of protons) or more (level populations).
    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).
    ne : n-D array, optional
        Electron density in m^-3. Same shape as T.
    rho : n-D array, optional
        Density in kg / m^-3. Same shape as T.
    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.
    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).
    comp : string, optional
        Options are: None (default), 'gzip', 'szip', 'lzf'.
    complev : integer or tuple, optional
        Compression level. Integer for 'gzip', 2-tuple for szip.
    """
```

Both zlib and szip compression are supported but again this is not recommended.

Writing wavelength files

Another utility function in `rh15d.py` is `make_wave_file`. It creates an RH wavelength file from a given array of wavelengths. It'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.
    """
```

Other languages

While many other languages have interfaces to read HDF5 files, there are no specific routines for reading the output from RH 1.5D. Support for other languages may be added later as demand requires.

Under RH 1.5D `idl/` directory is routine 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]
```

Analysis tools

Note: There is no organised package of analysis tools for the output of RH 1.5D. This should be added in the future. The IDL analysis suite of RH **does not work with RH 1.5D**.

CHAPTER 6

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.

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.

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.

CHAPTER 7

Appendices

A. Compiling netCDF

By far the easiest way to use RH is when the proper HDF5 library is already installed in the system. This is the case at most supercomputers, but not usually in user workstations or laptops. Even if your operative system can provide HDF5 through its packaging system (e.g. macports in macOS, rpm in linux distributions), this is often unsuitable for running RH. The reason is that there needs to be consistency between the compiler, MPI libraries used, and the HDF5 library must be compiled with parallel support.

Compiling all of this from scratch is burdensome and prone to fail. Care must be taken to ensure that each library is built properly, and that MPI functions well. Errors with MPI libraries are particularly difficult to diagnose.

The recommended approach is to use an MPI library already in the system. If this is not possible or it is not working properly, then MPI must also be compiled (see below for instructions for Open MPI). The following instructions go through the different steps of compiling the libraries. They assume a bash shell is used, so please adapt if you need csh or tcsh.

Note: The download binary is assumed to be *curl* (commonly available in macOS), but one can also use *wget* (commonly available in Linux). To change, replace *curl -O* by *wget -c*.

Note: The instructions below download specific versions of the libraries, the latest at the time of writing. These should work, but feel free to use more recent versions if available.

Compilers and installation directory

One should compile all the libraries with the same compiler or family of compilers. And of course be aware of the different flags of each. Once MPI is compiled, one should always use the *mpicc* and *mpif90* binaries for the compiler of HDF5. If compiling MPI, then one should use the proper name (e.g. *gcc*, *clang*, *icc*). For this, one uses the *CC* environment variable, e.g.:

```
export CC=mpicc      # or gcc, clang, icc,
export FC=gfortran   # or mpif90, ifort
```

All of the instructions below assume all the libraries to be installed under a directory local to the user, and set to the *MYPREFIX* environment variable, e.g.:

```
export MYPREFIX=/home/user/local
```

Compiling Open MPI (optional)

Use this if you don't currently have any MPI library or suspect it is not working properly (e.g. too old).