grackle Documentation

Release 3.1

Apr 03, 2018

Contents

Grackle is a chemistry and radiative cooling library for astrophysical simulations and models. Grackle has interfaces for C, C++, Fortran, and Python codes and provides:

- two options for primordial chemistry and cooling:
  1. non-equilibrium primordial chemistry network for atomic H, D, and He as well as H_2 and HD, including H_2 formation on dust grains.
  2. tabulated H and He cooling rates calculated with the photo-ionization code, Cloudy.
- tabulated metal cooling rates calculated with Cloudy.
- photo-heating and photo-ionization from two UV backgrounds:
  1. Faucher-Giguere et al. (2009).
  2. Haardt & Madau (2012).
- support for user-provided arrays of volumetric and specific heating rates.

The Grackle provides functions to update chemistry species; solve radiative cooling and update internal energy; and calculate cooling time, temperature, pressure, and ratio of specific heats (gamma).

Contents:

CHAPTER 1

Installation

The compilation process for grackle is very similar to that for Enzo. For more details on the Enzo build system, see the Enzo build documentation.

1.1 Dependencies

In addition to C/C++ and Fortran compilers, the following dependency must also be installed:

- HDF5, the hierarchical data format. HDF5 also may require the szip and zlib libraries, which can be found at the HDF5 website. Compiling with HDF5 1.8 or greater requires that the compiler directive `H5_USE_16_API` be specified. This can be done with `-DH5_USE_16_API`, which is in the machine specific make files.

1.2 Downloading

Grackle is available in a mercurial repository here. The mercurial site is here and an excellent tutorial can be found here. With mercurial installed, grackle can be obtained with the following command:

```
~ $ hg clone https://bitbucket.org/grackle/grackle
```

1.3 Building

1. Initialize the build system.

```
~ $ cd grackle
~/grackle $ ./configure
```

2. Proceed to the source directory.

```
~/grackle $ cd src/clib
```

3. Configure the build system.

Note: As of version 2.1, Grackle uses `libtool` for building and installation. As such, both shared and static libraries will be built automatically and it is not necessary to add the -fPIC compiler flag.

Compile settings for different systems are stored in files starting with "Make.mach" in the source directory. Grackle comes with three sample make macros: `Make.mach.darwin` for Mac OSX, `Make.mach.linux-gnu` for Linux systems, and an unformatted `Make.mach.unknown`. If you have a make file prepared for an Enzo install, it cannot be used straight away, but is a very good place to start.

Once you have chosen the make file to be used, a few variables should be set:

- `MACH_LIBTOOL` - path to `libtool` executable. Note, on a Mac, this should point to `glibtool`, which can be installed with macports or homebrew.
- `LOCAL_HDF5_INSTALL` - path to your hdf5 installation.
- `LOCAL_FC_INSTALL` - path to Fortran compilers (not including the bin subdirectory).
- `MACH_INSTALL_PREFIX` - path where grackle header and library files will be installed.
- `MACH_INSTALL_LIB_DIR` - path where libgrackle will be installed (only set if different from MACH_INSTALL_PREFIX/lib).
- `MACH_INSTALL_INCLUDE_DIR` - path where grackle header files will be installed (only set if different from MACH_INSTALL_PREFIX/include).

Once the proper variables are set, they are loaded into the build system by doing the following:

```
~/grackle/src/clib $ make machine-<system>
```

Where system refers to the make file you have chosen. For example, if you chose `Make.mach.darwin`, type:

```
~/grackle/src/clib $ make machine-darwin
```

Custom make files can be saved and loaded from a **.grackle** directory in the home directory.

1.3.1 Compiler Settings

There are three compile options available for setting the precision of baryon fields, compiler optimization, and enabling OpenMP. To see them, type:

```
 ~/grackle/src/clib $ make show-config

MACHINE: Darwin (OSX)
MACHINE-NAME: darwin

CONFIG_PRECISION  [precision-{32,64}]                    : 64
CONFIG_OPT  [opt-{warn,debug,high,aggressive}]           : high
CONFIG_OMP  [omp-{on,off}]                               : off
```

For example, to change the optimization to high, type:

```
~/grackle/src/clib $ make opt-high
```

Custom settings can be saved for later use by typing:

```
~/grackle/src/clib $ make save-config-<keyword>
```

They will be saved in the **.grackle** directory in your home directory. To reload them, type:

```
~/grackle/src/clib $ make load-config-<keyword>
```

For a list of all available make commands, type:

```
~/grackle/src/clib $ make help

========================================================================
   Grackle Makefile Help
========================================================================

   make                Compile and generate libgrackle
   make install        Copy the library somewhere
   make help           Display this help information
   make clean          Remove object files, executable, etc.
   make dep            Create make dependencies in DEPEND file

   make show-version   Display revision control system branch and revision
   make show-diff      Display local file modifications

   make help-config    Display detailed help on configuration make targets
   make show-config    Display the configuration settings
   make show-flags     Display specific compilation flags
   make default        Reset the configuration to the default values
```

4. Compile and Install

To build the code, type:

```
~/grackle/src/clib $ make
Updating DEPEND
Compiling calc_rates.F
Compiling cool1d_multi.F
....

Linking
Success!
```

Then, to install:

```
~/grackle/src/clib $ make install
```

5. Test your Installation

Once installed, you can test your installation with the provided example to assure it is functioning correctly. If something goes wrong in this process, check the `out.compile` file to see what went wrong during compilation, or use `ldd` (`otool -L` on Mac) on your executable to determine what went wrong during linking.

```
~/grackle/src/clib $ cd ../example
~/grackle/src/example $ make clean
~/grackle/src/example $ make

Compiling cxx_example.C
Linking
```

```
Success!

~/grackle/src/example $ ./cxx_example

The Grackle Version 2.2
Mercurial Branch   default
Mercurial Revision b4650914153d

Initializing grackle data.
with_radiative_cooling: 1.
primordial_chemistry: 3.
metal_cooling: 1.
UVbackground: 1.
Initializing Cloudy cooling: Metals.
cloudy_table_file: ../../input/CloudyData_UVB=HM2012.h5.
Cloudy cooling grid rank: 3.
Cloudy cooling grid dimensions: 29 26 161.
Parameter1: -10 to 4 (29 steps).
Parameter2: 0 to 14.849 (26 steps).
Temperature: 1 to 9 (161 steps).
Reading Cloudy Cooling dataset.
Reading Cloudy Heating dataset.
Initializing UV background.
Reading UV background data from ../../input/CloudyData_UVB=HM2012.h5.
UV background information:
Haardt & Madau (2012, ApJ, 746, 125) [Galaxies & Quasars]
z_min =  0.000
z_max = 15.130
Setting UVbackground_redshift_on to 15.130000.
Setting UVbackground_redshift_off to 0.000000.
Cooling time = -1.434987e+13 s.
Temperature = 4.637034e+02 K.
Pressure = 3.345738e+34.
gamma = 1.666645e+00.
```

In order to verify that Grackle is fully functional, try *running the test suite*.

CHAPTER 2

Running the Test Suite

Grackle contains a number of unit and answer tests to verify that everything is working properly. These will verify that:

- proper and comoving unit systems are consistent
- atomic, primordial collisional ionization equilibrium agrees with the analytical solution
- all code examples build and run
- all python examples run and give correct results
- all Python code conforms to PEP 8

Once you have installed *pygrackle*, the tests can be run from the **src** directory by typing `make test`:

```
~ $ cd grackle/src
~/grackle/src $ make test
```

or from the **src/python** directory by typing `py.test`:

```
~ $ cd grackle/src/python
~/grackle/src $ py.test

===================================== test session starts
=====================================
platform darwin -- Python 2.7.11, pytest-2.8.1, py-1.4.30, pluggy-0.3.1
rootdir: /Users/britton/Documents/work/simulation/grackle/grackle/src/python, inifile:
collected 13 items

tests/test_chemistry.py ...
tests/test_code_examples.py ....
tests/test_examples.py ........
tests/test_flake8.py .
tests/test_primordial.py .

================================== 17 passed in 68.83 seconds
==================================
```

Now it's time to *integrate grackle into your simulation code*.

CHAPTER 3

Adding Grackle to Your Simulation Code

The majority of this document follows the implementation of Grackle in a C++ simulation code. Full implementation examples for C, C++, and Fortran are also available in the Grackle source. See *Example Executables* for more information. For a list of all available functions, see the *API Reference*.

3.1 Example Executables

The grackle source code contains examples for C, C++, and Fortran codes. They are located in the **src/example** directory and provide examples of calling all of grackle's functions.

- **c_example.c** - C example
- **cxx_example.C** - C++ example
- **cxx_omp_example.C** - C++ example using OpenMP
- **fortran_example.F** - Fortran example

Once you have already installed the grackle library, you can build the examples by typing *make* and the name of the file without extension. For example, to build the C++ example, type:

```
$ make cxx_example
```

To run the example, make sure to add the path to the directory containing the installed **libgrackle.so** to your LD_LIBRARY_PATH (or DYLD_LIBRARY_PATH on Mac).

3.2 Header Files

Seven header files are installed with the grackle library. They are:

- **grackle.h** - the primary header file, containing declarations for all the available functions and data structures. This is the only header file that needs to be included for C and C++ codes.
- **grackle.def** - the header file to be used in Fortran codes. Only this file needs to be included.

- **grackle_types.h** - defines the variable type *gr_float*, the field structure *grackle_field_data*, and the units structure *code_units*.
- **grackle_chemistry_data.h** - defines the *chemistry_data* structure, which stores all Grackle run-time parameters and the *chemistry_data_storage* structure, which stores all chemistry and cooling rate data.
- **grackle_fortran_types.def** - similar to **grackle_types.h**, but used with Fortran codes. This defines the variable type *R_PREC* as either real*4 or real*8.
- **grackle_fortran_interface.def** - defines the Fortran interface, including the Fortran analogs of *grackle_field_data*, *code_units*, and `grackle_chemistry_data`.
- **grackle_macros.h** - contains some macros used internally.

For C and C++ codes, the only source file that needs to be included in your simulation code is **grackle.h**. For Fortran, use **grackle.def**. Since Grackle is written in C, including **grackle.h** in a C++ code requires the *extern "C"* directive.

```
extern "C" {
#include <grackle.h>
}
```

3.3 Data Types

The grackle library provides a configurable variable type to control the precision of the baryon fields passed to the grackle functions. For C and C++ codes, this is *gr_float*. For Fortran codes, this is *R_PREC*. The precision of these types can be configured with the *precision* compile option. Compile with *precision-32* to make *gr_float* and *R_PREC* a 4 byte float (*float* for C/C++ and *real*4* for Fortran). Compile with *precision-64* to make *gr_float* and *R_PREC* an 8 byte float (*double* for C/C++ and *real*8* for Fortran).

gr_float
: Floating point type used for the baryon fields. This is of type *float* if compiled with *precision-32* and type double if compiled with *precision-64*.

R_PREC
: The Fortran analog of *gr_float*. This is of type *real*4* if compiled with *precision-32* and type *real*8* if compiled with *precision-64*.

3.4 Enabling Output

By default, grackle will not print anything but error messages. However, a short summary of the running configuration can be printed by setting `grackle_verbose` to 1. In a parallel code, it is recommended that output only be enabled for the root process.

```
// Enable output
grackle_verbose = 1;
```

3.5 Code Units

It is strongly recommended to use comoving coordinates with any cosmological simulation. The *code_units* structure contains conversions from code units to CGS. If *comoving_coordinates* is set to 0, it is assumed that the fields passed into the solver are in the proper frame. All of the units (density, length, time, velocity, and expansion factor) must be set. When using the proper frame, *a_units* (units for the expansion factor) must be set to 1.0.

code_units
This structure contains the following members.

int **comoving_coordinates**
If set to 1, the incoming field data is assumed to be in the comoving frame. If set to 0, the incoming field data is assumed to be in the proper frame.

double **density_units**
Conversion factor to be multiplied by density fields to return densities in proper g/cm^3.

double **length_units**
Conversion factor to be multiplied by length variables to return lengths in proper cm.

double **time_units**
Conversion factor to be multiplied by time variables to return times in s.

double **velocity_units**
Conversion factor to be multiplied by velocities to return proper cm/s.

double **a_units**
Conversion factor to be multiplied by the expansion factor such that $a_{true} = a_{code} * a\_units$.

double **a_value**
The current value of the expansion factor in units of *a_units*. The conversion from redshift to expansion factor in code units is given by *a_value* = 1 / (1 + z) / *a_units*. If the simulation is not cosmological, *a_value* should be set to 1. Note, if *a_value* is set to something other than 1 in a non-cosmological simulation, all redshift dependent chemistry and cooling terms will be set corresponding to the redshift given.

```
code_units my_units;
my_units.comoving_coordinates = 0; // 1 if cosmological sim, 0 if not
my_units.density_units = 1.67e-24; // 1 m_H/cc
my_units.length_units = 3.086e21;  // 1 kpc
my_units.time_units = 3.15569e13;  // 1 Myr
my_units.velocity_units = my_units.length_units / my_units.time_units;
my_units.a_units = 1.0;            // units for the expansion factor
my_units.a_value = 1. / (1. + current_redshift) / my_units.a_units;
```

If *comoving_coordinates* is set to 1, it is assumed that the fields being passed to the solver are in the comoving frame. Hence, the units must convert from code units in the **comoving** frame to CGS in the **proper** frame.

Note: With `comoving_coordinate` set to 1, velocity units need to be defined in the following way.

```
my_units.velocity_units = my_units.a_units *
  (my_units.length_units / a_value) / my_units.time_units; // since u = a * dx/dt
```

For an example of using comoving units, see the units system in the Enzo code. For cosmological simulations, a comoving unit system is preferred, though not required, since it allows the densities to stay close to 1.0.

3.6 Chemistry Data

The main Grackle header file contains a structure of type *chemistry_data* called `grackle_data`, which contains all of the parameters that control the behavior of the solver. The routine, `set_default_chemistry_parameters()` is responsible for the initial setup of this structure and for setting of all the default parameter values. This function must be handed a pointer to an instance of *chemistry_data*, which will then be attached to `grackle_data`. The function will return an integer indicating success (1) or failure

(0). After this, parameters can then be set to their desired values by accessing `grackle_data`. See *Parameters and Data Files* for a full list of the available parameters.

chemistry_data
: This structure holds all grackle run-time parameters, which are listed in *Parameters and Data Files*.

chemistry_data_storage
: This structure holds all chemistry and cooling rate arrays. All functions described here make use of an internally stored instance of this type. The user will not normally encounter this data type, except when using the *Internal Functions*.

```
chemistry_data *my_grackle_data;
my_grackle_data = new chemistry_data;
if (set_default_chemistry_parameters(my_grackle_data) == 0) {
  fprintf(stderr, "Error in set_default_chemistry_parameters.\n");
}

// Set parameter values for chemistry.
// Now access the global copy of the chemistry_data struct (grackle_data).
grackle_data->use_grackle = 1;            // chemistry on
grackle_data->with_radiative_cooling = 1; // cooling on
grackle_data->primordial_chemistry = 3;   // molecular network with H, He, D
grackle_data->metal_cooling = 1;          // metal cooling on
grackle_data->UVbackground = 1;           // UV background on
grackle_data->grackle_data_file = "CloudyData_UVB=HM2012.h5"; // data file
```

Once the desired parameters have been set, the chemistry and cooling rates must be initialized by calling `initialize_chemistry_data()` with a pointer to the *code_units* struct created earlier. This function will return an integer indicating success (1) or failure (0).

```
// Finally, initialize the chemistry object.
if (initialize_chemistry_data(&my_units) == 0) {
  fprintf(stderr, "Error in initialize_chemistry_data.\n");
  return 0;
}
```

The Grackle is now ready to be used.

3.7 Running with OpenMP

As of version 2.2, Grackle can be run with OpenMP parallelism. To do this, the library must first be compiled with OpenMP support enabled by issuing the command, "make omp-on", before compiling. See *Compiler Settings* for more information on how to change settings.

For an example of how to compile your code with OpenMP, see the **cxx_omp_example.C** code example (*Example Executables*). Once your code has been compiled with OpenMP enabled, the number of threads used can be controlled by setting the *omp_nthreads* parameter, stored in the `grackle_data` struct.

```
// 8 threads per process
grackle_data->omp_nthreads = 8;
```

If not set, this parameter will be set to the maximum number of threads possible, as determined by the system or as configured by setting the `OMP_NUM_THREADS` environment variable.

3.8 Creating the Necessary Fields

As of version 3.0, the various density and energy fields are passed to Grackle's functions using a struct of type *grackle_field_data*. The struct contains information about the size and shape of the field arrays and pointers to all field arrays.

grackle_field_data
This structure is used to pass field data to Grackle's functions. It contains the following members:

int **grid_rank**
The active dimensions (not including ignored boundary zones) of the field arrays.

int* **grid_dimension**
This should point to an array of size *grid_rank*. This stores the size of the field arrays in each dimension.

int* **grid_start**
This should point to an array of size *grid_rank*. This stores the starting value in each dimension for the field data. This can be used to ignore boundary cells in grid data.

int* **grid_end**
This should point to an array of size *grid_rank*. This stores the end value in each dimension for the field data. This can be used to ignore boundary cells in grid data.

*gr_float** **grid_dx**
This is the grid cell width in *length_units*. This is currently used only in computing approximate H2 self-shielding when H2 is tracked (*primordial_chemistry* >= 2) and *H2_self_shielding* is set to 1.

*gr_float** **density**
Pointer to the density field array.

*gr_float** **HI_density**
Pointer to the HI density field array. Used when *primordial_chemistry* is set to 1, 2, or 3.

*gr_float** **HII_density**
Pointer to the HII density field array. Used when *primordial_chemistry* is set to 1, 2, or 3.

*gr_float** **HM_density**
Pointer to the H^- density field array. Used when *primordial_chemistry* is set to 2 or 3.

*gr_float** **HeI_density**
Pointer to the HeI density field array. Used when *primordial_chemistry* is set to 1, 2, or 3.

*gr_float** **HeII_density**
Pointer to the HeII density field array. Used when *primordial_chemistry* is set to 1, 2, or 3.

*gr_float** **HeIII_density**
Pointer to the HeIII density field array. Used when *primordial_chemistry* is set to 1, 2, or 3.

*gr_float** **H2I_density**
Pointer to the H_2 density field array. Used when *primordial_chemistry* is set to 2 or 3.

*gr_float** **H2II_density**
Pointer to the H_2^+ density field array. Used when *primordial_chemistry* is set to 2 or 3.

*gr_float** **DI_density**
Pointer to the DI density field array. Used when *primordial_chemistry* is set to 3.

*gr_float** **DII_density**
Pointer to the DII density field array. Used when *primordial_chemistry* is set to 3.

*gr_float** **HDI_density**
Pointer to the HD density field array. Used when *primordial_chemistry* is set to 3.

*gr_float** **e_density**
Pointer to the electron density field array. Used when *primordial_chemistry* is set to 1, 2, or 3. Note, the electron mass density should be scaled by the ratio of the proton mass to the electron mass such that the electron density in the code is the electron number density times the **proton** mass.

*gr_float** **metal_density**
Pointer to the metal density field array. Used when *metal_cooling* is set to 1.

*gr_float** **internal_energy**
Pointer to the internal energy field array.

*gr_float** **x_velocity**
Pointer to the x-velocity field array. Currently not used.

*gr_float** **y_velocity**
Pointer to the y-velocity field array. Currently not used.

*gr_float** **z_velocity**
Pointer to the z-velocity field array. Currently not used.

*gr_float** **volumetric_heating_rate**
Pointer to values containing volumetric heating rates. Rates should be in units of erg/s/cm^3. Used when *use_volumetric_heating_rate* is set to 1.

*gr_float** **specific_heating_rate**
Pointer to values containing specific heating rates. Rates should be in units of erg/s/g. Used when *use_specific_heating_rate* is set to 1.

gr_float ***RT_heating_rate**
Pointer to the radiation transfer heating rate field. Rates should be in units of erg/s/cm^3. Used when *use_radiative_transfer* is set to 1.

gr_float ***RT_HI_ionization_rate**
Pointer to the HI photo-ionization rate field used with radiative transfer. Rates should be in units of 1/*time_units*. Used when *use_radiative_transfer* is set to 1.

gr_float ***RT_HeI_ionization_rate**
Pointer to the HeI photo-ionization rate field used with radiative transfer. Rates should be in units of 1/*time_units*. Used when *use_radiative_transfer* is set to 1.

gr_float ***RT_HeII_ionization_rate**
Pointer to the HeII photo-ionization rate field used with radiative transfer. Rates should be in units of 1/*time_units*. Used when *use_radiative_transfer* is set to 1.

gr_float ***RT_H2_dissociation_rate**
Pointer to the H_2 photo-dissociation rate field used with radiative transfer. Rates should be in units of 1/*time_units*. Used when *use_radiative_transfer* is set to 1 and *primordial_chemistry* is either 2 or 3.

gr_float ***H2_self_shielding_length**
Pointer to a field containing lengths to be used for calculating molecular hydrogen column denisty for $H2_2$self-shielding. Used when *H2_self_shielding* is set to 2. Field data should be in *length_units*.

It is not necessary to attach a pointer to any field that you do not intend to use.

```
// Create struct for storing grackle field data
grackle_field_data my_fields;

// Set grid dimension and size.
```

```
// grid_start and grid_end are used to ignore ghost zones.
int field_size = 1;
my_fields.grid_rank = 3;
my_fields.grid_dimension = new int[3];
my_fields.grid_start = new int[3];
my_fields.grid_end = new int[3];
my_fields.grid_dx  = 1.0; // only matters if H2 self-shielding is used
for (int i = 0;i < 3;i++) {
  my_fields.grid_dimension[i] = 1;
  my_fields.grid_start[i] = 0;
  my_fields.grid_end[i] = 0;
}
my_fields.grid_dimension[0] = field_size;
my_fields.grid_end[0] = field_size - 1;

// Set field arrays.
my_fields.density         = new gr_float[field_size];
my_fields.internal_energy = new gr_float[field_size];
my_fields.x_velocity      = new gr_float[field_size];
my_fields.y_velocity      = new gr_float[field_size];
my_fields.z_velocity      = new gr_float[field_size];
// for primordial_chemistry >= 1
my_fields.HI_density      = new gr_float[field_size];
my_fields.HII_density     = new gr_float[field_size];
my_fields.HeI_density     = new gr_float[field_size];
my_fields.HeII_density    = new gr_float[field_size];
my_fields.HeIII_density   = new gr_float[field_size];
my_fields.e_density       = new gr_float[field_size];
// for primordial_chemistry >= 2
my_fields.HM_density      = new gr_float[field_size];
my_fields.H2I_density     = new gr_float[field_size];
my_fields.H2II_density    = new gr_float[field_size];
// for primordial_chemistry >= 3
my_fields.DI_density      = new gr_float[field_size];
my_fields.DII_density     = new gr_float[field_size];
my_fields.HDI_density     = new gr_float[field_size];
// for metal_cooling = 1
my_fields.metal_density   = new gr_float[field_size];
// volumetric heating rate (provide in units [erg s^-1 cm^-3])
my_fields.volumetric_heating_rate = new gr_float[field_size];
// specific heating rate (provide in units [egs s^-1 g^-1]
my_fields.specific_heating_rate = new gr_float[field_size];
// heating rate from radiative transfer calculations (provide in units [erg s^-1 cm^-
↪3]
my_fields.RT_heating_rate = new gr_float[field_size];
// HI ionization rate from radiative transfer calculations (provide in units of [ 1/
↪time_units ]
my_fields.RT_HI_ionization_rate = new gr_float[field_size];
// HeI ionization rate from radiative transfer calculations (provide in units of [1/
↪time_units]
my_fields.RT_HeI_ionization_rate = new gr_float[field_size];
// HeII ionization rate from radiative transfer calculations (provide in units of [1/
↪time_units]
my_fields.RT_HeII_ionization_rate = new gr_float[field_size];
// H2 dissociation rate from radiative transfer calculations (provide in units of [1/
↪time_units]
my_fields.RT_H2_dissociation_rate = new gr_float[field_size];
```

Note: The electron mass density should be scaled by the ratio of the proton mass to the electron mass such that the electron density in the code is the electron number density times the **proton** mass.

3.9 Calling the Available Functions

There are five functions available, one to solve the chemistry and cooling and four others to calculate the cooling time, temperature, pressure, and the ratio of the specific heats (gamma). The arguments required are the *code_units* structure and the *grackle_field_data* struct. For the chemistry solving routine, a timestep must also be given. For the four field calculator routines, the array to be filled with the field values must be created and passed as an argument as well.

3.9.1 Solve the Chemistry and Cooling

```
// some timestep (one million years)
double dt = 3.15e7 * 1e6 / my_units.time_units;

if (solve_chemistry(&my_units, &my_fields, dt) == 0) {
  fprintf(stderr, "Error in solve_chemistry.\n");
  return 0;
}
```

3.9.2 Calculating the Cooling Time

```
gr_float *cooling_time;
cooling_time = new gr_float[field_size];
if (calculate_cooling_time(&my_units, &my_fields,
                           cooling_time) == 0) {
  fprintf(stderr, "Error in calculate_cooling_time.\n");
  return 0;
}
```

3.9.3 Calculating the Temperature Field

```
gr_float *temperature;
temperature = new gr_float[field_size];
if (calculate_temperature(&my_units, &my_fields,
                          temperature) == 0) {
  fprintf(stderr, "Error in calculate_temperature.\n");
  return EXIT_FAILURE;
}
```

3.9.4 Calculating the Pressure Field

```
gr_float *pressure;
pressure = new gr_float[field_size];
if (calculate_pressure(&my_units, &my_fields,
```

```
                        pressure) == 0) {
  fprintf(stderr, "Error in calculate_pressure.\n");
  return EXIT_FAILURE;
}
```

3.9.5 Calculating the Gamma Field

```
gr_float *gamma;
gamma = new gr_float[field_size];
if (calculate_gamma(&my_units, &my_fields,
                    gamma) == 0) {
  fprintf(stderr, "Error in calculate_gamma.\n");
  return EXIT_FAILURE;
}
```

CHAPTER 4

Parameters and Data Files

4.1 Parameters

For all on/off integer flags, 0 is off and 1 is on.

int **use_grackle**
: Flag to activate the grackle machinery. Default: 0.

int **with_radiative_cooling**
: Flag to include radiative cooling and actually update the thermal energy during the chemistry solver. If off, the chemistry species will still be updated. The most common reason to set this to off is to iterate the chemistry network to an equilibrium state. Default: 1.

int **primordial_chemistry**
: Flag to control which primordial chemistry network is used. Default: 0.

- 0: no chemistry network. Radiative cooling for primordial species is solved by interpolating from lookup tables calculated with Cloudy.
- 1: 6-species atomic H and He. Active species: H, H^+, He, He^+, $^{++}$, e^-.
- 2: 9-species network including atomic species above and species for molecular hydrogen formation. This network includes formation from the H^- and H_2^+ channels, three-body formation (H+H+H and H+H+H_2), H_2 rotational transitions, chemical heating, and collision-induced emission (optional). Active species: above + H^-, H_2, H_2^+.
- 3: 12-species network include all above plus HD rotation cooling. Active species: above + D, D^+, HD.

Note: In order to make use of the non-equilibrium chemistry network (*primordial_chemistry* options 1-3), you must add and advect baryon fields for each of the species used by that particular option.

int **h2_on_dust**
: Flag to enable H_2 formation on dust grains, dust cooling, and dust-gas heat transfer follow Omukai (2000). This assumes that the dust to gas ratio scales with the metallicity. Default: 0.

int **metal_cooling**
: Flag to enable metal cooling using the Cloudy tables. If enabled, the cooling table to be used must be specified with the *grackle_data_file* parameter. Default: 0.

Note: In order to use the metal cooling, you must add and advect a metal density field.

---

int **cmb_temperature_floor**
: Flag to enable an effective CMB temperature floor. This is implemented by subtracting the value of the cooling rate at T_{CMB} from the total cooling rate. Default: 1.

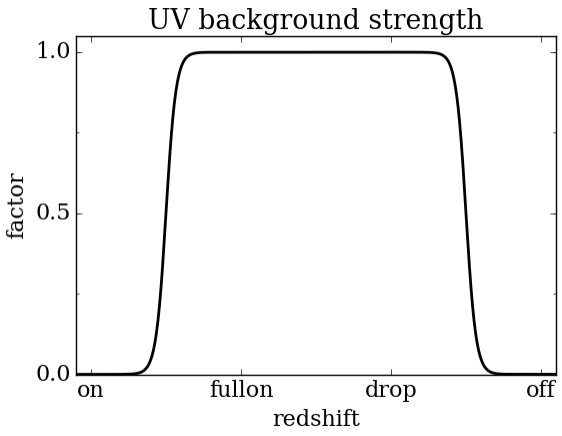
int **UVbackground**
: Flag to enable a UV background. If enabled, the cooling table to be used must be specified with the *grackle_data_file* parameter. Default: 0.

float **UVbackground_redshift_on**
: Used in combination with *UVbackground_redshift_fullon*, *UVbackground_redshift_drop*, and *UVbackground_redshift_off* to set an attenuation factor for the photo-heating and photo-ionization rates of the UV background model. See the figure below for an illustration its behavior. If not set, this parameter will be set to the highest redshift of the UV background data being used.

float **UVbackground_redshift_fullon**
: Used in combination with *UVbackground_redshift_on*, *UVbackground_redshift_drop*, and *UVbackground_redshift_off* to set an attenuation factor for the photo-heating and photo-ionization rates of the UV background model. See the figure below for an illustration its behavior. If not set, this parameter will be set to the highest redshift of the UV background data being used.

float **UVbackground_redshift_drop**
: Used in combination with *UVbackground_redshift_on*, *UVbackground_redshift_fullon*, and *UVbackground_redshift_off* to set an attenuation factor for the photo-heating and photo-ionization rates of the UV background model. See the figure below for an illustration its behavior. If not set, this parameter will be set to the lowest redshift of the UV background data being used.

float **UVbackground_redshift_off**
: Used in combination with *UVbackground_redshift_on*, *UVbackground_redshift_fullon*, and *UVbackground_redshift_drop* to set an attenuation factor for the photo-heating and photo-ionization rates of the UV background model. See the figure below for an illustration its behavior. If not set, this parameter will be set to the lowest redshift of the UV background data being used.



char* **grackle_data_file**
: Path to the data file containing the metal cooling and UV background tables. Default: "".

float **Gamma**
The ratio of specific heats for an ideal gas. A direct calculation for the molecular component is used if *primordial_chemistry* > 1. Default: 5/3.

int **three_body_rate**
Flag to control which three-body H_2 formation rate is used.

- 0: Abel, Bryan & Norman (2002)
- 1: Palla, Salpeter & Stahler (1983)
- 2: Cohen & Westberg (1983)
- 3: Flower & Harris (2007)
- 4: Glover (2008)
- 5: Forrey (2013).

The first five options are discussed in Turk et. al. (2011). Default: 0.

int **cie_cooling**
Flag to enable H_2 collision-induced emission cooling from Ripamonti & Abel (2004). Default: 0.

int **h2_optical_depth_approximation**
Flag to enable H_2 cooling attenuation from Ripamonti & Abel (2004). Default: 0.

int **photoelectric_heating**
Flag to enable a spatially uniform heating term approximating photo-electric heating from dust from Tasker & Bryan (2008). Default: 0.

int **photoelectric_heating_rate**
If *photoelectric_heating* enabled, the heating rate in units of erg cm^{-3} s^{-1}. Default: 8.5e-26.

int **Compton_xray_heating**
Flag to enable Compton heating from an X-ray background following Madau & Efstathiou (1999). Default: 0.

float **LWbackground_intensity**
Intensity of a constant Lyman-Werner H_2 photo-dissociating radiation field in units of 10^{-21} erg s^{-1} cm^{-2} Hz^{-1} sr^{-1}. Default: 0.

int **LWbackground_sawtooth_suppression**
Flag to enable suppression of Lyman-Werner flux due to Lyman-series absorption (giving a sawtooth pattern), taken from Haiman & Abel, & Rees (2000). Default: 0.

int **use_volumetric_heating_rate**
Flag to signal that an array of volumetric heating rates is being provided in the *volumetric_heating_rate* field of the *grackle_field_data* struct. Default: 0.

int **use_specific_heating_rate**
Flag to signal that an array of specific heating rates is being provided in the *specific_heating_rate* field of the *grackle_field_data* struct. Default: 0.

use_radiative_transfer
Flag to signal that arrays of ionization and heating rates from radiative transfer solutions are being provided. Only available if *primordial_chemistry* is greater than 0. HI, HeI, and HeII ionization arrays are provided in *RT_HI_ionization_rate*, *RT_HeI_ionization_rate*, and *RT_HeII_ionization_rate* fields, respectively, of the *grackle_field_data* struct. Associated heating rate is provided in the *RT_heating_rate* field, and H2 photodissociation rate can also be provided in the *RT_H2_dissociation_rate* field when *primordial_chemistry* is set to either 2 or 3. Default: 0.

radiative_transfer_coupled_rate_solver
Flag that must be enabled to couple the passed radiative transfer fields to the chemistry solver. Default: 0.

radiative_transfer_intermediate_step
Flag to enable intermediate stepping in applying radiative transfer fields to chemistry solver. Default: 0.

radiative_transfer_hydrogen_only
Flag to only use hydrogen ionization and heating rates from the radiative transfer solutions. Default: 0.

H2_self_shielding
Switch to enable approximate H_2 self-shielding from both the UV background dissociation rate and the H_2 dissociation rate given by *RT_H2_dissociation_rate* (if present). Three options exist for the length scale used in calculating the H_2 column density. Default: 0.

- 1: Use a Sobolev-like, spherically averaged method from Wolcott-Green et. al. 2011. This option is only valid for Cartesian grid codes in 3D.
- 2: Supply an array of lengths using the *H2_self_shielding_length* field.
- 3: Use the local Jeans length.

self_shielding_method
Switch to enable approximate self-shielding from the UV background. All three of the below methods incorporate Eq. 13 and 14 from Rahmati et. al. 2013. These equations involve using the spectrum averaged photoabsorption cross for the given species (HI or HeI). These redshift dependent values are pre-computed for the HM2012 and FG2011 UV backgrounds and included in their respective cooling data tables. Default: 0

Care is advised in using any of these methods. The default behavior is to apply no self-shielding, but this is not necessarily the proper assumption, depending on the use case. If the user desires to turn on self-shielding, we strongly advise using option 3. All options include HI self-shielding, and vary only in treatment of HeI and HeII. In options 2 and 3, we approximately account for HeI self-shielding by applying the Rahmati et. al. 2013 relations, which are only strictly valid for HI, to HeI under the assumption that it behaves similarly to HI. None of these options are completely correct in practice, but option 3 has produced the most reasonable results in test simulations. Repeating the analysis of Rahmati et. al. 2013 to directly parameterize HeI and HeII self-shielding behavior would be a valuable avenue of future research in developing a more complete self-shielding model. Each self-shielding option is described below.

- 0: No self shielding. Elements are optically thin to the UV background.
- **1: Not Recommended. Approximate self-shielding in HI only.** HeI and HeII are left as optically thin.
- **2: Approximate self-shielding in both HI and HeI. HeII remains** optically thin.
- **3: Approximate self-shielding in both HI and HeI, but ignoring** HeII ionization and heating from the UV background entirely (HeII ionization and heating rates are set to zero).

These methods only work in conjunction with using updated Cloudy cooling tables, denoted with "_shielding". These tables properly account for the decrease in metal line cooling rates in self-shielded regions, which can be significant.

int **omp_nthreads**
Sets the number of OpenMP threads. If not set, this will be set to the maximum number of threads possible, as determined by the system or as configured by setting the `OMP_NUM_THREADS` environment variable. Note, Grackle must be compiled with OpenMP support enabled. See *Running with OpenMP*.

4.2 Data Files

All data files are located in the **input** directory in the source.

The first three files contain the heating and cooling rates for both primordial and metal species as well as the UV background photo-heating and photo-ionization rates. For all three files, the valid density and temperature range is given below. Extrapolation is performed when outside of the data range. The metal cooling rates are stored for solar metallicity and scaled linearly with the metallicity of the gas.

Valid range:

- number density: $-10 < \log_{10}(n_H / cm^{-3}) < 4$
- temperature: the temperature range is $1 < \log_{10}(T / K) < 9$.

Data files:

- **CloudyData_noUVB.h5** - cooling rates for collisional ionization equilibrium.
- **CloudyData_UVB=FG2011.h5** - heating and cooling rates and UV background rates from the work of Faucher-Giguere et. al. (2009), updated in 2011. The maxmimum redshift is 10.6. Above that, collisional ionization equilibrium is assumed.
- **CloudyData_UVB=HM2012.h5** - heating and cooling rates and UV background rates from the work of Haardt & Madau (2012). The maximum redshift is 15.13. Above that, collisional ionization equilibrium is assumed.

To use the self-shielding approximation (see `self_shielding_method`), one must properly account for the change in metal line cooling rates in self-shielded regions. Using the optically thin tables described above can result in an order of magnitude overestimation in the net cooling rate at certain densities. We have re-computed these tables by constructing Jeans-length depth models in Cloudy at each density - temperature pair, tabulating the cooling and heating rates from the core of each of these clouds. These models enforce a maximum depth of 100 pc. In addition, these tables contain the spectrum averaged absorption cross sections needed for the Rahmati et. al. 2013 self-shielding approximations. Currently only the HM2012 table has been recomputed.

- **CloudyData_UVB=HM2012_shielded.h5** - updated heating and cooling rates with the HM2012 UV background, accounting for self-shielding.

The final file includes only metal cooling rates under collisional ionization equilibrium, i.e., no incident radiation field. This table extends to higher densities and also varies in metallicity rather than scaling proportional to the solar value. This captures the thermalization of metal coolants occuring at high densities, making this table more appropriate for simulations of collapsing gas-clouds.

Valid range:

- number density: $-6 < \log_{10}(n_H / cm^{-3}) < 12$
- metallicity: $-6 < \log_{10}(Z / Z_{sun}) < 1$
- temperature: the temperature range is $1 < \log_{10}(T / K) < 8$.

Data file:

- **cloudy_metals_2008_3D.h5** - collisional ionization equilibrium, metal cooling rates only.

CHAPTER 5

API Reference

Grackle has two versions of most functions. The *Primary Functions*, discussed in *Calling the Available Functions*, make use of the *grackle_field_data* struct and internally stored instances *chemistry_data* and *chemistry_data_storage* structs. A set of *Internal Functions* also exist and require the user to provide their own instances of the *chemistry_data* and *chemistry_data_storage*.

5.1 Primary Functions

int set_default_chemistry_parameters(chemistry_data *my_grackle_data);
Initializes the grackle_data data structure. This must be called before run-time parameters can be set.

Parameters

- **my_grackle_data** (chemistry_data*) – run-time parameters

Return type int

Returns 1 (success) or 0 (failure)

int initialize_chemistry_data(code_units *my_units);
Loads all chemistry and cooling data, given the set run-time parameters. This can only be called after set_default_chemistry_parameters().

Parameters

- **my_units** (code_units*) – code units conversions

Return type int

Returns 1 (success) or 0 (failure)

int solve_chemistry(code_units *my_units, grackle_field_data *my_fields, double dt_value);
Evolves the species densities and internal energies over a given timestep by solving the chemistry and cooling rate equations.

Parameters

- **my_units** (code_units*) – code units conversions
- **my_fields** (grackle_field_data*) – field data storage
- **dt_value** (*double*) – the integration timestep in code units

Return type int

Returns 1 (success) or 0 (failure)

int calculate_cooling_time(code_units *my_units, grackle_field_data *my_fields, gr_float *c
Calculates the instantaneous cooling time.

Parameters

- **my_units** (code_units*) – code units conversions
- **my_fields** (grackle_field_data*) – field data storage
- **cooling_time** (gr_float*) – array which will be filled with the calculated cooling time values

Return type int

Returns 1 (success) or 0 (failure)

int calculate_gamma(code_units *my_units, grackle_field_data *my_fields, gr_float *my_gamma
Calculates the effective adiabatic index. This is only useful with *primordial_chemistry* >= 2 as the only thing that alters gamma from the single value is H_2.

Parameters

- **my_units** (code_units*) – code units conversions
- **my_fields** (grackle_field_data*) – field data storage
- **my_gamma** (gr_float*) – array which will be filled with the calculated gamma values

Return type int

Returns 1 (success) or 0 (failure)

int calculate_pressure(code_units *my_units, grackle_field_data *my_fields, gr_float *press
Calculates the gas pressure.

Parameters

- **my_units** (code_units*) – code units conversions
- **my_fields** (grackle_field_data*) – field data storage
- **pressure** (gr_float*) – array which will be filled with the calculated pressure values

Return type int

Returns 1 (success) or 0 (failure)

int calculate_temperature(code_units *my_units, grackle_field_data *my_fields, gr_float *te
Calculates the gas temperature.

Parameters

- **my_units** (code_units*) – code units conversions
- **my_fields** (grackle_field_data*) – field data storage
- **temperature** (gr_float*) – array which will be filled with the calculated temperature values

Return type int

Returns 1 (success) or 0 (failure)

5.2 Internal Functions

These functions are mostly for internal use, but can also be used to call the various functions with different parameter values within a single code.

chemistry_data _set_default_chemistry_parameters(void);
Initializes and returns *chemistry_data* data structure. This must be called before run-time parameters can be set.

Returns data structure containing all run-time parameters and all chemistry and cooling data arrays

Return type *chemistry_data*

int _initialize_chemistry_data(chemistry_data *my_chemistry, chemistry_data_storage *my_rat
Creates all chemistry and cooling rate data and stores within the provided *chemistry_data_storage* struct. This can only be called after `_set_default_chemistry_parameters()`.

Parameters

- **my_chemistry** (*chemistry_data**) – the structure returned by `_set_default_chemistry_parameters()`
- **my_rates** (*chemistry_data_storage**) – chemistry and cooling rate data structure
- **my_units** (*code_units**) – code units conversions

Return type int

Returns 1 (success) or 0 (failure)

int _solve_chemistry(chemistry_data *my_chemistry, chemistry_data_storage *my_rates, code_u
Evolves the species densities and internal energies over a given timestep by solving the chemistry and cooling rate equations.

Parameters

- **my_chemistry** (*chemistry_data**) – the structure returned by `_set_default_chemistry_parameters()`
- **my_rates** (*chemistry_data_storage**) – chemistry and cooling rate data structure
- **my_units** (*code_units**) – code units conversions
- **dt_value** (*double*) – the integration timestep in code units
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int **) – array holding the size of the baryon field in each dimension
- **grid_start** (*int **) – array holding the starting indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones
- **grid_end** (*int **) – array holding the ending indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones.
- **density** (*gr_float**) – array containing the density values in code units
- **internal_energy** (*gr_float**) – array containing the specific internal energy values in code units corresponding to *erg/g*

- **x_velocity** (gr_float*) – array containing the x velocity values in code units
- **y_velocity** (gr_float*) – array containing the y velocity values in code units
- **z_velocity** (gr_float*) – array containing the z velocity values in code units
- **HI_density** (gr_float*) – array containing the HI densities in code units equivalent those of the density array. Used with *primordial_chemistry* >= 1.
- **HII_density** (gr_float*) – array containing the HII densities in code units equivalent those of the density array. Used with *primordial_chemistry* >= 1.
- **HM_density** (gr_float*) – array containing the H^- densities in code units equivalent those of the density array. Used with *primordial_chemistry* >= 2.
- **HeI_density** (gr_float*) – array containing the HeI densities in code units equivalent those of the density array. Used with *primordial_chemistry* >= 1.
- **HeII_density** (gr_float*) – array containing the HeII densities in code units equivalent those of the density array. Used with *primordial_chemistry* >= 1.
- **HeIII_density** (gr_float*) – array containing the HeIII densities in code units equivalent those of the density array. Used with *primordial_chemistry* >= 1.
- **H2I_density** (gr_float*) – array containing the H_2: densities in code units equivalent those of the density array. Used with *primordial_chemistry* >= 2.
- **H2II_density** (gr_float*) – array containing the H_2^+densities in code units equivalent those of the density array. Used with *primordial_chemistry* >= 2.
- **DI_density** (gr_float*) – array containing the DI (deuterium) densities in code units equivalent those of the density array. Used with *primordial_chemistry* = 3.
- **DII_density** (gr_float*) – array containing the DII densities in code units equivalent those of the density array. Used with *primordial_chemistry* = 3.
- **HDI_density** (gr_float*) – array containing the HD densities in code units equivalent those of the density array. Used with *primordial_chemistry* = 3.
- **e_density** (gr_float*) – array containing the e^- densities in code units equivalent those of the density array but normalized to the ratio of the proton to electron mass. Used with *primordial_chemistry* >= 1.
- **metal_density** (gr_float*) – array containing the metal densities in code units equivalent those of the density array. Used with *metal_cooling* = 1.

Return type int

Returns 1 (success) or 0 (failure)

int _calculate_cooling_time(chemistry_data *my_chemistry, chemistry_data_storage *my_rates,
Calculates the instantaneous cooling time.

Parameters

- **my_chemistry** (chemistry_data*) – the structure returned by _set_default_chemistry_parameters()
- **my_rates** (chemistry_data_storage*) – chemistry and cooling rate data structure
- **my_units** (code_units*) – code units conversions
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension

- **grid_start** (*int**) – array holding the starting indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones
- **grid_end** (*int**) – array holding the ending indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones.
- **density** (gr_float*) – array containing the density values in code units
- **internal_energy** (gr_float*) – array containing the specific internal energy values in code units corresponding to *erg/g*
- **x_velocity, y_velocity, z_velocity** (gr_float*) – arrays containing the x, y, and z velocity values in code units
- **HI_density, HII_density, HM_density, HeI_density, HeII_density, HeIII_density, H2I_density, H2II_density, DI_density, DII_density, HDI_density, e_density, metal_density** (gr_float*) – arrays containing the species densities in code units equivalent those of the density array
- **cooling_time** (gr_float*) – array which will be filled with the calculated cooling time values

Return type int

Returns 1 (success) or 0 (failure)

int _calculate_gamma(chemistry_data *my_chemistry, chemistry_data_storage *my_rates, code_u

Calculates the effective adiabatic index. This is only useful with *primordial_chemistry* >= 2 as the only thing that alters gamma from the single value is H_2.

Parameters

- **my_chemistry** (chemistry_data*) – the structure returned by _set_default_chemistry_parameters()
- **my_rates** (chemistry_data_storage*) – chemistry and cooling rate data structure
- **my_units** (code_units*) – code units conversions
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension
- **grid_start** (*int**) – array holding the starting indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones
- **grid_end** (*int**) – array holding the ending indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones.
- **density** (gr_float*) – array containing the density values in code units
- **internal_energy** (gr_float*) – array containing the specific internal energy values in code units corresponding to *erg/g*
- **HI_density, HII_density, HM_density, HeI_density, HeII_density, HeIII_density, H2I_density, H2II_density, DI_density, DII_density, HDI_density, e_density, metal_density** (gr_float*) – arrays containing the species densities in code units equivalent those of the density array
- **my_gamma** (gr_float*) – array which will be filled with the calculated gamma values

Return type int

Returns 1 (success) or 0 (failure)

int _calculate_pressure(chemistry_data *my_chemistry, chemistry_data_storage *my_rates, coo
Calculates the gas pressure.

Parameters

- **my_chemistry** (chemistry_data*) – the structure returned by _set_default_chemistry_parameters()
- **my_rates** (chemistry_data_storage*) – chemistry and cooling rate data structure
- **my_units** (code_units*) – code units conversions
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension
- **grid_start** (*int**) – array holding the starting indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones
- **grid_end** (*int**) – array holding the ending indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones.
- **density** (gr_float*) – array containing the density values in code units
- **internal_energy** (gr_float*) – array containing the specific internal energy values in code units corresponding to *erg/g*
- **HI_density, HII_density, HM_density, HeI_density, HeII_density, HeIII_density, H2I_density, H2II_density, DI_density, DII_density, HDI_density, e_density, metal_density** (gr_float*) – arrays containing the species densities in code units equivalent those of the density array
- **pressure** (gr_float*) – array which will be filled with the calculated pressure values

Return type int

Returns 1 (success) or 0 (failure)

int _calculate_temperature(chemistry_data *my_chemistry, chemistry_data_storage *my_rates,

Parameters

- **my_chemistry** (chemistry_data*) – the structure returned by _set_default_chemistry_parameters()
- **my_rates** (chemistry_data_storage*) – chemistry and cooling rate data structure
- **my_units** (code_units*) – code units conversions
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension
- **grid_start** (*int**) – array holding the starting indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones
- **grid_end** (*int**) – array holding the ending indices in each dimension of the active portion of the baryon fields. This is used to ignore ghost zones.
- **density** (gr_float*) – array containing the density values in code units
- **internal_energy** (gr_float*) – array containing the specific internal energy values in code units corresponding to *erg/g*

- **HI_density, HII_density, HM_density, HeI_density, HeII_density, HeIII_density, H2I_density, H2II_density, DI_density, DII_density, HDI_density, e_density, metal_density** (gr_float*) – arrays containing the species densities in code units equivalent those of the density array
- **temperature** (gr_float*) – array which will be filled with the calculated temperature values

Return type int

Returns 1 (success) or 0 (failure)

Calculates the gas temperature.

CHAPTER 6

Pygrackle: Running Grackle in Python

Grackle comes with a Python interface, called Pygrackle, which provides access to all of Grackle's functionality. Pygrackle requires the following Python packages:

- Cython
- flake8 (only required for the test suite)
- matplotlib
- NumPy
- py.test (only required for the test suite)
- yt

The easiest thing to do is follow the instructions for installing yt, which will provide you with Cython, matplotlib, and NumPy. Flake8 and py.test can then be installed via pip.

6.1 Installing Pygrackle

Once the Grackle library has been built and the above dependencies have been installed, Pygrackle can be installed by moving into the **src/python** directory and running `python setup.py install`.

```
~/grackle $ cd src/python
~/grackle/src/python $ python setup.py install
```

Note: Pygrackle can only be run when Grackle is compiled without OpenMP. See *Running with OpenMP*.

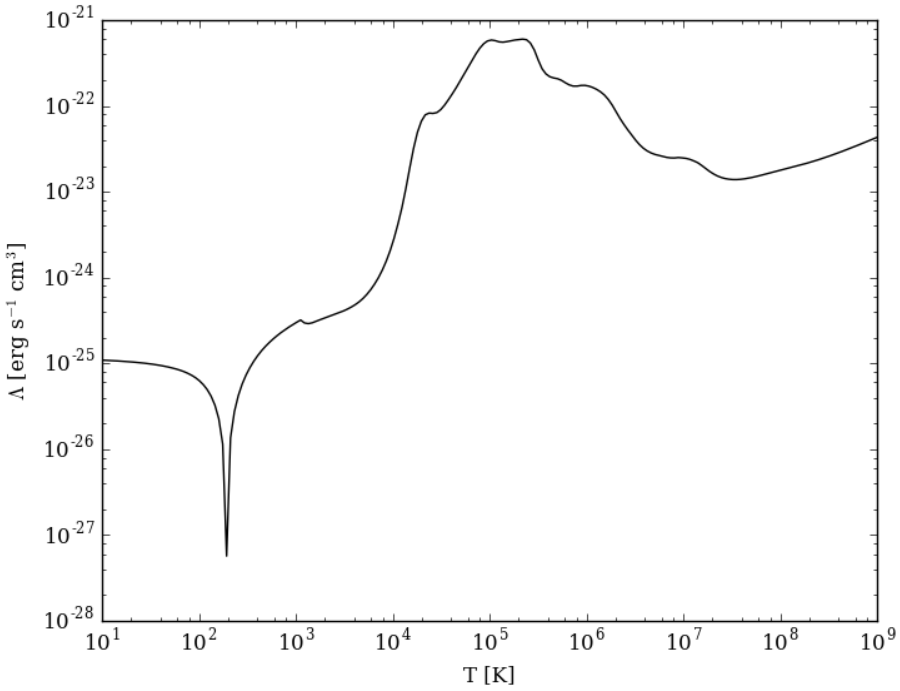
6.2 Running the Example Scripts

A number of example scripts are available in the **src/python/examples** directory. These scripts provide examples of ways that Grackle can be used in simplified models, such as solving the temperature evolution of a parcel of gas at constant density or in a free-fall model. Each example will produce a figure as well as a dataset that can be loaded and analyzed with yt.

6.2.1 Cooling Rate Figure Example

This sets up a one-dimensional grid at a constant density with logarithmically spaced temperatures from 10 K to 10^9 K. Radiative cooling is disabled and the chemistry solver is iterated until the species fractions have converged. The cooling time is then calculated and used to compute the cooling rate.

```
python cooling_rate.py
```



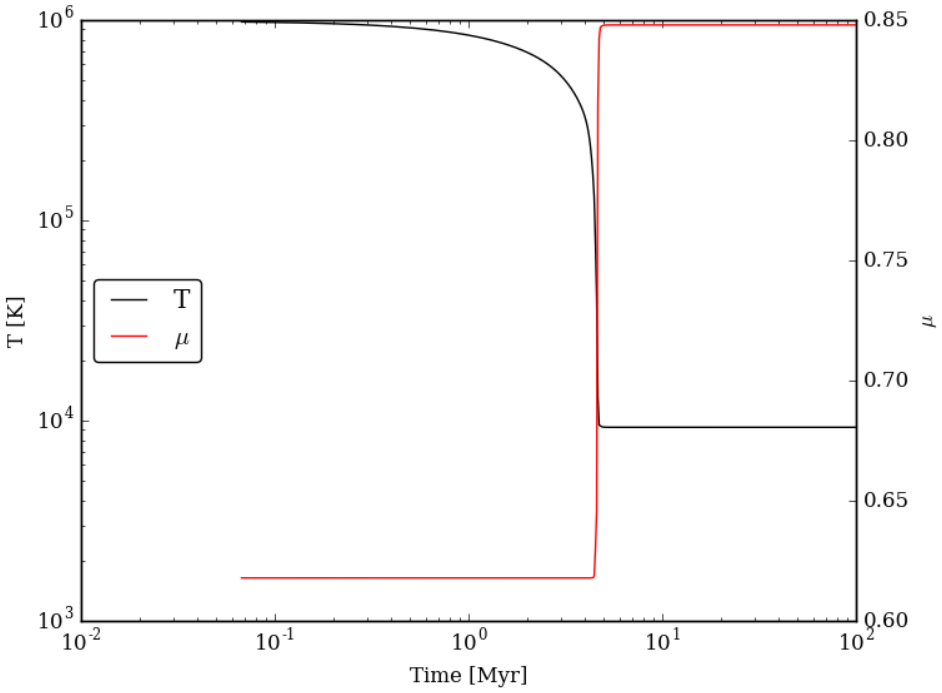
After the script runs, and hdf5 file will be created with a similar name. This can be loaded in with yt.

```
>>> import yt
>>> ds = yt.load("cooling_rate.h5")
>>> print ds.data["temperature"]
[  1.00000000e+01   1.09698580e+01   1.20337784e+01   1.32008840e+01, ...,
   7.57525026e+08   8.30994195e+08   9.11588830e+08   1.00000000e+09] K
>>> print ds.data["cooling_rate"]
[  1.09233398e-25   1.08692516e-25   1.08117583e-25   1.07505345e-25, ...,
   3.77902570e-23   3.94523273e-23   4.12003667e-23   4.30376998e-23] cm**3*erg/s
```

6.2.2 Cooling Cell Example

This sets up a single grid cell with an initial density and temperature and solves the chemistry and cooling for a given amount of time. The resulting dataset gives the values of the densities, temperatures, and mean molecular weights for all times.

```
python cooling_cell.py
```

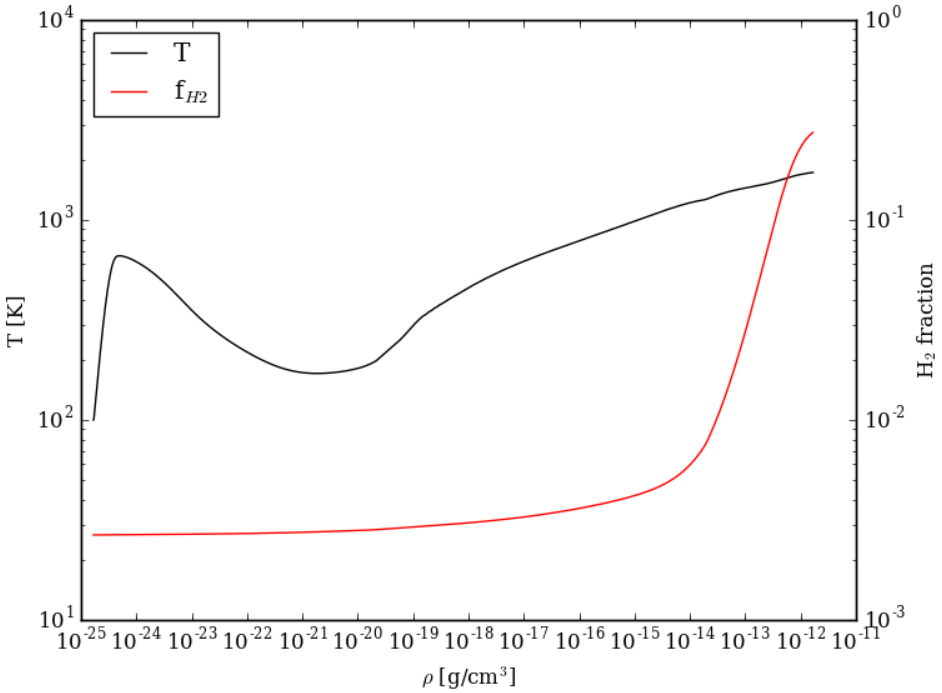



```
>>> import yt
>>> ds = yt.load("cooling_cell.h5")
>>> print ds.data["time"].to("Myr")
YTArray([  0.00000000e+00,   6.74660169e-02,   1.34932034e-01, ...,
         9.98497051e+01,   9.99171711e+01,   9.99846371e+01]) Myr
>>> print ds.data["temperature"]
YTArray([ 990014.56406726,  980007.32720091,  969992.99066987, ...,
          9263.81515866,    9263.81515824,    9263.81515865]) K
```

6.2.3 Free-Fall Collapse Example

This sets up a single grid cell with an initial number density of 1 cm^{-3}. The density increases with time following a free-fall collapse model. As the density increases, thermal energy is added to model heating via adiabatic compression. This can be useful for testing chemistry networks over a large range in density.

```
python freefall.py
```



The resulting dataset can be analyzed similarly as above.

```
>>> import yt
>>> ds = yt.load("freefall.h5")
>>> print ds.data["time"].to("Myr")
[   0.            0.45900816    0.91572127 ...,  219.90360841  219.90360855
  219.9036087 ] Myr
>>> print ds.data["density"]
[  1.67373522e-25   1.69059895e-25   1.70763258e-25 ...,   1.65068531e-12
   1.66121253e-12   1.67178981e-12] g/cm**3
>>> print ds.data["temperature"]
[   99.94958248   100.61345564   101.28160228 ...,  1728.89321898
  1729.32604568  1729.75744287] K
```

6.2.4 Simulation Dataset Example

This provides an example of using the grackle library for calculating chemistry and cooling quantities for a pre-existing simulation dataset. To run this example, you must also download the *IsolatedGalaxy* dataset from the yt sample data page.

```
python run_from_yt.py
```

CHAPTER 7

Grackle Community Code of Conduct

The community of participants in open source scientific projects is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences success and continued growth. We expect everyone in our community to follow these guidelines when interacting with others both inside and outside of our community. Our goal is to keep ours a positive, inclusive, successful, and growing community.

As members of the community,

- We pledge to treat all people with respect and provide a harassment- and bullying-free environment, regardless of sex, sexual orientation and/or gender identity, disability, physical appearance, body size, race, nationality, ethnicity, and religion. In particular, sexual language and imagery, sexist, racist, or otherwise exclusionary jokes are not appropriate.
- We pledge to respect the work of others by recognizing acknowledgment/citation requests of original authors. As authors, we pledge to be explicit about how we want our own work to be cited or acknowledged.
- We pledge to welcome those interested in joining the community, and realize that including people with a variety of opinions and backgrounds will only serve to enrich our community. In particular, discussions relating to pros/cons of various technologies, programming languages, and so on are welcome, but these should be done with respect, taking proactive measure to ensure that all participants are heard and feel confident that they can freely express their opinions.
- We pledge to welcome questions and answer them respectfully, paying particular attention to those new to the community. We pledge to provide respectful criticisms and feedback in forums, especially in discussion threads resulting from code contributions.
- We pledge to be conscientious of the perceptions of the wider community and to respond to criticism respectfully. We will strive to model behaviors that encourage productive debate and disagreement, both within our community and where we are criticized. We will treat those outside our community with the same respect as people within our community.
- We pledge to help the entire community follow the code of conduct, and to not remain silent when we see violations of the code of conduct. We will take action when members of our community violate this code such as contacting grackle.confidential@gmail.com (all emails sent to this address will be treated with the strictest confidence) or talking privately with the person.

This code of conduct applies to all community situations online and offline, including mailing lists, forums, social media, conferences, meetings, associated social events, and one-to-one interactions.

The Grackle Community Code of Conduct was adapted from the Astropy Community Code of Conduct, which was partially inspired by the PSF code of conduct.

CHAPTER 8

How to Develop Grackle

Grackle is a community project!

We are very happy to accept patches, features, and bugfixes from any member of the community! Grackle is developed using mercurial, primarily because it enables very easy and straightforward submission of changesets. We're eager to hear from you.

Note: If you already know how to use the mercurial version control system and are comfortable with handling it yourself, the quickest way to contribute to Grackle is to fork us on BitBucket, make your changes, push the changes to your fork and issue a pull request. The rest of this document is just an explanation of how to do that.

Keep in touch, and happy hacking!

8.1 Open Issues

If you're interested in participating in Grackle development, take a look at the issue tracker on bitbucket. If you are encountering a bug that is not already tracked there, please open a new issue.

8.2 Submitting Changes

We provide a brief introduction to submitting changes here. We encourage contributions from any user. While we do not discuss version control, mercurial or the advanced usage of BitBucket in detail here, we do provide an outline of how to submit changes and we are happy to provide further assistance or guidance on the mailing list.

8.2.1 Licensing

Grackle is under the Enzo public license, a BSD-like license.

All contributed code must be BSD-compatible. If you'd rather not license in this manner, but still want to contribute, please consider creating an external package, which we'll happily link to in the Grackle documentation.

8.2.2 How To Get The Source Code For Editing

Grackle is hosted on BitBucket, and you can see all of the Grackle repositories at http://bitbucket.org/grackle/. In order to modify the source code for Grackle, we ask that you make a "fork" of the main Grackle repository on bitbucket. A fork is simply an exact copy of the main repository (along with its history) that you will now own and can make modifications as you please. You can create a personal fork by visiting the Grackle bitbucket webpage at https://bitbucket.org/grackle/grackle/ . After logging in, you should see an option near the top right labeled "fork". Click this option, and then click the fork repository button on the subsequent page. You now have a forked copy of the Grackle repository for your own personal modification.

This forked copy exists on the bitbucket repository, so in order to access it locally, follow the instructions at the top of that webpage for that forked repository, namely run at a local command line:

```
$ hg clone http://bitbucket.org/<USER>/<REPOSITORY_NAME>
```

This downloads that new forked repository to your local machine, so that you can access it, read it, make modifications, etc. It will put the repository in a local directory of the same name as the repository in the current working directory. You should also run the following command, to make sure you have the most up-to-date version of Grackle checked out in your working directory.

```
$ hg update default
```

You can see any past state of the code by using the hg log command. For example, the following command would show you the last 5 changesets (modifications to the code) that were submitted to that repository.

```
$ cd <REPOSITORY_NAME>
$ hg log -l 5
```

Using the revision specifier (the number or hash identifier next to each changeset), you can update the local repository to any past state of the code (a previous changeset or version) by executing the command:

```
$ hg update revision_specifier
```

8.3 How to Use Mercurial with Grackle

If you're new to Mercurial, these three resources are pretty great for learning the ins and outs:

- http://hginit.com/
- http://book.mercurial-scm.org
- http://mercurial-scm.org/
- http://mercurial-scm.org/wiki

The commands that are essential for using mercurial include:

- `hg help` which provides help for any mercurial command. For example, you can learn more about the `log` command by doing `hg help log`. Other useful topics to use with `hg help` are `hg help glossary`, `hg help config`, `hg help extensions`, and `hg help revsets`.
- `hg commit` which commits changes in the working directory to the repository, creating a new "changeset object."

- `hg add` which adds a new file to be tracked by mercurial. This does not change the working directory.
- `hg pull` which pulls (from an optional path specifier) changeset objects from a remote source. The working directory is not modified.
- `hg push` which sends (to an optional path specifier) changeset objects to a remote source. The working directory is not modified.
- `hg log` which shows a log of all changeset objects in the current repository. Use `-G` to show a graph of changeset objects and their relationship.
- `hg update` which (with an optional "revision" specifier) updates the state of the working directory to match a changeset object in the repository.
- `hg merge` which combines two changesets to make a union of their lines of development. This updates the working directory.

We are happy to asnswers questions about mercurial use on on the mailing list to walk you through any troubles you might have. Here are some general suggestions for using mercurial:

- Named branches are to be avoided. Try using bookmarks (`see hg help bookmark`) to track work. (More info about bookmarks is available on the mercurial wiki)
- Make sure you set a username in your `~/.hgrc` before you commit any changes! All of the tutorials above will describe how to do this as one of the very first steps.
- Please avoid deleting your Grackle forks, as that deletes the pull request discussion from process from BitBucket's website, even if your pull request is merged.
- You should only need one fork. See *Making and Sharing Changes* for a description of the basic workflow.

8.4 Making and Sharing Changes

The simplest way to submit changes to Grackle is to do the following:

- Build Grackle from the mercurial repository
- Navigate to the root of the Grackle repository
- Make some changes and commit them
- Fork the Grackle repository on BitBucket
- Push the changesets to your fork
- Issue a pull request.

Here's a more detailed flowchart of how to submit changes.

1. Edit the source file you are interested in and test your changes.
2. Fork Grackle on BitBucket. (This step only has to be done once.) You can do this at: https://bitbucket.org/grackle/grackle/fork. Call this repository grackle.
3. Create a bookmark to track your work. For example: `hg bookmark my-first-pull-request`
4. Commit these changes, using `hg commit`. This can take an argument which is a series of filenames, if you have some changes you do not want to commit.
5. Remember that this is a large development effort and to keep the code accessible to everyone, good documentation is a must. Add in source code comments for what you are doing. Add in docstrings if you are adding a new function or class or keyword to a function. Add documentation to the appropriate section of the online docs so that people other than yourself know how to use your new code.

6. If your changes include new functionality or cover an untested area of the code, add a test. Commit these changes as well.
7. Push your changes to your new fork using the command:

```
hg push -B my-first-pull-request https://bitbucket.org/YourUsername/grackle/
```

   Where you should substitute the name of the bookmark you are working on for `my-first-pull-request`. If you end up doing considerable development, you can set an alias in the file `.hg/hgrc` to point to this path.

   Note: Note that the above approach uses HTTPS as the transfer protocol between your machine and BitBucket. If you prefer to use SSH - or perhaps you're behind a proxy that doesn't play well with SSL via HTTPS - you may want to set up an SSH key on BitBucket. Then, you use the syntax `ssh://hg@bitbucket.org/YourUsername/grackle`, or equivalent, in place of `https://bitbucket.org/YourUsername/grackle` in Mercurial commands. For consistency, all commands we list in this document use the HTTPS protocol.

8. Issue a pull request at https://bitbucket.org/YourUsername/grackle/pull-request/new A pull request is an automated way of asking people to review and accept the modifications you have made to your personal version of the code.

During the course of your pull request you may be asked to make changes. These changes may be related to style issues, correctness issues, or requesting tests. The process for responding to pull request code review is relatively straightforward.

1. Make requested changes, or leave a comment on the pull request page on Bitbucket indicating why you don't think they should be made.
2. Commit those changes to your local repository.
3. Push the changes to your fork:

   hg push https://bitbucket.org/YourUsername/grackle/

4. Your pull request will be automatically updated.

CHAPTER 9

Help

If you have any questions, please join the Grackle Users Google Group. Feel free to post any questions or ideas for development.

CHAPTER 10

Contributing

Development of Grackle happens in the open on Bitbucket here. We welcome new contributors. Please, see the *Grackle Community Code of Conduct*. For a guide to developing Grackle, see *How to Develop Grackle*.

CHAPTER 11

Citing grackle

The Grackle method paper can be found here.

The Grackle library was born out of the chemistry and cooling routines of the Enzo simulation code. As such, all of those who have contributed to Enzo development, and especially to the chemistry and cooling, have contributed to the Grackle.

If you used the Grackle library in your work, please cite it as "the Grackle chemistry and cooling library (Smith et al. 2017)." Also, please add a footnote to https://grackle.readthedocs.io/.

CHAPTER 12

Search

- search

Index

S

T

U

V

W

X

Y

Z