grackle Documentation

Release 1.0

January 10, 2014

Contents

Grackle is a chemistry and radiative cooling library for astrophysical simulations. 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 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).

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/brittonsmith/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, you may use it to compile grackle, although you will need to add `-fPIC` as a compile-time flag. 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 floating point and integer variables and for optimization. To see them, type:

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

MACHINE: Darwin (OSX)
MACHINE-NAME: darwin

CONFIG_PRECISION  [precision-{32,64}]                      : 64
CONFIG_INTEGERS  [integers-{32,64}]                        : 64
CONFIG_OPT  [opt-{warn,debug,cudadebug,high,aggressive}]   : debug
```

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 librackle
   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
```

Now it’s time to integrate grackle into your simulation code: *Adding Grackle to Your Simulation Code*

CHAPTER 2

Adding Grackle to Your Simulation Code

2.1 Example Executables

The grackle source code contains two C++ examples that links against the grackle library. They are located in the **src/example** directory and are called **example.C** and **table_example.C**. If you have already installed the grackle library, you can build the examples by typing:

```
$ make example
```

or

```
$ make table_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 **example.C**, which details the use of the full-featured grackle functions. The **table_example.C** file illustrates 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

Four source 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.
- **grackle_macros.h** - this contains basic variable type definitions.
- **chemistry_data.h** - this defines the primary data structure which all run time parameters as well as the chemistry, cooling, and UV background data.
- **code_units.h** - this 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**.

2.3 Data Types

The grackle library provides two variable sized data types, one for integers and one for floating point variables. With **grackle.h** included, both of these data types are available.

- *gr_int* - the integer data type. This type is a 32 bit integer (int) if compiled with *integers-32* and a 64 bit integer (long int) if compiled with *integers-64*.
- *gr_float* - the floating point data type. This type is a 32 bit float (float) if compiled with *precision-32* and a 64 bit float (double) if compiled with *precision-64*.

2.4 Code Units

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, and velocity) must be set. When using the proper frame, *a_units* (units for the expansion factor) must be set to 1.0.

```
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.

2.5 Chemistry Data

The *chemistry_data* structure contains all of the parameters for controlling the behavior of the chemistry and cooling solver. It also contains all of the actual chemistry and cooling rate data. The routine, *set_default_chemistry_parameters* creates the *chemistry_data* structure with the default settings and returns it. The parameters can then be set to their desired values. See *Parameters and Data Files* for a full list of the available parameters.

```
chemistry_data my_chemistry = set_default_chemistry_parameters();
// Set parameter values for chemistry.
my_chemistry.use_grackle = 1;            // chemistry on
my_chemistry.with_radiative_cooling = 1; // cooling on
my_chemistry.primordial_chemistry = 3;   // molecular network with H, He, D
my_chemistry.metal_cooling = 1;          // metal cooling on
my_chemistry.UVbackground = 1;           // UV background on
my_chemistry.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.
gr_float initial_redshift = 100.;
gr_float a_value = 1. / (1. + initial_redshift);

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

The *chemistry_data* structure is now ready to be used.

2.6 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.
gr_int field_size = 10;
gr_int grid_rank = 3;
// If grid rank is less than 3, set the other dimensions,
// start indices, and end indices to 0.
gr_int grid_dimension[3], grid_start[3], grid_end[3];
for (int i = 0;i < 3;i++) {
  grid_dimension[i] = 0; // 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.

2.7 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* and *chemistry_data* structures, 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.

2.7.1 Solve the Chemistry and Cooling

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

if (solve_chemistry(my_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;
}
```

2.7.2 Calculating the Cooling Time

```
gr_float *cooling_time;
cooling_time = new gr_float[field_size];
if (calculate_cooling_time(my_chemistry, 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;
}
```

2.7.3 Calculating the Temperature Field

```
gr_float *temperature;
temperature = new gr_float[field_size];
if (calculate_temperature(my_chemistry, 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;
}
```

2.7.4 Calculating the Pressure Field

```
gr_float *pressure;
pressure = new gr_float[field_size];
if (calculate_pressure(my_chemistry, 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;
}
```

2.7.5 Calculating the Gamma Field

```
gr_float *gamma;
gamma = new gr_float[field_size];
if (calculate_gamma(my_chemistry, 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;
}
```

2.8 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 **table_example.C** file in the **src/example** directory for an example.

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.

2.8.1 Solve the Cooling

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

if (solve_chemistry(my_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;
}
```

2.8.2 Calculating the Cooling Time

```
gr_float *cooling_time;
cooling_time = new gr_float[field_size];
if (calculate_cooling_time(my_chemistry, 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;
}
```

2.8.3 Calculating the Temperature Field

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

CHAPTER 3

Parameters and Data Files

3.1 Parameters

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

`use_grackle` (int) Flag to activate the grackle machinery. Default: 0.

`with_radiative_cooling` (int) 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.

`primordial_chemistry` (int) 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 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 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.

`h2_on_dust` (int)

- 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.

`metal_cooling` (int) 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.

`cmb_temperature_floor` (int) 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.

`UVbackground` (int) 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.

`grackle_data_file` (string) Path to the data file containing the metal cooling and UV background tables. Default: "".

`Gamma` (float) 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.

`three_body_rate` (int) 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.

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

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

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

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

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

`LWbackground_intensity` (float) 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.

`LWbackground_sawtooth_suppression` (int) 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 $-10 < \log_{10} (n_H / cm^{-3}) < 4$ and the temperature range is $1 < \log_{10} (T / K) < 9$. 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 maxmimum 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.

CHAPTER 4

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.

4.1 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
```

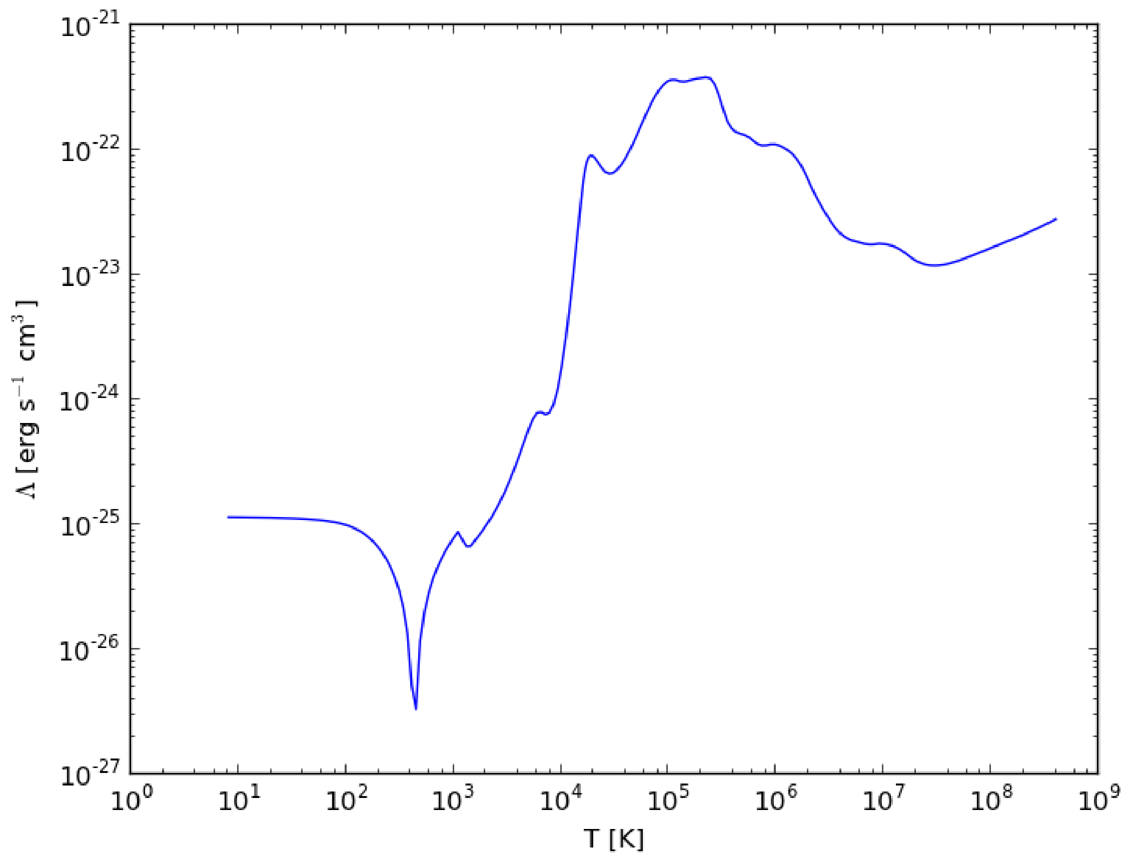
4.2 Running the Example Scripts

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.

4.2.1 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.

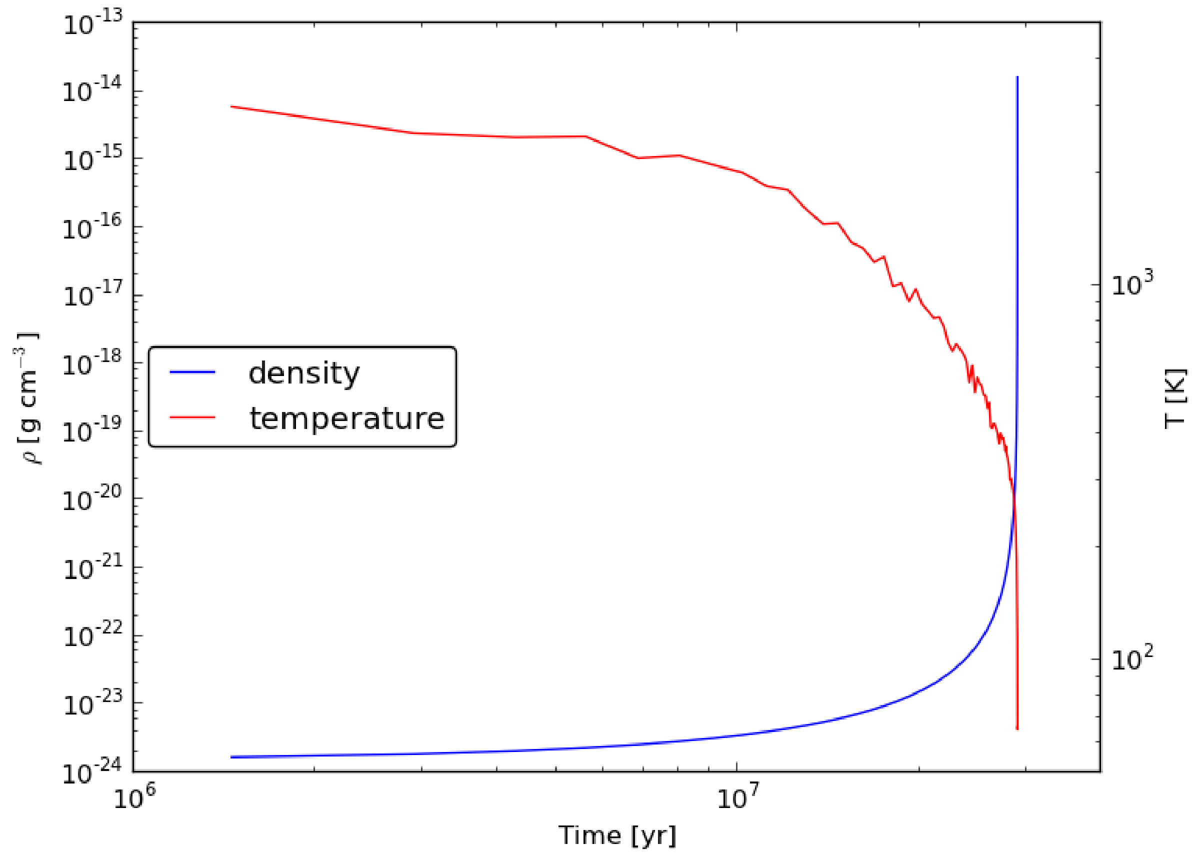
```
python cooling_rate.py
```



4.2.2 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
```



4.2.3 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
```

CHAPTER 5

Help

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

CHAPTER 6

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. 2013; Kim, J. et al. 2013)." Also, please add a footnote to https://grackle.readthedocs.org/.

The Enzo Collaboration, Bryan, G. L., Norman, M. L., et al. 2013, arXiv:1307.2265

Kim, J.-h., Abel, T., Agertz, O., et al. 2013, arXiv:1308.2669

CHAPTER 7

Search

- *search*