
grackle Documentation

Release 2.0

October 01, 2014

1	Installation	3
1.1	Dependencies	3
1.2	Downloading	3
1.3	Building	3
2	Adding Grackle to Your Simulation Code	7
2.1	Example Executables	7
2.2	Header Files	8
2.3	Data Types	8
2.4	Enabling Output	8
2.5	Code Units	9
2.6	Chemistry Data	9
2.7	Creating the Necessary Fields	10
2.8	Calling the Available Functions	11
2.9	Pure Tabulated Mode	13
3	Parameters and Data Files	15
3.1	Parameters	15
3.2	Data Files	16
4	API Reference	19
4.1	Functions using structs (best for C and C++)	19
4.2	Functions without structs (best for Fortran)	24
4.3	Internal Functions	26
5	The Python Examples	29
5.1	Installing the Python Wrappers	29
5.2	Running the Example Scripts	29
6	Help	33
7	Citing grackle	35
8	Search	37

Grackle is a chemistry and radiative cooling library for astrophysical simulations with interfaces for C, C++, and Fortran codes. It is a generalized and trimmed down version of the chemistry network of the [Enzo](#) simulation code. Grackle provides:

- two options for primordial chemistry and cooling:
 1. non-equilibrium primordial chemistry network for atomic H, D, and He as well as H₂ and HD, including H₂ 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 \(2012\)](#).

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 (γ).

Contents:

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.

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. Starting with an Enzo make file, you will likely have to add to the following variables:

- MACH_CPPFLAGS - -fPIC

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

- 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, optimization, and building static or dynamic libraries. 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_SHARED    [shared-{yes,no}]              : yes
```

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 coolld_multi.F
....

Linking
Success!
```

Then, to install:

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

Now it's time to integrate grackle into your simulation code: [Adding Grackle to Your Simulation Code](#)

Adding Grackle to Your Simulation Code

This document follows the example files, **cxx_example.C** and **cxx_table_example.C**. For a list of all available functions, see the [API Reference](#).

2.1 Example Executables

The grackle source code contains examples for C, C++, and Fortran codes. They are located in the **src/example** directory and detail different uses of the grackle library.

- **c_example.c** - full functionality C example that uses the units and chemistry data structures.
- **c_table_example.c** - tabulated cooling only (no chemistry) C example that uses the units and chemistry data structures.
- **c_example_nostruct.c** - full functionality C example that uses the `initialize_grackle_()` function instead of data structures.
- **c_table_example_nostruct.c** - tabulated cooling only (no chemistry) C example that uses the `initialize_grackle_()` function instead of data structures.
- **cxx_example.C** - full functionality C++ example that uses the units and chemistry data structures.
- **cxx_table_example.C** - tabulated cooling only (no chemistry) C++ example that uses the units and chemistry data structures.
- **fortran_example.F** - full functionality Fortran example that uses the `initialize_grackle_()` function.
- **fortran_table_example.F** - tabulated cooling only (no chemistry) Fortran example that uses the `initialize_grackle_()` function.

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).

This document follows **cxx_example.C**, which details the use of the full-featured grackle functions. The table examples illustrate the use of the Grackle with fully tabulated cooling functions only. In this mode, a simplified set of functions are available. For information on these, see [Pure Tabulated Mode](#).

2.2 Header Files

Six 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_types.h** - defines the variable type `gr_float` to be used for the baryon fields passed to the grackle functions. This can be either a 4 or 8 byte float, allowing the code to be easily configured for either single or double precision baryon fields.
- **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_macros.h** - contains some macros used internally.
- **chemistry_data.h** - defines the primary data structure which all run time parameters as well as the chemistry, cooling, and UV background data.
- **code_units.h** - defines the structure containing conversions from code units to CGS.

The only source file that needs to be included in your simulation code is **grackle.h**. Since this is a C++ example and the Grackle is pure C, we must surround the include with the ‘extern “C”’ directive.

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

2.3 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*.

2.4 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.

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

2.5 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 $a_{\text{true}} = a_{\text{code}} * a_{\text{units}}$.

```
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
```

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.

2.6 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 as well as all of the actual chemistry and cooling rate data. The routine, `set_default_chemistry_parameters` is responsible for the initial setup of this structure and for setting of all the default parameter values. The parameters can then be set to their desired values. See [Parameters and Data Files](#) for a full list of the available parameters. The function will return an integer indicating success (1) or failure (0).

chemistry_data

This structure holds all grackle run time parameter and all chemistry and cooling data arrays.

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

// Set parameter values for chemistry.
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 with the *initialize_chemistry_data*. This function also requires the initial value of the expansion factor for setting internal units. If the simulation is not cosmological, the expansion factor should be set to 1. The initializing function will return an integer indicating success (1) or failure (0).

```
// Set initial expansion factor (for internal units).
// Set expansion factor to 1 for non-cosmological simulation.
double initial_redshift = 100.;
double a_value = 1. / (1. + initial_redshift) / my_units.a_units;

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

The Grackle is now ready to be used.

2.7 Creating the Necessary Fields

With the *code_units* and *chemistry_data* structures ready, the only thing left is to create the arrays to carry the species densities. Pointers for all fields must be created, but the arrays only need to be allocated if the fields are going to be used by the chemistry network. Variables containing the dimensionality of the data, the active dimensions (not including the ghost zones), and the starting and ending indices for each dimensions must also be created.

```
// Allocate field arrays.
gr_float *density, *energy, *x_velocity, *y_velocity, *z_velocity,
    *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;

// Set grid dimension and size.
// grid_start and grid_end are used to ignore ghost zones.
int field_size = 10;
int grid_rank = 3;
// If grid rank is less than 3, set the other dimensions to 1 and
// start indices and end indices to 0.
int grid_dimension[3], grid_start[3], grid_end[3];
for (int i = 0; i < 3; i++) {
    grid_dimension[i] = 1; // the active dimension not including ghost zones.
```

```

    grid_start[i] = 0;
    grid_end[i] = 0;
}
grid_dimension[0] = field_size;
grid_end[0] = field_size - 1;

density      = new gr_float[field_size];
energy       = new gr_float[field_size];
x_velocity   = new gr_float[field_size];
y_velocity   = new gr_float[field_size];
z_velocity   = new gr_float[field_size];
// for primordial_chemistry >= 1
HI_density   = new gr_float[field_size];
HII_density  = new gr_float[field_size];
HeI_density  = new gr_float[field_size];
HeII_density = new gr_float[field_size];
HeIII_density = new gr_float[field_size];
e_density    = new gr_float[field_size];
// for primordial_chemistry >= 2
HM_density   = new gr_float[field_size];
H2I_density  = new gr_float[field_size];
H2II_density = new gr_float[field_size];
// for primordial_chemistry >= 3
DI_density   = new gr_float[field_size];
DII_density  = new gr_float[field_size];
HDI_density  = new gr_float[field_size];
// for metal_cooling = 1
metal_density = 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.

2.8 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, the field size and dimension variables, and the field arrays themselves. In some cases, the current value of the expansion factor must also be given and for the chemistry solving routine, a timestep must 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.

2.8.1 Solve the Chemistry and Cooling

```

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

if (solve_chemistry(&my_units,
                    a_value, dt,
                    grid_rank, grid_dimension,
                    grid_start, grid_end,
                    density, energy,
                    x_velocity, y_velocity, z_velocity,
                    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) == 0) {
    fprintf(stderr, "Error in solve_chemistry.\n");
    return 0;
}
```

2.8.2 Calculating the Cooling Time

```
gr_float *cooling_time;
cooling_time = new gr_float[field_size];
if (calculate_cooling_time(&my_units,
    a_value,
    grid_rank, grid_dimension,
    grid_start, grid_end,
    density, energy,
    x_velocity, y_velocity, z_velocity,
    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,
    cooling_time) == 0) {
    fprintf(stderr, "Error in calculate_cooling_time.\n");
    return 0;
}
```

2.8.3 Calculating the Temperature Field

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

2.8.4 Calculating the Pressure Field

```
gr_float *pressure;
pressure = new gr_float[field_size];
if (calculate_pressure(&my_units,
    grid_rank, grid_dimension,
    density, energy,
    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,
        pressure) == 0) {
    fprintf(stderr, "Error in calculate_pressure.\n");
    return 0;
}

```

2.8.5 Calculating the Gamma Field

```

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

```

2.9 Pure Tabulated Mode

If you only intend to run simulations using the fully tabulated cooling (*primordial_chemistry* set to 0), then a simplified set of functions are available. These functions do not require pointers to be given for the field arrays for the chemistry species densities. See the `cxx_table_example.C`, `c_table_example.c`, `c_table_example_nostruct.c`, and `fortran_table_example.F` files in the `src/example` directory for examples.

Note: No simplified function is available for the calculation of the gamma field since gamma is only altered in Grackle by the presence of H_2 .

2.9.1 Solve the Cooling

```

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

if (solve_chemistry(&my_units,
    a_value, dt,
    grid_rank, grid_dimension,
    grid_start, grid_end,
    density, energy,
    x_velocity, y_velocity, z_velocity,
    metal_density) == 0) {
    fprintf(stderr, "Error in solve_chemistry.\n");
    return 0;
}

```

2.9.2 Calculating the Cooling Time

```
gr_float *cooling_time;
cooling_time = new gr_float[field_size];
if (calculate_cooling_time(&my_units,
                        a_value,
                        grid_rank, grid_dimension,
                        grid_start, grid_end,
                        density, energy,
                        x_velocity, y_velocity, z_velocity,
                        metal_density,
                        cooling_time) == 0) {
    fprintf(stderr, "Error in calculate_cooling_time.\n");
    return 0;
}
```

2.9.3 Calculating the Temperature Field

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

2.9.4 Calculating the Pressure Field

```
gr_float *pressure;
pressure = new gr_float[field_size];
if (calculate_pressure(&my_units,
                    grid_rank, grid_dimension,
                    density, energy,
                    pressure) == 0) {
    fprintf(stderr, "Error in calculate_pressure.\n");
    return 0;
}
```

Parameters and Data Files

3.1 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. A simplified set of functions are available (though not required) for use in this mode. For more information, see *Pure Tabulated Mode*.
- 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 H₂⁺ channels, three-body formation (H+H+H and H+H+H₂), H₂ rotational transitions, chemical heating, and collision-induced emission (optional). Active species: above + H⁻, H₂, H₂⁺.
- 3: 12-species network include all above plus HD rotation cooling. Active species: above plus 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 H₂ 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 `c:data:grackle_data_file` parameter. Default: 0.

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 \(2008\)](#). These 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 $\text{erg cm}^{-3} \text{s}^{-1}$. Default: $8.5\text{e-}26$.

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 $10^{-21} \text{ erg s}^{-1} \text{ cm}^{-2} \text{ Hz}^{-1} \text{ sr}^{-1}$. 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.

3.2 Data Files

These files contain the metal heating and cooling rates and the UV background photo-heating and photo-ionization rates. For all three files, the number density range is $-10 < \log_{10} (n_{\text{H}} / \text{cm}^{-3}) < 4$ and the temperature range is $1 < \log_{10} (T / \text{K}) < 9$. Extrapolation is performed when outside of the data range. All data files are located in the **input** directory in the source.

- **CloudyData_noUVB.h5** - metal cooling rates for collisional ionization equilibrium.
- **CloudyData_UVB=FG2011.h5** - metal heating and cooling rates and UV background rates from the work of [Faucher-Giguere et. al. \(2009\)](#), updated in 2011. The maximum redshift is 10.6. Above that, collisional ionization equilibrium is assumed.

- **CloudyData_UVB=HM2012.h5** - metal 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.

API Reference

The Grackle has a few different versions of the various functions for solving the chemistry and cooling and calculating related fields. One set of functions requires the user to work with C structs, while the other does not. The set that does not use structs is simpler to implement in Fortran codes. Both of these rely internally on a *chemistry_data* type struct called `grackle_data`, which exists in the `grackle` namespace. A third set of functions also exists that requires the user to hold and pass their own *chemistry_data* struct.

4.1 Functions using structs (best for C and C++)

These functions require the user to directly access the `grackle_data` data structure to set parameters and to create a `code_units` struct to control the unit system. These functions are used in the examples, `c_example.c`, `c_table_example.c`, `cxx_example.C`, and `cxx_table_example.C`.

int set_default_chemistry_parameters();

Initializes the `grackle_data` data structure. This must be called before run time parameters can be set.

Return type int

Returns 1 (success) or 0 (failure)

int initialize_chemistry_data(code_units *my_units, double a_value);

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
- **a_value** (*double*) – the expansion factor in code units ($a_{\text{code}} = a / a_{\text{units}}$)

Return type int

Returns 1 (success) or 0 (failure)

int solve_chemistry(code_units *my_units, double a_value, double dt_value, int grid_rank, ...)

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
- **a_value** (*double*) – the expansion factor in code units ($a_{\text{code}} = a / a_{\text{units}}$)
- **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 H₂: densities in code units equivalent those of the density array. Used with `primordial_chemistry >= 2`.
- **H2II_density** (*gr_float**) – array containing the H₂⁺ 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(code_units *my_units, double a_value, int grid_rank, int *grid_dimension, int *grid_start, int *grid_end, gr_float *density, gr_float *internal_energy, gr_float *x_velocity, gr_float *y_velocity, gr_float *z_velocity, gr_float *HI_density, gr_float *HII_density, gr_float *HM_density, gr_float *HeI_density, gr_float *HeII_density, gr_float *HeIII_density, gr_float *H2I_density, gr_float *H2II_density, gr_float *DI_density, gr_float *DII_density, gr_float *HDI_density, gr_float *e_density, gr_float *metal_density, gr_float *cooling_time)

Calculates the instantaneous cooling time.

Parameters

- **my_units** (*code_units**) – code units conversions
- **a_value** (*double*) – the expansion factor in code units ($a_{\text{code}} = a / a_{\text{units}}$)
- **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, 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(code_units *my_units, int grid_rank, int *grid_dimension, gr_float *density, gr_float *internal_energy, gr_float *x_velocity, gr_float *y_velocity, gr_float *z_velocity, gr_float *HI_density, gr_float *HII_density, gr_float *HM_density, gr_float *HeI_density, gr_float *HeII_density, gr_float *HeIII_density, gr_float *H2I_density, gr_float *H2II_density, gr_float *DI_density, gr_float *DII_density, gr_float *HDI_density, gr_float *e_density, gr_float *metal_density, gr_float *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
- **a_value** (*double*) – the expansion factor in code units ($a_{\text{code}} = a / a_{\text{units}}$)
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension
- **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(code_units *my_units, int grid_rank, int *grid_dimension, gr_float *pressure)
Calculates the gas pressure.

Parameters

- **my_units** (*code_units**) – code units conversions
- **a_value** (*double*) – the expansion factor in code units ($a_{\text{code}} = a / a_{\text{units}}$)
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension
- **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(code_units *my_units, int grid_rank, int *grid_dimension, gr_float *temperature)
Calculates the gas temperature.

Parameters

- **my_units** (*code_units**) – code units conversions
- **a_value** (*double*) – the expansion factor in code units ($a_{\text{code}} = a / a_{\text{units}}$)
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension
- **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)

4.1.1 Tabular-Only Functions

These are slimmed down functions that require `primordial_chemistry = 0` and use only the tabulated cooling rates (no chemistry).

int solve_chemistry_table(code_units *my_units, double a_value, double dt_value, int grid_rank, int grid_dimension, int* grid_start, int* grid_end, gr_float* density, gr_float* internal_energy, gr_float* x_velocity, gr_float* y_velocity, gr_float* z_velocity, gr_float* metal_density)
 Evolves the internal energies over a given timestep by solving the cooling rate equations. This version allows only for the use of the tabulated cooling functions.

Parameters

- **my_units** (*code_units**) – code units conversions
- **a_value** (*double*) – the expansion factor in code units ($a_{\text{code}} = a / a_{\text{units}}$)
- **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
- **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_table(code_units *my_units, double a_value, int grid_rank, int grid_dimension, int* grid_start, int* grid_end, gr_float* density, gr_float* internal_energy, gr_float* x_velocity, gr_float* y_velocity, gr_float* z_velocity)
 Calculates the instantaneous cooling time. This version allows only for the use of the tabulated cooling functions.

Parameters

- **my_units** (*code_units**) – code units conversions
- **a_value** (*double*) – the expansion factor in code units ($a_{\text{code}} = a / a_{\text{units}}$)
- **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, y_velocity, z_velocity** (*gr_float**) – arrays containing the x, y, and z velocity values in code units

- **metal_density** (*gr_float**) – array containing the metal densities in code units equivalent those of the density array. Used with `metal_cooling = 1`.
- **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_pressure_table(code_units *my_units, int grid_rank, int *grid_dimension, gr_float *

Calculates the gas pressure. This version allows only for the use of the tabulated cooling functions.

Parameters

- **my_units** (*code_units**) – code units conversions
- **a_value** (*double*) – the expansion factor in code units ($a_{code} = a / a_{units}$)
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension
- **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*
- **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_table(code_units *my_units, int grid_rank, int *grid_dimension, gr_float *

Calculates the gas temperature. This version allows only for the use of the tabulated cooling functions.

Parameters

- **my_units** (*code_units**) – code units conversions
- **a_value** (*double*) – the expansion factor in code units ($a_{code} = a / a_{units}$)
- **grid_rank** (*int*) – the dimensionality of the grid
- **grid_dimension** (*int**) – array holding the size of the baryon field in each dimension
- **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*
- **pressure** (*gr_float**) – array which will be filled with the calculated pressure values

Return type int

Returns 1 (success) or 0 (failure)

4.2 Functions without structs (best for Fortran)

These functions do not use any structs and are therefore much simpler to implement in Fortran codes. These are used in the example files, `c_example_nostruct.c`, `c_table_example_nostruct.c`, `fortran_example.F`, and `fortran_table_example.F`.

Note: In Fortran codes, these should be called without the trailing underscore. The variable types can be mapped to Fortran as: `int*` becomes `integer`, `double*` becomes `real*8`, and `gr_float*` becomes `R_PREC`.

```
int initialize_grackle(int *comoving_coordinates, double *density_units, double *length_units)
    Initializes the grackle data structures and associated chemistry and cooling data. This performs the operations
    of both set_default_chemistry_parameters() and initialize_chemistry_data().
```

Parameters

- **comoving_coordinates** (*int**) – `comoving_coordinates` parameter
- **density_units** (*double**) – `density_units` conversion factor
- **length_units** (*double**) – `length_units` conversion factor
- **time_units** (*double**) – `time_units` conversion factor
- **velocity_units** (*double**) – `velocity_units` conversion factor
- **a_units** (*double**) – `a_units` conversion factor
- **a_value** (*double**) – expansion factor in code units (`a_code = a / a_units`)
- **use_grackle** (*int**) – `use_grackle` parameter
- **with_radiative_cooling** (*int**) – `with_radiative_cooling` parameter
- **grackle_file** (*char**) – `grackle_data_file` parameter
- **primordial_chemistry** (*int**) – `primordial_chemistry` parameter
- **metal_cooling** (*int**) – `metal_cooling` parameter
- **UVbackground** (*int**) – `UVbackground` parameter
- **h2_on_dust** (*int**) – `h2_on_dust` parameter
- **cmb_temperature_floor** (*int**) – `cmb_temperature_floor` parameter
- **gamma** (*double**) – `Gamma` parameter

Note: The last argument should be omitted.

```
int solve_chemistry(int *comoving_coordinates, double *density_units, double *length_units)
    Evolves the species densities and internal energies over a given timestep by solving the chemistry and cooling
    rate equations.
```

```
int calculate_cooling_time(int *comoving_coordinates, double *density_units, double *length_units)
    Calculates the instantaneous cooling time.
```

```
int calculate_gamma(int *comoving_coordinates, double *density_units, double *length_units)
    Calculates the effective adiabatic index. This is only useful with primordial_chemistry  $\geq 2$  as the only
    thing that alters gamma from the single value is  $H_2$ .
```

```
int calculate_pressure(int *comoving_coordinates, double *density_units, double *length_units)
    Calculates the gas pressure.
```

```
int calculate_temperature(int *comoving_coordinates, double *density_units, double *length_units)
    Calculates the gas temperature.
```

4.2.1 Tabular-Only Functions

These are slimmed down functions that require `primordial_chemistry = 0` and use only the tabulated cooling rates (no chemistry).

int solve_chemistry_table(int *comoving_coordinates, double *density_units, double *lengthscales, double *timestep)
Evolves the internal energies over a given timestep by solving the cooling rate equations. This version allows only for the use of the tabulated cooling functions.

int calculate_cooling_time_table(int *comoving_coordinates, double *density_units, double *lengthscales, double *timestep)
Calculates the instantaneous cooling time. This version allows only for the use of the tabulated cooling functions.

int calculate_pressure_table(int *comoving_coordinates, double *density_units, double *lengthscales, double *timestep)
Calculates the gas pressure. This version allows only for the use of the tabulated cooling functions.

int calculate_temperature_table(int *comoving_coordinates, double *density_units, double *lengthscales, double *timestep)
Calculates the gas temperature. This version allows only for the use of the tabulated cooling functions.

4.3 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();

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, code_units *my_units, double a_value, double a_code, double a_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_chemistry** (*chemistry_data**) – the structure returned by `_set_default_chemistry_parameters()`
- **my_units** (*code_units**) – code units conversions
- **a_value** (*double*) – the expansion factor in code units ($a_{code} = a / a_{units}$)

Return type int

Returns 1 (success) or 0 (failure)

int _solve_chemistry(chemistry_data *my_chemistry, code_units *my_units, double a_value, double a_code, double a_units)
Evolves the species densities and internal energies over a given timestep by solving the chemistry and cooling rate equations.

int _calculate_cooling_time(chemistry_data *my_chemistry, code_units *my_units, double a_value, double a_code, double a_units)
Calculates the instantaneous cooling time.

int _calculate_gamma(chemistry_data *my_chemistry, code_units *my_units, int grid_rank, int grid_rank_max)
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 H₂.

int _calculate_pressure(chemistry_data *my_chemistry, code_units *my_units, int grid_rank, int grid_rank_max)
Calculates the gas pressure.

int _calculate_temperature(chemistry_data *my_chemistry, code_units *my_units, int grid_rank, int grid_rank_max)
Calculates the gas temperature.

4.3.1 Tabular-Only Functions

These are slimmed down functions that require `primordial_chemistry = 0` and use only the tabulated cooling rates (no chemistry).

- int `_solve_chemistry_table`(chemistry_data *my_chemistry, code_units *my_units, double a_val)**
Evolves the internal energies over a given timestep by solving the cooling rate equations. This version allows only for the use of the tabulated cooling functions.
- int `_calculate_cooling_time_table`(chemistry_data *my_chemistry, code_units *my_units, double a_val)**
Calculates the instantaneous cooling time. This version allows only for the use of the tabulated cooling functions.
- int `_calculate_pressure_table`(chemistry_data *my_chemistry, code_units *my_units, int grid)**
Calculates the gas pressure. This version allows only for the use of the tabulated cooling functions.
- int `_calculate_temperature_table`(chemistry_data *my_chemistry, code_units *my_units, int grid)**
Calculates the gas temperature. This version allows only for the use of the tabulated cooling functions.

The Python Examples

These example works with a python wrapper that calls the various library functions. These require [Cython](#) to be installed. The best thing to do is to install the [yt analysis toolkit](#), which includes Cython.

5.1 Installing the Python Wrappers

After building the grackle library, some additional environment variables must be set. Move into the **src/python** directory and run the **set_libs** script and follow the instructions. Then, run *python setup.py install* to build and install the python wrappers.

```
~/grackle $ cd src/python
~/grackle/src/python $ ./set_libs
Issue the following commands:
export PYTHONPATH=$PYTHONPATH:/grackle/src/python
export LD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/grackle/src/python/./clib
You can also set your LD_LIBRARY_PATH to point to where you installed libgrackle.
~/grackle/src/python $ python setup.py install
running install
running build
running config_cc
[clipped]
running install_clib
customize UnixCCompiler
```

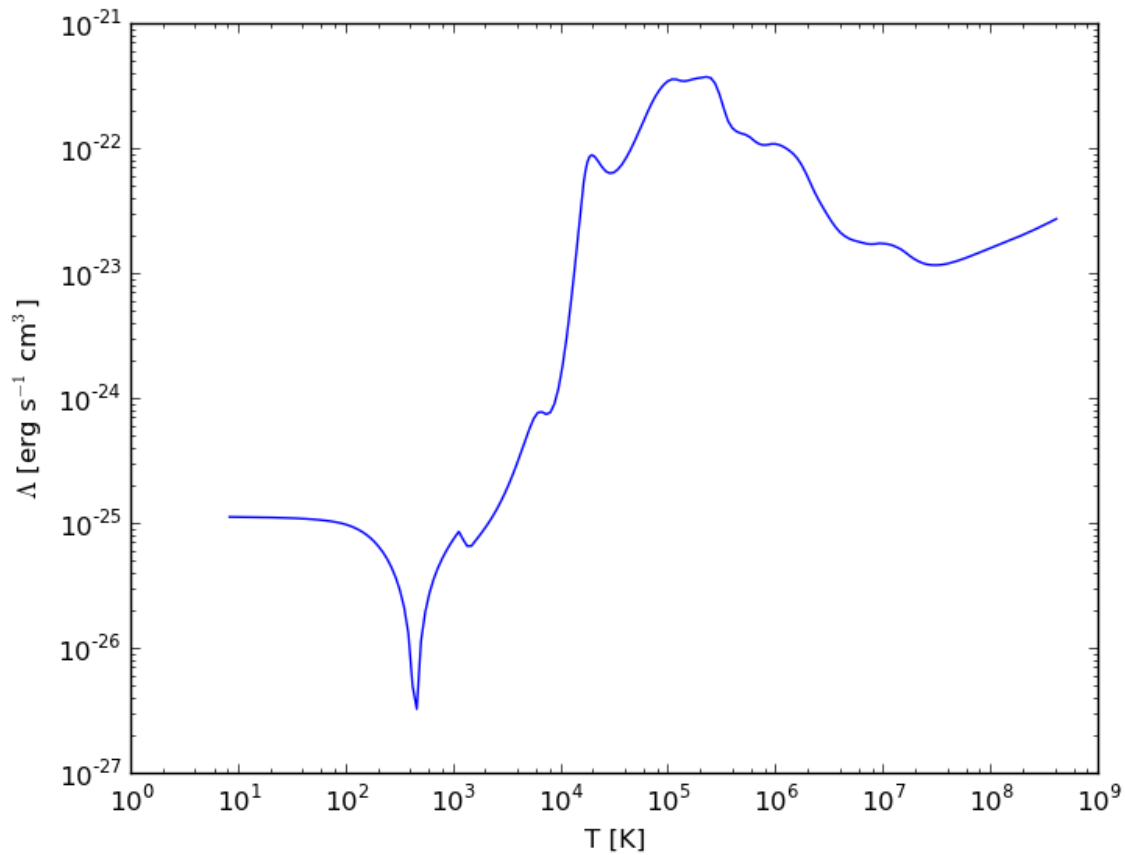
5.2 Running the Example Scripts

The python examples are located in the **src/python/examples** directory. Before running them, make sure to copy the data files from the **input** directory into this directory.

5.2.1 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. This script also provides a good example for setting up cosmological unit systems.

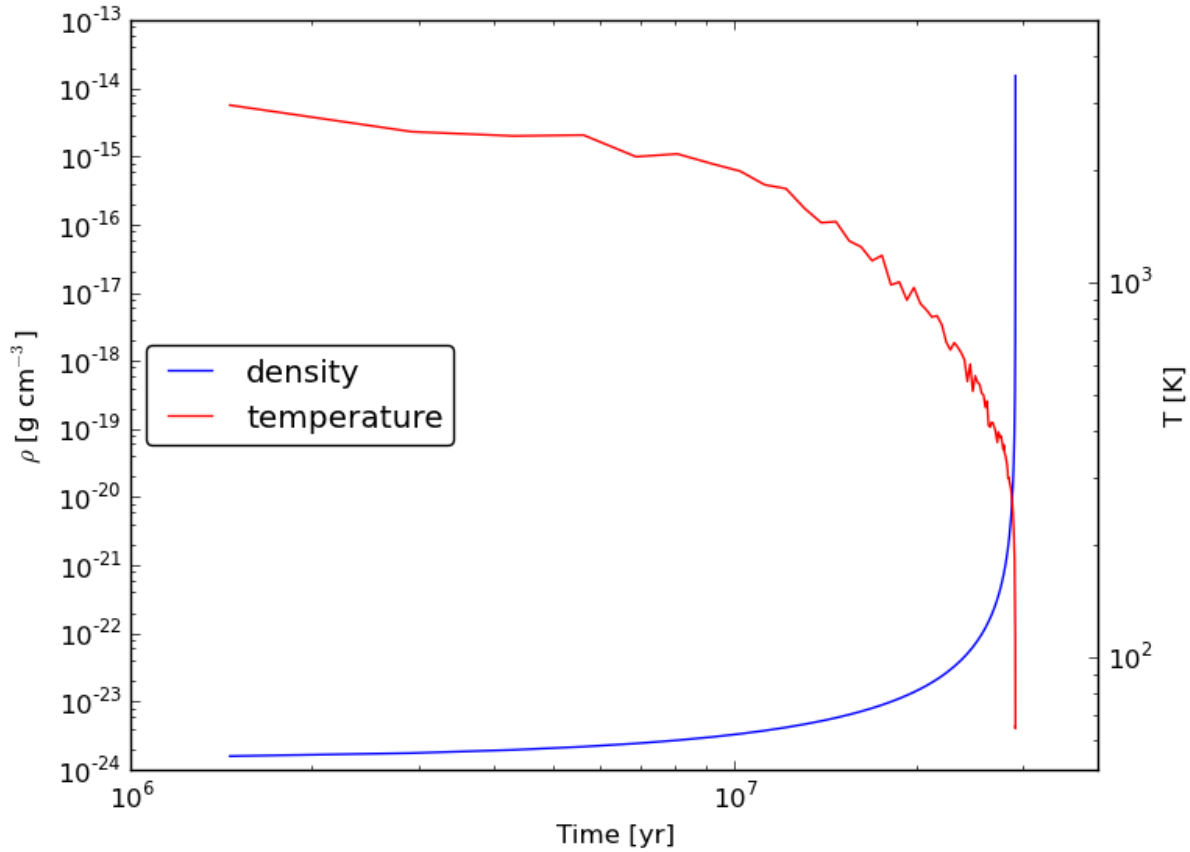
```
python cooling_rate.py
```



5.2.2 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 adiabatic compression heating. This can be useful for testing chemistry networks over a large range in density.

```
python freefall.py
```



5.2.3 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 have [yt](#) installed and must also download the *IsolatedGalaxy* dataset from the [yt sample data page](#).

```
python run_from_yt.py
```

Help

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

Citing grackle

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. There is currently no paper that specifically presents the Grackle library on its own, but the functionality was fully described in the [Enzo method paper](#). The Grackle was originally designed for the [AGORA Project](#) and first referred to by name in the [AGORA method paper](#).

If you used the Grackle library in your work, please cite it as “the Grackle chemistry and cooling library (The [Enzo Collaboration et al. 2014](#); [Kim, J. et al. 2014](#)).” Also, please add a footnote to <https://grackle.readthedocs.org/>.

The Enzo Collaboration, Bryan, G. L., Norman, M. L., et al. 2014, ApJS, 211, 19

Kim, J.-h., Abel, T., Agertz, O., et al. 2014, ApJS, 210, 14

Search

- *search*