DESPOTIC Documentation

Release 2.0

Mark R. Krumholz

December 20, 2015

Contents:

CHAPTER

ONE

LICENSE AND CITATIONS

DESPOTIC is distributed under the terms of the GNU General Public License version 3.0. The text of the license is included in the main directory of the repository as `GPL-3.0.txt`.

If you cite DESPOTIC in any published work, please cite the code paper: DESPOTIC – A New Software Library to Derive the Energetics and SPectra of Optically Thick Interstellar Clouds, Krumholz, M. R. 2014, Monthly Notices of the Royal Astronomical Society, 437, 1662.

CHAPTER

TWO

INSTALLING DESPOTIC

DESPOTIC is distributed in two ways.

2.1 Installing from git

The full source distribution, including example Python programs using it and sample cloud descriptor files, is available from bitbucket, and can be obtained via git by doing:

```
git clone git@bitbucket.org:krumholz/despotic.git
```

The package can then be installed by doing:

```
python setup.py install
```

in the `despotic` directory. You may need to preface this with `sudo` if you want to install in a way that is accessible for all users on the system.

2.2 Installing from pip

DESPOTIC is also available through the python package index. If you have pip installed, you can just type:

```
pip install despotic
```

As with the setup via `git`, you may need to preface this with `sudo` to install globally.

2.3 Setting Up the Environment

DESPOTIC creates a local cache of atomic and molecular data files. You can specify the location of this cache by setting the `$DESPOTIC_HOME` environment variable; the data cache will be created in `$DESPOTIC_HOME/LAMDA` If you are using a `bash`-like shell, the syntax to set the location of `$DESPOTIC_HOME` is:

```
export DESPOTIC_HOME = /path/to/despotic
```

while for a `csh`-like shell, it is:

```
setenv DESPOTIC_HOME /path/to/despotic
```

If `$DESPOTIC_HOME` is not set, then DESPOTIC will attempt to create the cache the directory from which is is run.

2.4 Requirements and Dependencies

DESPOTIC requires

- *scipy >= 0.11.0*
- *cython >= 0.20.x*
- *matplotlib >= 1.3.x*

CHAPTER

THREE

QUICKSTART

3.1 Introduction

DESPOTIC is a tool to Derive the Energetics and SPectra of Optically Thick Interstellar Clouds. It can perform a variety of calculations regarding the chemical and thermal state of interstellar clouds, and predict their observable line emission. DESPOTIC treats clouds in a simple one-zone model, and is intended to allow rapid, interactive exploration of parameter space.

In this Quickstart, we will walk through a basic interactive python session using DESPOTIC. This will work equally well from an ipython shell or in an ipython notebook.

3.2 The `cloud` Class

The basic object in DESPOTIC, which provides an interface to most of its functionality, is the class `cloud`. This class stores the basic properties of an interstellar cloud, and provides methods to perform calculations on those properties. The first step in most DESPOTIC sessions is to import this class:

```
from despotic import cloud
```

The next step is generally to input data describing a cloud upon which computations are to be performed. The input data describe the cloud's physical properties (density, temperature, etc.), the bulk composition of the cloud, what emitting species it contains, and the radiation field around it. While it is possible to enter the data manually, it is usually easier to read the data from a file, using the *Cloud Files* format. For this Quickstart, we'll use one of the configuration files that ship with DESPOTIC and that are included in the `cloudfiles` subdirectory of the DESPOTIC distribution. To create a cloud whose properties are as given in a particular cloud file, we simply invoke the constructor with the `fileName` optional argument set equal to a string containing the name of the file to be read:

```
gmc = cloud(fileName="cloudfiles/MilkyWayGMC.desp", verbose=True)
```

Note that, if you're not running DESPOTIC from the directory where you installed it, you'll need to include the full path to the `cloudfiles` subdirectory in this command. Also note the optional argument `verbose`, which we have set to `True`. Most DESPOTIC methods accept the `verbose` argument, which causes them to produce printed output containing a variety of information. By default DESPOTIC operations are silent.

3.3 Computing Temperatures

At this point most of the calculations one could want to do on a cloud are provided as methods of the `cloud` class. One of the most basic is to set the cloud to its equilibrium dust and gas temperatures. This is accomplished via the `setTempEq` method:

```
gmc.setTempEq(verbose=True)
```

With `verbose` set to `True`, this command will produce variety of output as it iterates to calculate the equilibrium gas and dust temperatures, before finally printing `True`. This illustrates another feature of DESPOTIC commands: those that iterate return a value of `True` if they converge, and `False` if they do not.

To see the gas and dust temperatures to which the cloud has been set, we can simply print them:

```
print gmc.Tg
print gmc.Td
```

This shows that DESPOTIC has calculated an equilibrium gas temperature of 10.2 K, and an equilibrium dust temperature of 14.4 K.

3.4 Line Emission

Next we might wish to compute the CO line emission emerging from the cloud. We do this with the `cloud` method `lineLum`:

```
lines = gmc.lineLum("co")
```

The argument `co` specifies that we are interested in the emission from the CO molecule. This method returns a `list` of `dict`, each of which gives information about one of the CO lines. The `dict` contains a variety of fields, but one of them is the velocity-integrated brightness temperature of the line. Again, we can just print the values we want. The first element in the list is the $J = 1 \rightarrow 0$ line, and the velocity-integrated brightness temperature is listed as `intTB` in the `dict`. Thus to get the velocity-integrated brightness temperature of the first line, we just do:

```
print lines[0]['intTB']
```

This shows that the velocity-integrated brightness temperature of the CO $J = 1 \rightarrow 0$ line is 79 K km/s.

3.5 Heating and Cooling Rates

Finally, we might wish to know the heating and cooling rates produced by various processes, which lets us determined what sets the thermal balance in the cloud. This may be computed using the method `dEdt`, as follows:

```
rates = gmc.dEdt()
```

This method returns a `dict` that contains all the heating and cooling terms for gas and dust. For example, we can print the rates of cosmic ray heating and CO line cooling via:

```
print rates["GammaCR"]
print rates["LambdaLine"]["co"]
```

CHAPTER

FOUR

FUNCTIONAL GUIDE TO DESPOTIC CAPABILITIES

This section covers the most common tasks for which DESPOTIC can be used. They are not intended to provide a comprehensive overview of the library's capabilities, and users who wish to understand every available option should refer to *Full Documentation of All DESPOTIC Classes and Functions*. The routines used in this section are all described in full detail there. For all of these operations, the user should first have imported the basic DESPOTIC class `cloud` by doing:

```
from despotic import cloud
```

In the examples below we will also assume that `matplotlib` and `numpy` have both been imported, via:

```
import matplotlib.pyplot as plt
import numpy as np
```

4.1 Unit Conventions

In this section, and in general when using DESPOTIC, there are two important conventions to keep in mind.

1. All quantities are in CGS units unless otherwise specified. The main exceptions are quantites that are normalized to Solar or Solar neighborhood values (e.g. metallicity and interstellar radiation field strength) and quantities where the conventional unit is non-CGS (e.g. integrated brightness temperatures are expressed in the usual units of K km/s).
2. All rates are expressed per H nucleus, rather than per unit mass or per unit volume. This includes chemical abundances. Thus for example a heating rate of $\Gamma = 10^{-26}$ should be understood as 10^{-26} erg/s/H nucleus. An abundance $x_{\mathrm{He}} = 0.1$ should be understood as 1 He atom per 10 H nuclei.

4.2 Line Emission

The most basic task in DESPOTIC is computing the line emission emerging from a cloud of specified physical properties. The first step in any computation of line emission is to create a cloud object with the desired properties. This is often most easily done by creating a DESPOTIC cloud file (see *Cloud Files*), but the user can also create a cloud with the desired properties manually. The properties that are important for line emission are the gas volume density, column density, temperature, velocity dispersion, and chemical composition; these are stored in the cloud class and the composition class within it. For example, the following code snippet:

```
mycloud = cloud()
mycloud.nH = 1.0e2
mycloud.colDen = 1.0e22
mycloud.sigmaNT = 2.0e5
mycloud.Tg = 10.0
mycloud.comp.xoH2 = 0.1
mycloud.comp.xpH2 = 0.4
mycloud.comp.xHe = 0.1
```

creates a cloud with all its parameters set to default values, a volume density of H nuclei $n_{\rm H} = 10^2\,\rm cm^{-3}$, a column density of H nuclei $N_{\rm H} = 10^{22}\,\rm cm^{-2}$, a non-thermal velocity dispersion of $\sigma_{\rm NT} = 2.0\times10^5\,\rm cm\,s^{-1}$, a gas temperature of $T_g = 10\,\rm K$, and a composition that is 0.1 ortho-$\rm H_2$ molecules per H nucleus, 0.4 para-$\rm H_2$ molecules, and 0.1 He atoms per H nucleus.

The next step is to specify the emitting species whose line emission is to be computed. As with the physical properties of the cloud, this is often most easily done by creating a cloud file. However, it can also be done manually by using the cloud.addEmitter method, which allows users to add emitting species to clouds. The following code snippet adds CO as an emitting species, at an abundance of 10^{-4} CO molecules per H nucleus:

```
mycloud.addEmitter("CO", 1.0e-4)
```

The first argument is the name of the emitting species, and the second is the abundance. The requisite molecular data file will be read from disk if it is available, or automatically downloaded from the Leiden Atomic and Molecular Database if it is not (see *Atomic and Molecular Data*).

Once an emitter has been added, only a single call is required to calculate the luminosity of its lines:

```
lines = mycloud.lineLum("CO")
```

The argument is just the name of the species whose line emission should be computed. Note that it must match the name of an existing emitter, and that emitter names are case-sensitive. The value returned by this procedure, which is stored in the variable `lines`, is a `list` of `dict`, with each `dict` describing the properties of a single line. Lines are ordered by frequency, from lowest to highest. Each `dict` contains the following keys-value pairs

- `freq` is the line frequency in Hz
- `intIntensity` is the frequency-integrated intensity of the line after subtracting off the CMB contribution, in $\rm erg\,cm^{-2}\,s^{-1}\,sr^{-1}$
- `intTB` is the velocity-integrated brightness temperature (again subtracting off the CMB) in $\rm K\,km\,s^{-1}$
- `lumPerH` is the rate of energy emission in the line per H nucleus in the cloud, in $\rm erg\,s^{-1}$.

This is a partial list of what the `dict` contains; see *cloud* for a complete listing.

Once the data been obtained, the user can do what he or she wishes with them. For example, to plot velocity-integrated brightness temperature versus line frequency, the user might do:

```
freq = [l["freq"] for l in lines]
TB = [l["intTB"] for l in lines]
plt.plot(freq, TB, "o")
```

4.3 Heating and Cooling Rates

To use DESPOTIC's capability to calculate heating and cooling rates, in addition to the quantities specified for a calculation of line emission one must also add the quantities describing the dust and the radiation field. As before, this is most easily accomplished by creating a DESPOTIC cloud file (see *Cloud Files*), but the data can also be input manually. The code snippet below does so:

```
mycloud.dust.alphaGD   = 3.2e-34    # Dust-gas coupling coefficient
mycloud.dust.sigma10   = 2.0e-25    # Cross section for 10K thermal radiation
mycloud.dust.sigmaPE   = 1.0e-21    # Cross section for photoelectric heating
mycloud.dust.sigmaISRF = 3.0e-22    # Cross section to the ISRF
mycloud.dust.beta      = 2.0        # Dust spectral index
mycloud.dust.Zd        = 1.0        # Abundance relative to Milky Way
mycloud.Td             = 10.0       # Dust temperature
mycloud.rad.TCMB       = 2.73       # CMB temperature
mycloud.rad.TradDust   = 0.0        # IR radiation field seen by the dust
mycloud.rad.ionRate    = 2.0e-17    # Primary ionization rate
mycloud.rad.chi        = 1.0        # ISRF normalized to Solar neighborhood
```

These quantities specify the dust-gas coupling constant, the dust cross section to 10 K thermal radiation, the dust cross section to the 8 - 13.6 eV photons the dominate photoelectric heating, the dust cross section to the broader interstellar radiation field responsible for heating the dust, the dust spectral index, the dust abundance relative to the Milky Way value, the dust temperature, the cosmic microwave background temperature, the infrared radiation field that heats the dust, the primary ionization rate due to cosmic rays and x-rays, and the ISRF strength normalized to the Solar neighborhood value. All of the numerical values shown in the code snippet above are in fact the defaults, and so none of the above commands are strictly necessary. However, it is generally wise to set quantities explicitly rather than relying on default values.

Once these data have been input, one may compute all the heating and cooling terms that DESPOTIC includes using the `cloud.dEdt` routine:

```
rates = mycloud.dEdt()
```

This call returns a dict which contains the instantaneous rates of heating and cooling. The entries in the dict are: `GammaPE`, the gas photoelectric heating rate, `GammaCR`, the gas heating rate due to cosmic ray and X-ray ionization, `GammaGrav`, the gas heating rate due to gravitational compression, `GammaDustISRF`, the dust heating rate due to the ISRF, `GammaDustCMB`, the dust heating rate due to the CMB, `GammaDustIR`, the dust heating rate due to the IR field, `GammaDustLine`, the dust heating rate due to absorption of line photons, `PsiGD`, the gas-dust energy exchange rate (positive means gas heating, dust cooling), `LambdaDust`, the dust cooling rate via thermal emission, and `LambdaLine`, the gas cooling rate via line emission. This last quantity is itself a dict, with one entry per emitting species and the dictionary keys corresponding to the emitter names. Thus in the above example, one could see the cooling rate via CO emission by doing:

```
print rates["LambdaLine"]["CO"]
```

4.4 Temperature Equilibria

Computing the equilibrium temperature requires exactly the same quantities as computing the heating and cooling rates; indeed, the process of computing the equilibrium temperature simply amounts to searching for values of T_g and T_d such that the sum of the heating and cooling rates returned by `cloud.dEdt` are zero. One may perform this calculation using the `cloud.setTempEq` method:

```
mycloud.setTempEq()
```

This routine iterates to find the equilibrium gas and dust temperatures, and returns True if the iteration converges. After this call, the computed dust and gas temperatures may simply be read off:

```
print mycloud.Td, mycloud.Tg
```

The `cloud.setTempEq` routine determines the dust and gas temperatures simultaneously. However, there are many situations where it is preferable to solve for only one of these two, while leaving the other fixed. This may be accomplished by the calls:

```
mycloud.setDustTempEq()
mycloud.setGasTempEq()
```

These routines, respectively, set `mycloud.Td` while leaving `mycloud.Tg` fixed, or vice-versa. Solving for one temperature at a time is often faster, and if dust-gas coupling is known to be negligible will produce nearly identical results as solving for the two together.

4.5 Time-Dependent Temperature Evolution

To perform computations of time-dependent temperature evolution, DESPOTIC provides the method `cloud.tempEvol`. In its most basic form, this routine simply accepts an argument specifying the amount of time for which the cloud is to be integrated, and returning the temperature as a function of time during this evolution (note that executing this command may take a few minutes, depending on your processor):

```
mycloud.Tg = 50.0          # Start the cloud out of equilibrium
tFinal = 20 * 3.16e10      # 20 kyr
Tg, t = mycloud.tempEvol(tFinal)
```

The two values returned are arrays, the second of which gives a series of 100 equally-spaced times between 0 and `tFinal`, and the first of which gives the temperatures at those times. The number of output times, the spacing between them, and their exact values may all be controlled by optional arguments – see *cloud* for details. At the end of this evolution, the cloud temperature `mycloud.Tg` will be changed to its value at the end of 20 kyr of evolution, and the dust temperature `mycloud.Tg` will be set to its thermal equilibrium value at that cloud temperature.

If one wishes to examine the intermediate states in more detail, one may also request that the full state of the cloud be saved at every time:

```
clouds, t = mycloud.tempEvol(tFinal, fullOutput=True)
```

The `fullOutput` optional argument, if `True`, causes the routine to return a full copy of the state of the cloud at each output time, instead of just the gas temperature `Tg`. In this case, `clouds` is a sequence of 100 `cloud` objects, and one may interrogate their states (e.g. calculating their line emission) using the usual routines.

4.6 Chemical Equilibria

DESPOTIC can also compute the chemical state of clouds from a chemical network. Full details on chemical networks are given in *Chemistry and Chemical Networks*, but for this example we will use a simple network that DESPOTIC ships with, that of Nelson & Langer (1999, ApJ, 524, 923). This network computes the chemistry of carbon and oxygen in a region where the hydrogen is fully molecular. For more details see *NL99*.

To perform computations with this network, one must first import the class that defines it:

```
from despotic.chemistry import NL99
```

One can set the equilibrium abundances of a cloud to the equilibrium values determined by the network via the command:

```
mycloud.setChemEq(network=NL99)
```

The argument `network` specifies that the calculation should use the `NL99` class. This call sets the abundances of all the emitters that are included in the network to their equilibrium values. In this case, the network includes CO, and thus it sets the CO abundance to a new value:

```
print mycloud.emitters["CO"].abundance
```

One can also see the abundances of all the species included in the network, including those that do not correspond to emitters in the cloud, by printing the chemical network property `abundances`:

```
print mycloud.chemnetwork.abundances
```

Once the chemical network is associated with the cloud, subsequent calls to `setChemEq` need not include the `network` keyword. DESPOTIC assumes that all subsequent chemical calculations are to be performed with the same chemical network unless it is explicitly told otherwise via a call to `setChemEq` or `chemEvol` (see *Time-Dependent Chemical Evolution*) that specifies a different chemical network.

4.7 Simultaneous Chemical and Thermal Equilibria

The `setChemEq` routine (see *Chemical Equilibria*) called with no extra arguments leaves the gas temperature fixed. However, it is also possible to compute a simultaneous equilibrium for the temperature and the thermal state. To do so, we first import a chemical network to be used, in this case the Nelson & Langer (1999) network (see *Chemical Equilibria*):

```
from despotic.chemistry import NL99
```

We then call `cloud.setChemEq` with an optional keyword `evolveTemp`

```
mycloud.setChemEq(network=NL99, evolveTemp=``iterate``)
```

The `network` keyword specifies that the computation should use the NL99 network, while `evolveTemp` specifies how to handle the simultaneous thermal and chemical equilibrium calculation. The options available are

- `fixed`: gas temperature is held fixed
- `iterate`: calculation iterates between computing chemical and gas thermal equilibria, i.e., chemical equilibrium is computed at fixed temperature, equilibrium gas temperature (see *Temperature Equilibria*) is computed for fixed abundances, and the process is repeated until the temperature and abundances converge; dust temperature is held fixed
- `iterateDust`: same as `iterate`, except the dust temperature is iterated as well
- `gasEq`: gas temperature is always set to its instantaneous equilibrium value as the chemical state is evolved toward equilibrium; dust temperature is held fixed
- `fullEq`: same as `gasEq`, except that both gas and dust temperatures are set to their instantaneous equilibrium values
- `evol`: chemical state and gas temperature are evolved in time together, while dust temperature is always set to its instantaneous equilibrium value; evolution stops once gas temperature and abundances stop changing significantly

Note that, while in general the different evolution methods will converge to the same answer, there is no guarantee that they will do in systems where multiple equilibria exist.

4.8 Time-Dependent Chemical Evolution

DESPOTIC can also calculate time-dependent chemical evolution. This is accomplished through the method cloud.chemEvol. At with `cloud.tempEvol` (see *Time-Dependent Temperature Evolution*), this routine accepts an argument specifying the amount of time for which the cloud is to be integrated, and returning the chemical abundances as a function of time during this evolution:

```
mycloud.rad.ionRate = 2.0e-16 # Raise the ionization rate a lot
tFinal = 0.5 * 3.16e13 # 0.5 Myr
abd, t = mycloud.chemEvol(tFinal, network=NL99)
```

Note that the `network=NL99` option may be omitted if one has previously assigned that network to the cloud (for example by executing the examples in *Chemical Equilibria*).

The output quantity abd here is an object of class `abundanceDict`, which is a specialized dict for handling chemical abundances – see *abundanceDict*. One can examine the abundances of specific species just by giving their chemical names. For example, to see the time-dependent evolution of the abundances of CO, C, and C^+, one could do:

```
plt.plot(t, abd["CO"])
plt.plot(t, abd["C"])
plt.plot(t, abd["C+"])
```

As with `setChemEq`, this routine modifies the abundances of emitters in the cloud to the values they achieve at the end of the evolution, so to see the final CO abundance one could do:

```
print mycloud.emitters["CO"].abundance
```

4.9 Multi-Zone Clouds

While most DESPOTIC functionality is provided through the `cloud` class, which represents a single cloud, it is sometimes useful to have a cloud that contains zones of different optical depths. This functionality is provided through the `zonedcloud` class. A `zonedcloud` is just a collection of `cloud` objects that are characterized by having different column densities (and optionally volume densities), and on which all the operations listed above can be performed in a batch fashion.

One can create a `zonedcloud` in much the same way as a `cloud`, but reading from an input file:

```
from despotic import zonedcloud
zc = zonedcloud(fileName="cloudfiles/MilkyWayGMC.desp")
```

A `zonedcloud` is characterized by column densities for each of its zones, which can be accessed through the `colDen` property:

```
print zc.colDen
```

The column densities of all zones, and the number of zones, can be controlled when the `zonedcloud` is created using the keywords `nZone` and `colDen`; see *zonedcloud* for the full list of keywords.

Once a `zonedcloud` exists, all of the functions described above in this section are available for it, and will be applied zone by zone. For example, one can do:

```
zc.setTempEq()
print zc.Tg
```

to set and then print the temperature in each zone. Commands the report observable quantities or abundances will return appropriately-weighted sums over the entire cloud. For example:

```
zc.lineLum('co')[0]
```

returns a dict describing the $J = 1 \rightarrow 0$ line of CO. The quantities `intTB` and `intIntensity` that are part of the dict and contain the velocity-integrated brightness temperature and frequency-integrated intensity, respectively (see *Line Emission*), are sums over all zones, while ones like `Tex` (the excitation temperature) that do not make sense to sum are returned as an array giving zone-by-zone values.

4.10 Computing Line Profiles

Line profile computation operates somewhat differently then the previous examples, because it is provided through a stand-alone procedure rather than through the cloud class. This procedure is called lineProfLTE, and may be imported directly from the DESPOTIC package. The routine also requires emitter data stored in an `emitterData` object. The first step in a line profile calculation is therefore to import these two objects into the python environment:

```
from despotic import lineProfLTE
from despotic import emitterData
```

The second step is to read in the emitter data. The interface to read emitter data is essentially identical to the one used to add an emitter to a cloud. One simply declares an `emitterData` object, giving the name of the emitter as an argument:

```
csData = emitterData('CS') # Reads emitter data for the CS molecule
```

Alternately, emitter data may be obtained from a `cloud`, since clouds store emitter data for all their emitters. Using the examples from the previous sections:

```
coData = mycloud.emitters["CO"].data
```

copies the emitter data for CO to the variable `coData`.

The third step is to specify the radius of the cloud, and the profiles of any quantities within the cloud that are to change with radius, including density, temperature, radial velocity, and non-thermal velocity dispersion. Each of these can be constant, but the most interesting applications are when one or more of them are not, in which case they must be defined by functions. These function each take a single argument, the radius in units where the outer radius of the cloud is unity, and return a single floating point value, giving the quantity in question in CGS units. For example, to compute line profiles through a cloud of spatially-varying temperature and infall velocity, one might define the functions:

```
R = 0.02 * 3.09e18 # 0.2 pc
def TProf(r):
    return 8.0 + 12.0*np.exp(-r**2/(2.0*0.5**2))
def vProf(r):
    return -4.0e4*r
```

The first function sets a temperature that varies from 20 K in the center of close to 8 K at the outer edge, and the second defines a velocity that varies from 0 in the center to -0.4 km s^{-1} (where negative indicates infall) at the outer edge. Similar functions can be defined by density and non-thermal velocity dispersion if the user so desires. Alternately, the user can simply define them as constants:

```
ncs = 0.1       # CS density 0.1 cm^-3
sigmaNT = 2.0e4 # Non-thermal velocity dispersion 0.2 km s^-1
```

The final step is to use the `lineProfLTE` routine to compute the brightness temperature versus velocity:

```
TB, v = lineProfLTE(cs, 2, 1, R, ncs, TProf, vProf, sigmaNT).
```

Here the first argument is the emitter data, the second and third are the upper and lower quantum states between which the line is to be computed (ordered by energy, with ground state = 0), followed by the cloud radius, the volume density, the temperature, the velocity, and the non-thermal velocity dispersion. Each of these quantities can be either a float or a callable function of one variable, as in the example above. If it is a float, that quantity is taken to be constant, independent of radius. This routine returns two arrays, the first of which is the brightness temperature and the second of which is the velocity at which that brightness temperature is computed, relative to line center. These can be examined in any of the usual numpy ways, for example by plotting them:

```
plt.plot(v, TB)
```

By default the velocity is sampled at 100 values. The routine attempts to guess a reasonable range of velocities based on the input values of radial velocity and velocity dispersion, but these defaults may be overridden by the optional argument `vLim`, which is a sequence of two values giving the lower and upper limits on the velocity:

```
TB, v = lineProfLTE(cs, 2, 1, R, ncs, TProf, vProf, sigmaNT,
                    vLim=[-2e5,2e5]).
```

A variety of other optional arguments can be used to control the velocities at which the brightness temperature is computed. It is also possible to compute line profiles at positions offset from the geometric center of the cloud, using the optional argument offset – see *lineProfLTE*.

4.11 Escape Probability Geometries

DESPOTIC supports three possible geometries that can be used when computing escape probabilities, and which are controlled by the `escapeProbGeom` optional argument. This argument is accepted by all DESPOTIC functions that use the escape probability formalism, including all those involving computation of line emission. This optional argument, if included, must be set equal to one of the three strings `sphere` (the default), `slab`, or `LVG`. These choices correspond to spherical geometry, slab geometry, and the large velocity gradient approximation, respectively.

CHAPTER FIVE

CLOUD FILES

5.1 File Structure

DESPOTIC cloud files contain descriptions of clouds that can be read by the `cloud` or `zonedcloud` classes, using either the class constructor or the read method; see *cloud* for details. This section contains a description of the format for these files. It is recommended but not required that cloud files have names that end in the extension `.desp`.

Each line of a cloud file must be blank, contain a comment starting with the character `#`, or contain a key-value pair formatted as:

```
key = value
```

The line may also contain comments after `value`, again beginning with `#`. Any content after `#` is treated as a comment and is ignored. DESPOTIC keys are case-insensitive, and whitespace around keys and values are ignored. For the full list of keys, see *Cloud file keys and their meanings*. All quantities must be in CGS units. Key-value pairs may be placed in any order, with the exception of the key `H2opr`, which provides a means of specify the ratio of ortho-to-para-H_2, instead of directly setting the ortho-and para-H_2 abundances. If this key is specified, it must precede the key `xH2`, which gives the total H_2 abundance including both ortho- and para- species. Not all keys are required to be present. If left unspecified, most quantities default to a fiducial Milky Way value (if a reasonable one exists, e.g., for the gas-dust coupling constant and ISRF strength) or to 0 (if it does not, e.g., for densities and chemical abundances).

Table 5.1: Cloud file keys and their meanings

Key	Units	Description
Physical Properties		
nH	cm^{-3}	Volume density of H nuclei
colDen	cm^{-2}	Column density of H nuclei, averaged over area
sigmaNT	$\mathrm{cm\ s}^{-1}$	Non-thermal velocity dispersion
Tg	K	Gas temperature
Td	K	Dust temperature
Dust Properties		
alphaGD	$\mathrm{erg\ cm}^3\ \mathrm{K}^{-3/2}$	Gas-dust collisional coupling coefficient
sigmaD10	$\mathrm{cm}^2\ \mathrm{H}^{-1}$	Dust cross section per H to 10 K thermal radiation
sigmaDPE	$\mathrm{cm}^2\ \mathrm{H}^{-1}$	Dust cross section per H to 8-13.6 eV photons
sigmaDISRF	$\mathrm{cm}^2\ \mathrm{H}^{-1}$	Dust cross section per H averaged over ISRF
betaDust	Dimensionless	Dust IR spectral index
Zdust	Dimensionless	Dust abundance normalized to Milky Way value
Radiation Field Properties		
TCMB	K	CMB temperature
TradDust	K	Temperature of the dust-trapped IR radiation field
ionRate	s^{-1}	Primary cosmic ray / x-ray ionization rate
chi	Dimensionless	ISRF strength, normalized to Solar neighborhood
Chemical composition		
emitter		See *Emitters*

5.2 Emitters

The `emitter` key is more complex than most, and requires special mention. Lines describing emitters follow the format:

```
emitter = name abundance [noextrap] [energySkip] [file:FILE] [url:URL]
```

Here the brackets indicate optional items, and the optional items may appear in any order, but must be after the two mandatory ones.

The first mandatory item, `name`, gives name of the emitting molecule or atom. Note that molecule and atom names are case sensitive, in the sense that DESPOTIC will not assume that `co` and `CO` describe the same species. Any string is acceptable for `name`, but if the file or URL containing the data for that species is not explicitly specified, the name is used to guess the corresponding file name in the Leiden Atomic and Molecular Database (LAMDA) – see *Atomic and Molecular Data*. It is therefore generally advisable to name a species following LAMDA convention, which is that molecules are specified by their chemical formula, with a number specifying the atomic weight preceding the if the species is not the most common isotope. Thus LAMDA refers to $^{28}\mathrm{Si}^{16}\mathrm{O}$ (the molecule composed of the most common isotopes) as `sio`, $^{29}\mathrm{Si}^{16}\mathrm{O}$ as `29sio`, and $^{12}\mathrm{C}^{18}\mathrm{O}$ as `c18o`. The automatic search for files in LAMDA also includes common variants of the file name used in LAMDA. The actual file name from which DESPOTIC reads data for a given emitter is stored in the emitterData class – see *emitterData*.

The second mandatory item, `abundance`, gives the abundance of that species relative to H nuclei. For example, an abundance of 0.1 would indicate that there is 1 of that species per 10 H nuclei.

The optional items `noextrap` and `energySkip` change how DESPOTIC performs computations with that species. If `noextrap` is set, DESPOTIC will raise an error if any attempt is made to calculate a collision rate coefficient between that species and one of the bulk components (H, He, etc.) that is outside the range tabulated in the data file. If not, DESPOTIC will instead handle temperatures outside the tabulated range by substituting the closest temperature inside the tabulated range. Note that this behavior can be altered within a DESPOTIC program by using the `extrap` property of the `emitterData` class – see *Full Documentation of All DESPOTIC Classes and Functions*.

The optional item `energySkip` specifies that a species should be ignored when computing heating and cooling rates via the `cloud.dEdt` method, and by extension whenever thermal equilibria or thermal evolution are computed for that cloud. However, line emission from that species can still be computed using the `cloud.lineLum` method. This option is therefore useful for including species for which the line emission is an interesting observable, but which are irrelevant to the thermal balance and thus can be omitted when calculating cloud thermal properties in order to save computational time.

Finally, the optional items `file:FILE` and `url:URL` specify locations of atomic and molecular data files, either on the local file system or on the web. This capability is useful in part because some LAMDA files do not follow the usual naming convention, or because for some species LAMDA provides more than one version of the data for that species (e.g., two versions of the data file for atomic C exist, one with only the low-lying IR levels, and another including the higher-energy UV levels). File specifications must be of the form `file:FILE` with `FILE` replaced by a file name, which can include both absolute and relative paths. If no path or a relative path is given, DESPOTIC searches for the file first in the current directory, and then in the directory `$DESPOTIC_HOME/LAMDA`, where `$DESPOTIC_HOME` is an environment variable. If it is not specified, DESPOTIC just looks for a directory called LAMDA relative to the current directory.

The `url:URL` option can be used to specify the location of a file on the web, usually somewhere on the LAMDA website. It must be specified as `url:URL`, where `URL` is replaced by an absolute or relative URL. If an absolute URL is given, DESPOTIC attempts to download the file from that location. If a relative URL is given, DESPOTIC attempts to download the file from at `http://$DESPOTIC_LAMDAURL/datafiles/URL`, where `$DESPOTIC_LAMDAURL` is an environment variable. If this environment variable is not specified, DESPOTIC searches for the file at `http://home.strw.leidenuniv.nl/~moldata/URL`.

CHAPTER

SIX

ATOMIC AND MOLECULAR DATA

DESPOTIC requires atomic and molecular data to work. This section describes how it handles these data, both on disk and in its internal workings.

6.1 The Local Database

DESPOTIC uses atomic and molecular data in the format specified by the Leiden Atomic and Molecular Database. The user can manually supply the required data files, but the more common use case is to access the data directly from LAMDA. When emitter data is required, DESPOTIC will attempt to guess the name of the required data file and download it automatically – see *Emitters*. When DESPOTIC downloads a file from LAMDA, it caches a local copy for future use. The next time the same emitter is used, unless DESPOTIC is given an explicit URL from which the file should be fetched, it will use the local copy instead of re-downloading the file from LAMDA. (However, see *Keeping the Atomic and Molecular Data Up to Date*.)

The location of the database is up to the user, and is specified through the environment variable `$DESPOTIC_HOME`. If this environment variable is set, LAMDA files will be places in the directory `$DESPOTIC_HOME/LAMDA`, and that is the default location that will be searched when a file is needed. If the environment variable `$DESPOTIC_HOME` is not set, DESPOTIC looks for files in a subdirectory LAMDA of the current working directory, and caches files in that directory if they are downloaded. It is recommended that users set a `$DESPOTIC_HOME` environment variable when working with DESPOTIC, so as to avoid downloading and caching multiple copies of LAMDA for different projects in different directories.

6.2 Keeping the Atomic and Molecular Data Up to Date

The data in LAMDA are updated regularly as new calculations or laboratory experiments are published. Some of these updates add new species, but some also provide improved data on species that are already in the database. DESPOTIC attempts to ensure that its locally cached data are up to date by putting an expiration date on them. By default, if DESPOTIC discovers that a given data file is more than six months old, it will re-download that file from LAMDA. This behavior can be overridden by manually specifying a file name, either in the cloud file (see *Cloud Files*) or when invoking the `cloud.addEmitter` or `emitter.__init__` methods. Users can also force updates of the local database more frequently using the `refreshLamda` function – see *refreshLamda*.

6.3 DESPOTIC's Internal Model for Atomic and Molecular Data

When it is running, DESPOTIC maintains a list of emitting species for which data have been read within the `emitter` module (see *emitter*). Whenever a new emitter is created, either for an existing cloud, for a new cloud being created, or as a free-standing object of the emitter class, DESPOTIC checks the emitter name against the central list. If the name is found in the list, DESPOTIC will simply use the stored data for that object rather than re-reading the file containing the data. This is done as an efficiency measure, and also to ensure consistency between emitters of the same species associated with different clouds. However, this model has some important consequences of which the user should be aware.

1. Since data on level structure, collision rates, etc. (everything stored in the `emitterData` class – see *emitterData*) is shared between all emitters of the same name, and any alterations made to the data for one emitter will affect all others of the same name.
2. It is not possible to have two emitters of the same name but with different data. Should a user desire to achieve this for some reason (e.g., to compare results computed using an older LAMDA file and a newer one), the way to achieve this is to give the two emitters different names, such as `co_ver1` and `co_ver2`.
3. Maintenance of a central emitter list affects how deepcopy and pickling operations operate on emitters. See *emitterData* for details.

CHAPTER

SEVEN

CHEMISTRY AND CHEMICAL NETWORKS

The chemistry capabilities in DESPOTIC are located in the `despotic.chemistry` module.

7.1 Operations on Chemical Networks

The central object in a chemical calculation is a chemical network, which consists of a set of chemical species and a set of reactions that change the concentrations of each of them. Formally, a network is defined by a set of N species with abundances $\mathbf{x}$ (defined as number density of each species per H nucleus), and a function $d\mathbf{x}/dt = \mathbf{f}(\mathbf{x}, \mathbf{p})$ that gives the time rate of change of the abundances. The vector $\mathbf{p}$ specifies a set of parameters on which the reaction rates depend, and it may include quantities such as the ambient radiation field (for photoreactions), the density, the temperature, etc. The numerical implementation of chemical networks is described in *Defining New Chemical Networks: the chemNetwork Class*. Given any chemical network, DESPOTIC is capable of two operations:

- Given an initial set of abundances at $\mathbf{x}(t_0)$ at time t_0, compute the abundances at some later time t_1. This is simply a matter of numerically integrating the ordinary differential equation $d\mathbf{x}/dt = \mathbf{f}(\mathbf{x}, \mathbf{p})$ from t_0 to t_1, where $\mathbf{f}$ is a known function. In DESPOTIC, this capability is implemented by the routine `chemEvol` in the `despotic.chemistry.chemEvol` module. The `cloud` class contains a wrapper around this routine, which allows it to be called to operate on a specific instance of `cloud` or `zonedcloud`.
- Find an equilibrium set of abundances $\mathbf{x}_{\mathrm{eq}}$ such that $d\mathbf{x}/dt = \mathbf{f}(\mathbf{x}_{\mathrm{eq}}, \mathbf{p}) = 0$. Note that these is in general no guarantee that such an equilibrium exists, or that it is unique, and there are no general techniques for identifying such equilibria for arbitrary vector functions $\mathbf{f}$. DESPOTIC handles this problem by explicitly integrating the ODE $d\mathbf{x}/dt = \mathbf{f}(\mathbf{x}, \mathbf{p})$ until $\mathbf{x}$ reaches constant values (within some tolerance) or until a specified maximum time is reached. In DESPOTIC, this capability is implemented by the routine `setChemEq` in the `despotic.chemistry.setChemEq` module. The `cloud` and `zonedcloud` classed contains a wrapper around this routine, which allows it to be called to operate on a specific instance of `cloud` or `zonedcloud`.

7.2 Predefined Chemical Networks

DESPOTIC ships with two predefined chemical networks, described below.

7.2.1 `NL99`

The `NL99` network implements the reduced C-O network introduced by Nelson & Langer (1999, Astrophysical Journal, 524, 923; hereafter NL99). Readers are refereed to that paper for a full description of the network and the physical approximations on which it relies. To summarize briefly here, the network is intended to capture the chemistry of carbon and oxygen as it occurs at moderate densities and low temperatures in H_2 dominated clouds. It includes the species C, C^+, CHx, CO, HCO^+, H_3^+, He^+, O, OHx, M, and M^+. Several of these are "super-species" that agglomerate several distinct species with similar reaction rates and pathways, including CHx (where $x = 1 - 4$), OHx (where

$x = 1 - 2$), and M and M^+ (which are stand-ins for metals such as iron and nickel). The network involves two-body reactions among these species, as well as photochemical reactions induces by UV from the ISRF and reactions initiated by cosmic ray ionizations. In addition to the initial abundances of the various species, the network depends on the ISRF, the ionization rate, and the total abundances of C and O nuclei.

In implementing the NL99 network in DESPOTIC there are a three design choices to be made. First, photochemical and ionization reactions depend on the UV radiation field strength and the ionization rate. When performing computations on a cloud, DESPOTIC takes these from the parameters `chi` and `ionRate` that are part of the radiation class attached to the cloud.

Second, photochemical reactions depend on the amount of shielding against the ISRF provided by dust, and, in the case of the reaction $\mathrm{CO} \rightarrow \mathrm{C} + \mathrm{O}$, line shielding by CO and H_2. Following its usual approximation for implementing such shielding in a one-zone model, DESPOTIC takes the relevant column density to be $N_{\mathrm{H}}/2$, where N_{H} is the column density colDen of the cloud, so that the typical amount of shielding is assumed to correspond to half the area-averaged column density. For the dust shielding, NL99 express the shielding in terms of the V-band extinction A_V; unless instructed otherwise, DESPOTIC computes this via

$$A_V = 0.4\sigma_{\mathrm{PE}}(N_{\mathrm{H}}/2).$$

This ratio of V-band to 100 nm extinction is intermediate between the values expected for Milky Way and SMC dust opacity curves, as discussed in Krumholz, Leroy, & McKee (2011, Astrophysical Journal, 731, 25). However, the user may override this choice. For line shielding, DESPOTIC computes the H_2 and CO column densities

$$N_{\mathrm{H}_2} = x_{\mathrm{H}_2} N_{\mathrm{H}}/2$$
$$N_{\mathrm{CO}} = x_{\mathrm{CO}} N_{\mathrm{H}}/2,$$

which amounts to assuming that the CO and H_2 are uniformly distributed. Note that the NL99 network explicitly assumes $x_{\mathrm{H}_2} = 0.5$, as no reactions involving atomic H are included – see *NL99_GC* for a network that does include hydrogen chemistry. These column densities are then used to find a shielding factor by interpolating the tabulated values of van Dishoeck & Black (1988, Astrophysical Journal, 334, 771).

The third choice is how to map between the species included in the chemistry network and the actual emitting species that are required to compute line emission, cooling, etc. This is non-trivial both because the chemical network includes super-species, because the chemical network does not distinguish between ortho- and para- sub-species while the rest of DESPOTIC does, and because the network does not distinguish between different isotopomers of the same species, while the rest of DESPOTIC does. This does not create problems in mapping from cloud emitter abundances to the chemical network, since the abundances can simply be summed, but it does create a question about how to map output chemical abun- dances computed by the network into the abundances of emitters that can be operated on by the remainder of DESPOTIC. In order to handle this issue, DESPOTIC makes the following assumptions:

1. OHx is divided evenly between OH and OH_2
2. The ratio of ortho- to para- for all species is the same as that of H_2
3. The abundances ratios of all isotopomers of a species remain fixed as reactions occur, so, for example, the final abundance ratio of $\mathrm{C}^{18}\mathrm{O}$ to $\mathrm{C}^{16}\mathrm{O}$ as computed by the chemical network is always the same as the initial one.

7.2.2 `NL99_GC`

The `NL99_GC` network is an extension of the `NL99` network to handle hydrogen chemistry as well as carbon and oxygen chemistry, following the recipe described in Glover & Clark (2012, MNRAS, 421, 116). This network effectively uses `NL99` for carbon and oxygen (with some updates to the rate coefficients) and the network of Glover & Mac Low (2007, ApJS, 169, 239) for the hydrogen chemistry. Self-shielding of hydrogen is handled via the approximate analytic shielding functon of Draine & Bertoldi (1996, ApJ, 468, 269). Effective column densities for shielding are computed as in `NL99`. Calculations of hydrogen chemistry are assumed to leave the ratio of ortho- to para- H_2 unchanged from the initial value, or set it to 0.25 if the initial value is undefined (e.g., because the calculation started with a composition that was all H and no H_2). All other assumptions are completely analogous to `NL99`.

7.3 Defining New Chemical Networks: the `chemNetwork` Class

DESPOTIC implements chemical networks through the abstract base class `chemNetwork`, which is defined by module `despotic.chemistry.chemNetwork` – see *Full Documentation of All DESPOTIC Classes and Functions* for full details. This class defines the required elements that all chemistry networks must contain; users who wish to implement their own chemistry networks must derive them from this class, and must override the class methods listed below. Users are encouraged to examine the two *Predefined Chemical Networks* for examples of how to derive a chemical network class from `chemNetwork`.

For any class `cn` derived from `chemNetwork`, the user is required to define the following non-callable traits:

- `cn.specList`: a list of strings that describes the chemical species included in the network. The names in `cn.specList` can be arbitrary, and are not used for any purpose other than providing human-readable labels on outputs. However, it is often convenient to match the names to the names of emitters, as this makes it convenient to add the emitters back to the cloud later.
- `cn.x`: a numpy array of rank 1, with each element specifying the abundance of a particular species in the network. The number of elements in the array must match the length of `cn.specList`. As with all abundances in DESPOTIC, abundances must be specified relative to H nuclei, so that if, for example, `x[3]` is `0.1`, this means that there is 1 particle of species 3 per 10 H nuclei.
- `cn.cloud`: an instance of the `cloud` class to which the chemical network is attached. This can be `None`, as chemical networks can be free-standing at not attached to specified instances of `cloud`. However, much of the functionality of chemical networks is based around integration with the `cloud` class.

In addition, a class `cn` derived from `chemNetwork` must define the following callable attributes:

- `cn.__init__(self, cloud=None, info=None)`: this is the initialization method. It must accept two keyword arguments. The first, `cloud`, is an instanced of the `cloud` class to which this chemical network will be attached. This routine should set `cn.cloud` equal to the input `cloud` instance, and it may also extract information from the input `cloud` instances in order to initialize `cn.x` or any other required quantities. The second keyword argument, `info`, is a dict containing any additional initialization parameters that the chemical network can be or must be passed upon instantiation.
- `cn.dxdt(self, xin, time)`: this is a method that computes the time derivative of the abundances. Given an input numpy array of abundances `xin` (which is the same shape as `cn.x`) and a time `time`, it must return a numpy array giving the time derivative of all abundances in units of s^{-1}.
- `cn.applyAbundances(self, addEmitters=False)`: this is a method to take the abundances stored in the chemical network and use them to update the abundances of the corresponding emitters in the `cloud` instance associated with this chemical network. This method is called at the end of every chemical calculation, and is responsible for copying information from the chemical network back into the cloud. The optional Boolean argument `addEmitters`, if `True`, specifies that the method should attempt to not only alter the abundances of any emitters associated with the cloud, it should also attempt to add new emitters that are included in the chemical network, and whose abundances are to be determined from it. It is up to the user whether or not to honor this request and implement this behavior. Failure to do so will not prevent the chemical network from operating, but should at least be warned so that other users are not surprised.

Finally, the following is an optional attribute of the derived class `cn`:

- `cn.abundances`: this is a property that returns the abundance information defined in `cn.x` as a object of class *abundanceDict*. This class is a utility class that provides an interface to the chemical abundances in a network that operates like a Python dict. The property abundances is defined in the base `chemNetwork` class, so it is available by inheritance regardless of whether the user defines `cn.abundances`. However, the user may find it convenient to override `chemNetwork.abundances` to provide more information, e.g., to provide abundances of species that are not explicitly evolved in the network, and are instead derived via conservation relations. Both the *NL99* and *NL99_GC* classes do this.

Once a chemical network class that defines the above methods has been defined, that class can be passed as an argument associated with the `network` keyword to the `cloud.setChemEq` and `cloud.chemEvol` methods (and their equivalents in `zonedcloud`), and these methods will automatically perform chemical calculations using the input network.

In setting up chemical networks, it often convenient to make use of the *Helper Modules for Chemical Networks* that are provided.

7.4 Helper Modules for Chemical Networks

The modules below, implemented in `despotic.chemistry`, are intended as helpers for defining and working with chemical networks.

7.4.1 abundanceDict

The `abundanceDict` class provides a dict-like interface to numpy arrays containing species abundances. The motivation for its existence is that it is desirable to keep lists of chemical species abundances in a numpy array for speed of computation, but for humans it is much more convenient to be able to query and print chemical species by name, with a dict-like interface. The `abundanceDict` class overlays a dict-like structure on a numpy array, making it possible to combine the speed of a numpy array with the convenience of a dict.

Usage is simple. To define an `abundanceDict` one simply provides a set of chemical species names and a numpy array containing abundances of those speces:

```
from despotic.chemistry import abundanceDict

# specList = list of strings containing species names
# x = numpy array containing species abundances
abd = abundanceDict(specList, x)
```

Once created, an `abundanceDict` can be handled much like a dict, in which the species names in `specList` are the keys:

```
# Print abundance of the species CO
print abd.abundances["CO"]

# Set the C+ abundance to 1e-10
abd["C+"] = 1e-10
```

These operations will alter the underlying numpy array appropriately; the array may also be accessed directly, as `abundanceDict.x`. However, the performance penalty for referring to objects by name rather than by index in `abundanceDict.x` is negligibly small in almost all cases, so it is recommended to use keys, as this makes for much more human-readable code.

Mathematical operations on the abundances will pass through and be applied to the underlying numpy array:

```
# Double the abundance of every species
abd = 2*abd

# Add the abundances of two different abundanceDict's, abd1 and abd2
abd3 = abd1 + abd2
```

Finally, `abundanceDict` instances differ from regular dict objects in that, once they are created, their species lists are immutable; species abundances can be changed, but species cannot be added or removed.

For full details on `abundanceDict`, see *Full Documentation of All DESPOTIC Classes and Functions*.

7.4.2 `reactions`

The `reactions` module provides a generic way to describe chemical reactions and compute their rates. It contains one basic class and several specialized derived classes to compute particualr types of reactions. The goal of all the classes in `reactions` is to allow users to describe the reactions in a chemical network in human-readable forms, but then compute their rates using efficient numpy operations at high speed.

`reaction_matrix`

The most general class in `reactions` is called `reaction_matrix`. To instantiate it, one provides a list of species and a list of reactions between them:

```
from despotic.chemistry.reactions import reaction_matrix
rm = reaction_matrix(specList, reactions)
```

Here `specList` is a list of strings giving the names of the chemical species, while `reactions` is a list of dict's, one dict per reaction, describing each reaction. Each dict in `reactions` contains two keys: `reactions["spec"]` gives the list of species involved in the reaction, and `reactions["stoich"]` gives the correspnding stoichiometric factors, with reactants on the left hand side having negative factors (indicating that they are destroyed by the reaction) and those on the right having positive factors (indicating that they are created). Thus for example to include the reactions

$$\mathrm{C} + \mathrm{O} \rightarrow \mathrm{CO}$$
$$\mathrm{H} + \mathrm{H} \rightarrow \mathrm{H}_2$$

in a network, one could define the reactions as:

```
reactions \
   = [ { "spec" : ["C", "O", "CO"], "stoich" : [-1, -1, 1] },
       { "spec" : ["H", "H2"], "stoich" : [-2, 1] } ]
```

Once a `reaction_matrix` has been defined, one can compute the rates of change of all species using the method `reaction_matrix.dxdt`:

```
dxdt = rm.dxdt(x, n, ratecoef)
```

Here `x` is a numpy array giving the instantaneous abundances of all species, `n` is the number density of H nuclei, and `ratecoeff` is a numpy array giving the rate coefficient for all reactions. The quantity returned is the instantaneous rate of change of all abundances `x`. The density-dependence of the reaction rates implied by the stoichiometric factors in the reaction list is computed automatically.

`cr_reactions`

The `cr_reactions` class is a specialized class derived from *reaction_matrix* that is intended to handle cosmic ray-induced reactions – those where the rate is proportional to the cosmic ray ionization rate.

Instantiation of a `cr_reactions` object takes the same two arguments as *reaction_matrix*, but `reactions`, the list of reactions to be passed, is altered in that each reaction takes an additional key, `rate`, that gives the proportionality between the reaction rate and the cosmic ray ionization rate. Thus for example if we wished to include the reactions

$$\mathrm{CR} + \mathrm{H} \rightarrow \mathrm{H}^+ + \mathrm{e}^-$$
$$\mathrm{CR} + \mathrm{H}_2 \rightarrow \mathrm{H}_2^+ + \mathrm{e}^-$$

in a network, with the former occuring at a rate per particle equal to the rate of primary cosmic ray ionizations, and the other at a rate per particle that is twice the rate of primary ionizations, we would define:

```
from despotic.chemistry.reactions import cr_reactions
specList = ["H", "H2", "H+", "H2+", "e-"]
reactions \
   = [ { "spec" : ["H", "H+", "e-"], "stoich" : [-1, 1, 1],
         "rate" : 1.0 },
       { "spec" : ["H2", "H2+", "e-"], "stoich" : [-1, 1, 1],
         "rate" : 2.0 } ]
cr = cr_reactions(specList, reactions)
```

Once a `cr_reactions` object is instantiated, it can be used to compute the rates of change all species using the `cr_reactions.dxdt` routine:

```
dxdt = cr.dxdt(x, n, ionrate)
```

Here `x` is a numpy array giving the current abundances, `n` is the number density of H nuclei, and `ionrate` is the cosmic ray primary ionization rate.

Note that, in most cases, the variable `n` will not be used, because it is needed only if there is more than one reactant on the left hand side of a reaction. It is provided to enable the case where a cosmic ray ionization is followed immediately by another reaction, and the network combines the two steps for speed. For example, the *NL99* network combines the two reactions

$$\mathrm{CR} + \mathrm{H}_2 \rightarrow \mathrm{H}_2^+ + \mathrm{e}^-$$
$$\mathrm{H}_2^+ + \mathrm{H}_2 \rightarrow \mathrm{H}_3^+ + \mathrm{H} + \mathrm{e}^-$$

into the single super-reaction

$$\mathrm{CR} + 2\mathrm{H}_2 \rightarrow \mathrm{H}_3^+ + \mathrm{H} + \mathrm{e}^-$$

and the rate of this super-reaction does depend on density.

`photoreactions`

The `photoreactions` class is a specialized class derived from *reaction_matrix* that is intended to handle reactions that are driven by FUV photons, and thus have rates proportional to the interstellar radiation field (ISRF) strength, modified by dust and gas shielding, and possibly also by line shielding.

Instantiation of `photoreactions` takes the same two arguments as *reaction_matrix*: a list of species, and a list of reactions, each element of which is a dict giving the reactants and stoichiometric factors. The dict also contains three additional keys that are specific to photoreactions: `rate` gives the reaction rate per reactant per second in an ISRF equal to the unshielded Solar neighborhood value (formally, with strength chracterized by $\chi = 1$ – see *Cloud file keys and their meanings*). The key `av_fac` describes the optical depth per unit visual extinction provided by dust for the photons driving the reaction. That is, reaction rates will be reduced by a factor $\exp(-\mathrm{av\_fac} \times A_V)$. Finally, the key `shield_fac`, which is optional, can be set equal to a callable that describes the rate by which the reaction rate is reduced due to line shielding. This function can take any number of argument – see below. Thus the final reaction rate per reactant will be equal to

$$\mathrm{reaction\,rate} = \chi \times \mathrm{rate} \times \mathrm{shield\_fac} \times \exp(-\mathrm{av\_fac} \times A_V)$$

As an example, suppose we wished to include the reactions

$$\gamma + \mathrm{C} \rightarrow \mathrm{C}^+ + \mathrm{e}^-$$
$$\gamma + \mathrm{CO} \rightarrow \mathrm{C} + \mathrm{O}$$

The first reaction occurs at a rate per C atom $5.1 \times 10^{-10}\,\mathrm{s}^{-1}$ in unshielded space, and the optical depth to the photons producing the reaction is $3.0 \times A_V$. There is no significant self-shielding. The second reaction occurs at a rate per CO molecule $1.7 \times 10^{-10}\,\mathrm{s}^{-1}$ in unshielded space, with a dust optical depth equal to $1.7 \times A_V$, and with an additional function describing line shielding called `fShield_CO`. We could create a `photoeactions` object containing this reaction by doing:

```
from despotic.chemistry.reactions import photoreactions
specList = ["C", "C+", "e-", "O", "CO"]
reactions \
   = [ { "spec" : ["C", "C+", "e-"], "stoich" : [-1, 1, 1],
         "rate" : 5.1e-10, "av_fac" : 3.0 },
       { "spec" : ["CO", "C", "O"], "stoich" : [-1, 1, 1],
         "rate" : 1.7e-10, "av_fac" : 1.7,
         "shield_fac" : fShield_CO } ]
phr = photoreactions(specList, reactions)
```

Once a photoreactions object exists, the rates of change of all species due to the included photoreactions can be computed using the `photoreactions.dxdt` routine. Usage is as follows:

```
dxdt = phr.dxdt(x, n, chi, AV,
                shield_args=[[f1_arg1, f1_arg2, ...],
                             [f2_arg1, f2_arg2, ...],
                             ... ],
                shield_kw_args = [ { "f1_kw1" : val1,
                                     "f1_kw2" : val2, ... },
                                   { "f2_kw1" : val1,
                                     "f2_kw2" : val2, ... },
                                   ... ])
```

Here `x` is a numpy array giving the current abundances, `n` is the number density of H nuclei, `chi` is the unshielded ISRF strength, and `AV` is the dust visual extinction. The optional arguments `shield_args` and `shield_kw_args` are used to pass positional arguments and keyword arguments, respectively, to the shielding functions. Each one is a list. The first entry in the list for `shield_args` is a list containing the positional arguments to be passed to the first shielding function provided in the `reactions` dict passed to the constructor. The second entry in `shield_args` is a list containing positional arguments to be passed to the second shielding function, and so forth. Similarly, the first entry in the `shield_kw_args` list is a dict containing the keywords and their values to be passed to the first shielding function, etc. The length of the `shield_args` and `shield_kw_args` must be equal to the number of shielding functions provided in `reactions`, but elements of the lists can be set to `None` if a particular shielding function does not take positional or keyword arguments. Moreover, both `shield_args` and `shield_kw_args` are optional. If they are omitted, then all shielding functions are invoked with no positional or keyword arguments, respectively.

Thus in the above example, suppose that the function `fShield_CO` returns the factor by which the reaction rate is reduced by CO line shielding. It takes two arguments: `NH2` and `NCO`, giving the column densities of H_2 and CO, respectively. It also accepts a keyword argument `order` that specifies the order of interpolation to be used in computing the shielding function from a table. In this case one could compute the rates of change of the abundances via:

```
dxdt = phr.dxdt(x, n, chi, AV,
                shield_args = [[NH2, NCO]],
                shield_kw_args = [ {"order" : 2} ])
```

This would call `fShield_CO` as:

```
fShield_CO(NH2, NCO, order=2)
```

CHAPTER

EIGHT

FULL DOCUMENTATION OF ALL DESPOTIC CLASSES AND FUNCTIONS

8.1 despotic classes

8.1.1 `cloud`

class `despotic.`**`cloud`**(*fileName=None*, *verbose=False*)

A class describing the properties of an interstellar cloud, and providing methods to perform calculations using those properties.

Parameters

fileName [string] name of file from which to read cloud description

verbose [Boolean] print out information about the cloud as we read it

Class attributes

nH [float] number density of H nuclei, in cm^-3

colDen [float] center-to-edge column density of H nuclei, in cm^-2

sigmaNT [float] non-thermal velocity dispersion, in cm s^-1

dVdr [float] radial velocity gradient, in s^-1 (or cm s^-1 cm^-1)

Tg [float] gas kinetic temperature, in K

Td [float] dust temperature, in K

comp [class composition] a class that stores information about the chemical composition of the cloud

dust [class dustProp] a class that stores information about the properties of the dust in a cloud

rad [class radiation] the radiation field impinging on the cloud

emitters [dict] keys of the dict are names of emitting species, and values are objects of class emitter

chemnetwork [chemical network class (optional)] a chemical network that is to be used to perform time-dependent chemical evolution calcualtions for this cloud

`abundances`

This property contains the abundances of all emitting species, stored as an abundanceDict

`addEmitter`(*emitName*, *emitAbundance*, *emitterFile=None*, *emitterURL=None*, *energySkip=False*, *extrap=True*)

Method to add an emitting species

Pamameters

emitName [string] name of the emitting species

emitAbundance [float] abundance of the emitting species relative to H

emitterFile [string] name of LAMDA file containing data on this species; this option overrides the default

emitterURL [string] URL of LAMDA file containing data on this species; this option overrides the default

energySkip [Boolean] if set to True, this species is ignored when calculating line cooling rates

extrap [Boolean] If set to True, collision rate coefficients for this species will be extrapolated to temperatures outside the range given in the LAMDA table. If False, no extrapolation is perfomed, and providing temperatures outside the range in the table produces an error

Returns Nothing

`chemEvol`(*tFin*, *tInit=0.0*, *nOut=100*, *dt=None*, *tOut=None*, *network=None*, *info=None*, *addEmitters=False*, *evolveTemp='fixed'*, *isobaric=False*, *tempEqParam=None*, *dEdtParam=None*)

Evolve the chemical abundances of this cloud in time.

Parameters

tFin [float] end time of integration, in sec

tInit [float] start time of integration, in sec

nOut [int] number of times at which to report the temperature; this is ignored if dt or tOut are set

dt [float] time interval between outputs, in sec; this is ignored if tOut is set

tOut [array] list of times at which to output the temperature, in sec; must be sorted in increasing order

network [chemical network class] a valid chemical network class; this class must define the methods __init__, dxdt, and applyAbundances; if None, the existing chemical network for the cloud is used

info [dict] a dict of additional initialization information to be passed to the chemical network class when it is instantiated

addEmitters [Boolean] if True, emitters that are included in the chemical network but not in the cloud's existing emitter list will be added; if False, abundances of emitters already in the emitter list will be updated, but new emiters will not be added to the cloud

evolveTemp ['fixed' | 'gasEq' | 'fullEq' | 'evol'] how to treat the temperature evolution during the chemical evolution; 'fixed' = treat tempeature as fixed; 'gasEq' = hold dust temperature fixed, set gas temperature to instantaneous equilibrium value; 'fullEq' = set gas and dust temperatures to instantaneous equilibrium values; 'evol' = evolve gas temperature in time along with the chemistry, assuming the dust is always in instantaneous equilibrium

isobaric [Boolean] if set to True, the gas is assumed to be isobaric during the evolution (constant pressure); otherwise it is assumed to be isochoric; note that (since chemistry networks at present are not allowed to change the mean molecular weight), this option has no effect if evolveTemp is 'fixed'

tempEqParam [None | dict] if this is not None, then it must be a dict of values that will be passed as keyword arguments to the cloud.setTempEq, cloud.setGasTempEq, or cloud.setDustTempEq routines; only used if evolveTemp is not 'fixed'

dEdtParam [None | dict] if this is not None, then it must be a dict of values that will be passed as keyword arguments to the cloud.dEdt routine; only used if evolveTemp is 'evol'

Returns

time [array of floats] array of output times, in sec

abundances [class abundanceDict] an abundanceDict giving the abundances as a function of time

Tg [array] gas temperature as a function of time; returned only if evolveTemp is not 'fixed'

Td [array] dust temperature as a function of time; returned only if evolveTemp is not 'fixed' or 'gasEq'

Raises despoticError, if network is None and the cloud does not already have a defined chemical network associated with it

chemabundances

The property contains the abundances of all species in the chemical network, stored as an abundanceDict

dEdt (*c1Grav=0.0*, *thin=False*, *LTE=False*, *fixedLevPop=False*, *noClump=False*, *escapeProbGeom='sphere'*, *PsiUser=None*, *sumOnly=False*, *dustOnly=False*, *gasOnly=False*, *dustCoolOnly=False*, *dampFactor=0.5*, *verbose=False*, *overrideSkip=False*)

Return instantaneous values of heating / cooling terms

Parameters

c1Grav [float] if this is non-zero, the cloud is assumed to be collapsing, and energy is added at a rate Gamma_grav = c1 mu_H m_H cs^2 sqrt(4 pi G rho)

thin [Boolean] if set to True, cloud is assumed to be opticall thin

LTE [Boolean] if set to True, gas is assumed to be in LTE

fixedLevPop [Boolean] if set to True, level populations and escape probabilities are not recomputed, so the cooling rate is based on whatever values are stored

escapeProbGeom [string, 'sphere' or 'LVG' or 'slab'] specifies the geometry to be assumed in calculating escape probabilities

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

dampFactor [float] damping factor to use in level population calculations; see emitter.setLevPopEscapeProb

PsiUser [callable] A user-specified function to add additional heating / cooling terms to the calculation. The function takes the cloud object as an argument, and must return a two-element array Psi, where Psi[0] = gas heating / cooling rate, Psi[1] = dust heating / cooling rate. Positive values indicate heating, negative values cooling, and units are assumed to be erg s^-1 H^-1.

sumOnly [Boolean] if true, rates contains only four entries: dEdtGas and dEdtDust give the heating / cooling rates for the gas and dust summed over all terms, and maxAbsdEdtGas and maxAbsdEdtDust give the largest of the absolute values of any of the contributing terms for dust and gas

gasOnly [Boolean] if true, the terms GammaISRF, GammaDustLine, LambdaDust, and PsiUserDust are omitted from rates. If both gasOnly and sumOnly are true, the dict contains only dEdtGas

dustOnly [Boolean] if true, the terms GammaPE, GammaCR, LambdaLine, GamamDLine, and PsiUserGas are omitted from rates. If both dustOnly and sumOnly are true, the dict contains only dEdtDust. Important caveat: the value of dEdtDust returned in this case will not exactly match that returned if dustOnly is false, because it will not contain the contribution from gas line cooling radiation that is absorbed by the dust

dustCoolOnly [Boolean] as dustOnly, but except that now only the terms LambdaDust, PsiGD, and PsiUserDust are computed

overrideSkip [Boolean] if True, energySkip directives are ignored, and cooling rates are calculated for all species

Returns

rates [dict] A dict containing the values of the various heating and cooling rate terms; all quantities are in units of erg s^-1 H^-1, and by convention positive = heating, negative = cooling; for dust-gas exchange, positive indicates heating of gas, cooling of dust

Elements of the dict are as follows by default, but can be altered by the additional keywords listed below:

GammaPE [float] photoelectric heating rate

GammaCR [float] cosmic ray heating rate

GammaGrav [float] gravitational contraction heating rate

LambdaLine [dict] cooling rate from lines; dictionary keys correspond to species in the emitter list, values give line cooling rate for that species

PsiGD [float] dust-gas energy exchange rate

GammaDustISRF [float] dust heating rate due to the ISRF

GammaDustCMB [float] dust heating rate due to CMB

GammaDustIR [float] dust heating rate due to IR field

GammaDustLine [float] dust heating rate due to absorption of line radiation

LambdaDust [float] dust cooling rate due to thermal emission

PsiUserGas [float] gas heating / cooling rate from user-specified function; only included if PsiUser != None

PsiUserDust [float] gas heating / cooling rate from user-specified function; only included is PsiUser != None

`lineLum`(*emitName*, *LTE=False*, *noClump=False*, *transition=None*, *thin=False*, *intOnly=False*, *TBOnly=False*, *lumOnly=False*, *escapeProbGeom='sphere'*, *dampFactor=0.5*, *noRecompute=False*, *abstol=1e-08*, *verbose=False*)

Return the frequency-integrated intensity of various lines

Parameters

emitName [string] name of the emitter for which the calculation is to be performed

LTE [Boolean] if True, and level populations are unitialized, they will be initialized to their LTE values; if they are initialized, this option is ignored

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

transition [list of two arrays] if left as None, luminosity is computed for all transitions; otherwise only selected transitions are computed, with transition[0] = array of upper states transition[1] = array of lower states

thin [Boolean] if True, the calculation is done assuming the cloud is optically thin; if level populations are uninitialized, and LTE is not set, they will be computed assuming the cloud is optically thin

intOnly [Boolean] if true, the output is simply an array containing the frequency-integrated intensity of the specified lines; mutually exclusive with TBOnly and lumOnly

TBOnly [Boolean] if True, the output is simply an array containing the velocity-integrated brightness temperatures of the specified lines; mutually exclusive with intOnly and lumOnly

lumOnly [Boolean] if True, the output is simply an array containing the luminosity per H nucleus in each of the specified lines; mutually eclusive with intOnly and TBOonly

escapeProbGeom ['sphere' | 'LVG' | 'slab'] sets problem geometry that will be assumed in calculating escape probabilities; ignored if the escape probabilities are already initialized

dampFactor [float] damping factor to use in level population calculations; see emitter.setLevPopEscapeProb

noRecompute [False] if True, level populations and escape probabilities are not recomputed; instead, stored values are used

Returns res : list or array

if intOnly, TBOnly, and lumOnly are all False, each element of the list is a dict containing the following fields:

'freq' [float] frequency of the line in Hz

'upper' [int] index of upper state, with ground state = 0 and states ordered by energy

'lower' [int] index of lower state

'Tupper' [float] energy of the upper state in K (i.e. energy over kB)

'Tex' [float] excitation temperature relating the upper and lower levels

'intIntensity' [float] frequency-integrated intensity of the line, with the CMB contribution subtracted off; units are erg cm^-2 s^-1 sr^-1

'intTB' [float] velocity-integrated brightness temperature of the line, with the CMB contribution subtracted off; units are K km s^-1

'lumPerH' [float] luminosity of the line per H nucleus; units are erg s^-1 H^-1

'tau' [float] optical depth in the line, not including dust

'tauDust' [float] dust optical depth in the line

if intOnly, TBOnly, or lumOnly are True: res is an array containing the intIntensity, TB, or lumPerH fields of the dict described above

read (*fileName*, *verbose=False*)

Read the composition from a file

Pamameters

fileName [string] string giving the name of the composition file

verbose [Boolean] print out information about the cloud as it is read

Returns Nothing

Remarks For the format of cloud files, see the documentation

setChemEq (*tEqGuess=None*, *network=None*, *info=None*, *addEmitters=False*, *tol=1e-06*, *maxTime=1e+16*, *verbose=False*, *smallabd=1e-15*, *convList=None*, *evolveTemp='fixed'*, *isobaric=False*, *tempEqParam=None*, *dEdtParam=None*, *maxTempIter=50*)

Set the chemical abundances for a cloud to their equilibrium values, computed using a specified chemical network.

Parameters

tEqGuess [float] a guess at the timescale over which equilibrium will be achieved; if left unspecified, the code will attempt to estimate this time scale on its own

network [chemNetwork object] the chemNetwork object to use; if None, the existing chemnetwork member of the class (if it exists) is used

info [dict] a dict of additional initialization information to be passed to the chemical network class when it is instantiated

addEmitters [Boolean] if True, emitters that are included in the chemical network but not in the cloud's existing emitter list will be added; if False, abundances of emitters already in the emitter list will be updated, but new emiters will not be added to the cloud

evolveTemp ['fixed' | 'iterate' | 'iterateDust' | 'gasEq' | 'fullEq' | 'evol'] how to treat the temperature evolution during the chemical evolution; 'fixed' = treat tempeature as fixed; 'iterate' = iterate between setting the gas temperature and chemistry to equilibrium; 'iterateDust' = iterate between setting the gas and dust temperatures and the chemistry to equilibrium; 'gasEq' = hold dust temperature fixed, set gas temperature to instantaneous equilibrium value as the chemistry evolves; 'fullEq' = set gas and dust temperatures to instantaneous equilibrium values while evolving the chemistry network; 'evol' = evolve gas temperature in time along with the chemistry, assuming the dust is always in instantaneous equilibrium

isobaric [Boolean] if set to True, the gas is assumed to be isobaric during the evolution (constant pressure); otherwise it is assumed to be isochoric; note that (since chemistry networks at present are not allowed to change the mean molecular weight), this option has no effect if evolveTemp is 'fixed'

tempEqParam [None | dict] if this is not None, then it must be a dict of values that will be passed as keyword arguments to the cloud.setTempEq, cloud.setGasTempEq, or cloud.setDustTempEq routines; only used if evolveTemp is not 'fixed'

dEdtParam [None | dict] if this is not None, then it must be a dict of values that will be passed as keyword arguments to the cloud.dEdt routine; only used if evolveTemp is 'evol'

tol [float] tolerance requirement on the equilibrium solution

convList [list] list of species to include when calculating tolerances to decide if network is converged; species not listed are not considered. If this is None, then all species are considered in deciding if the calculation is converged.

smallabd [float] abundances below smallabd are not considered when checking for convergence; set to 0 or a negative value to consider all abundances, but beware that this may result in false non-convergence due to roundoff error in very small abundances

maxTempIter [int] maximum number of iterations when iterating between chemistry and temperature; only used if evolveTemp is 'iterate' or 'iterateDust'

verbose [Boolean] if True, diagnostic information is printed as the calculation proceeds

Returns

converged [Boolean] True if the calculation converged, False if not

Raises despoticError, if network is None and the cloud does not already have a defined chemical network associated with it

Remarks The final abundances are written to the cloud whether or not the calculation converges.

`setDustTempEq`(*PsiUser=None*, *Tdinit=None*, *noLines=False*, *noClump=False*, *verbose=False*, *dampFactor=0.5*)

Set Td to equilibrium dust temperature at fixed Tg

Parameters

Tdinit [float] initial guess for gas temperature

PsiUser [callable] A user-specified function to add additional heating / cooling terms to the calculation. The function takes the cloud object as an argument, and must return a two-element array Psi, where Psi[0] = gas heating / cooling rate, Psi[1] = dust heating / cooling rate. Positive values indicate heating, negative values cooling, and units are assumed to be erg s^-1 H^-1.

noLines [Boolean] If True, line heating of the dust is ignored. This can make the calculation significantly faster.

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

dampFactor [float] damping factor to use in level population calculations; see emitter.setLevPopEscapeProb

verbose [Boolean] if True, diagnostic information is printed

Returns

success [Boolean] True if dust temperature calculation converged, False if not

setGasTempEq(*c1Grav=0.0*, *thin=False*, *noClump=False*, *LTE=False*, *Tginit=None*, *fixedLevPop=False*, *escapeProbGeom='sphere'*, *PsiUser=None*, *verbose=False*)

Set Tg to equilibrium gas temperature at fixed Td

Parameters

c1Grav [float] if this is non-zero, the cloud is assumed to be collapsing, and energy is added at a rate Gamma_grav = c1 mu_H m_H cs^2 sqrt(4 pi G rho)

thin [Boolean] if set to True, cloud is assumed to be opticall thin

LTE [Boolean] if set to True, gas is assumed to be in LTE

Tginit [float] initial guess for gas temperature

fixedLevPop [Boolean] if set to True, level populations are held fixed at the starting value, rather than caclculated self-consistently from the temperature

escapeProbGeom ['sphere' | 'LVG' | 'slab'] specifies the geometry to be assumed in computing escape probabilities

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

PsiUser [callable] A user-specified function to add additional heating / cooling terms to the calculation. The function takes the cloud object as an argument, and must return a two-element array Psi, where Psi[0] = gas heating / cooling rate, Psi[1] = dust heating / cooling rate. Positive values indicate heating, negative values cooling, and units are assumed to be erg s^-1 H^-1.

verbose [Boolean] if True, print status messages while running

Returns

success [Boolean] True if the calculation converges, False if it does not

setTempEq(*c1Grav=0.0*, *thin=False*, *noClump=False*, *LTE=False*, *Tinit=None*, *fixedLevPop=False*, *escapeProbGeom='sphere'*, *PsiUser=None*, *verbose=False*, *tol=0.0001*)

Set Tg and Td to equilibrium gas and dust temperatures

Parameters

c1Grav [float] coefficient for rate of gas gravitational heating

thin [Boolean] if set to True, cloud is assumed to be opticall thin

LTE [Boolean] if set to True, gas is assumed to be in LTE

Tinit [array(2)] initial guess for gas and dust temperature

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

fixedLevPop [Boolean] if set to true, level populations are held fixed at the starting value, rather than caclculated self-consistently from the temperature

escapeProbGeom ['sphere' | 'LVG' | 'slab'] specifies the geometry to be assumed in computing escape probabilities

PsiUser [callable] A user-specified function to add additional heating / cooling terms to the calculation. The function takes the cloud object as an argument, and must return a two-element array Psi, where Psi[0] = gas heating / cooling rate, Psi[1] = dust heating / cooling rate. Positive values indicate heating, negative values cooling, and units are assumed to be erg s^-1 H^-1.

verbose [Boolean] if True, the code prints diagnostic information as it runs

Returns

success [Boolean] True if the iteration converges, False if it does not

`setVirial`(*alphaVir=1.0*, *setColDen=False*, *setnH=False*, *NTonly=False*)
Set sigmaNT, colDen, or nH to the value required to give a virial ratio of unity

Parameters

alphaVir [float] virial ratio to be used in computation; defaults to 1

setColDen [Boolean] if True, sigmaNT and nH are fixed, and colDen is altered to give the desired virial ratio

setnH [Boolean] if True, sigmaNT and colDen are fixed, and nH is altered to give the desired virial ratio

NTonly [Boolean] if True, the virial ratio is computed considering only the non-thermal component of the velocity dispersion

Returns Nothing

Remarks By default the routine fixes nH and colDen and computes sigmaNT, but this can be overridden by specifying either setColDen or setnH. It is an error to set both of these to True.

`tempEvol`(*tFin*, *tInit=0.0*, *c1Grav=0.0*, *noClump=False*, *thin=False*, *LTE=False*, *fixedLevPop=False*, *escapeProbGeom='sphere'*, *nOut=100*, *dt=None*, *tOut=None*, *PsiUser=None*, *isobaric=False*, *fullOutput=False*, *dampFactor=0.5*, *verbose=False*)
Calculate the evolution of the gas temperature in time

Parameters

tFin [float] end time of integration, in sec

tInit [float] start time of integration, in sec

c1Grav [float] coefficient for rate of gas gravitational heating

thin [Boolean] if set to True, cloud is assumed to be opticall thin

LTE [Boolean] if set to True, gas is assumed to be in LTE

isobaric [Boolean] if set to True, cooling is calculated isobarically; otherwise (default behavior) it is computed isochorically

fixedLevPop [Boolean] if set to true, level populations are held fixed at the starting value, rather than caclculated self-consistently from the temperature

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

escapeProbGeom [string, 'sphere' or 'LVG' or 'slab'] specifies the geometry to be assumed in computing escape probabilities

nOut [int] number of times at which to report the temperature; this is ignored if dt or tOut are set

dt [float] time interval between outputs, in s; this is ignored if tOut is set

tOut [array] list of times at which to output the temperature, in s; must be sorted in increasing order

PsiUser [callable] A user-specified function to add additional heating / cooling terms to the calculation. The function takes the cloud object as an argument, and must return a two-element array Psi, where Psi[0] = gas heating / cooling rate, Psi[1] = dust heating / cooling rate. Positive values indicate heating, negative values cooling, and units are assumed to be erg s^-1 H^-1.

fullOutput [Boolean] if True, the full cloud state is returned at every time, as opposed to simply the gas temperature

dampFactor [float] damping factor to use in calculating level populations (see emitter for details)

Returns if fullOutput == False:

Tg [array] array of gas temperatures at specified times, in K

time [array] array of output times, in sec

if fullOutput == True:

cloudState [list] each element of the list is a deepcopy of the cloud state at the corresponding time; there is one list element per output time

time [array of floats] array of output times, in sec

Remarks If the settings for nOut, dt, or tOut are such that some of the output times requested are past tEvol, the cloud will only be evolved up to time tEvol. Similarly, if the last output time is less than tEvol, the cloud will still be evolved up to time tEvol, and the gas temperature Tg will be set to its value at that time.

8.1.2 `collPartner`

class `despotic.`**`collPartner`**(*fp*, *nlev*, *extrap=True*)

A utility class to store information about a particular collision partner for a given species, and to interpolate collision rates from those data

Parameters

fp [file] a file object that points to the start of the collision rate data for one species in a LAMDA file

nlev [int] number of levels for the emitting species

extrap [Boolean] if True, then computing the collision rate with a temperature that is outside the table will result in the maximum or minimum value in the table being returned; if False, it will raise an error

Class attributes

nlev [int] number of energy levels for the emitting species

ntrans [int] number of collisional transitions in the data table

ntemp [int] number of temperatures in the data table

tempTable [array(ntemp)] list of temperatues at which collision rate coefficients are given

colUpper [int array(ntrans)] list of upper states for collisions

colLower [int array(ntrans)] list of lower states for collisions

colRate [array(ntrans, ntemp)] table of downward collision rate coefficients, in cm^3 s^-1

colRateInterp [list(ntrans) of functions] each function in the list takes one variable, the temperature, as an argument, and returns the collision rate coefficient for the corresponding transition at the given temperature; only downard transitions are included

`colRateMatrix`(*temp*, *levWgt*, *levTemp*)

Return interpolated collision rates for all transitions at a given temperature, stored as an nlev x nlev matrix.

Parameters

temp [float] temperature at which collision rates are computed, in K

levWgt [array] array of statistical weights for each level

levTemp [array] array of level energies, measured in K

Returns

k [array, shape (nlev, nlev)] collision rates at the specified temperature; element i,j of the matrix gives the collision rate from state i to state j

`colRates`(*temp*, *transition=None*)

Return interpolated collision rates for all transitions at a given temperature or list of temperatures

Parameters

temp [float | array] temperature(s) at which collision rates are computed, in K

transition [array of int, shape (2, N)] list of upper and lower states for which collision rates are to be computed; default behavior is to computer for all known transitions

Returns

rates [array, shape (ntrans) | array, shape (ntrans, ntemp)] collision rates at the specified temperature(s)

8.1.3 `composition`

class `despotic.composition`

A class describing the chemical composition of the dominant components (hydrogen, helium, electrons) of an interstellar cloud, and for computing various quantities from them.

Parameters None

Class attributes

xHI [float] abundance of HI per H nucleus

xoH2 [float] abundance of ortho-H2 per H nucleus (note that the maximum possible value of xoH2 is 0.5, since it is per H nucleus)

xpH2 [float] abundance of para-H2 per H nucleus (note that the maximum possible value of xoH2 is 0.5, since it is per H nucleus)

xH2 [float] sum of xoH2 and xpH2

xHe [float] abundance of He per H nucleus

xe [float] abundance of free electrons per H nucleus

xHplus [float] abundance of H+ per H nucleus

mu [float] mean mass per free particle, in units of H mass

muH [float] mean mass per H nucleus, in units of H mass

qIon [float] energy added to the gas per primary CR / x-ray ionization

cv [float] dimensionless specific heat per H nucleus at constant volume; the usual specific heat per unit volume may be obtained by multiplying this by nH * kB, and the specific heat per unit mass may be obtained by multiplying by nH * muH * kB

`computeCv`(*T*, *noSet=False*, *Jmax=40*)

Compute the dimensionless specific heat per H nucleus; the dimensional specific heat per H nucleus is this value multiplied by kB, the dimensional specific heat per unit volume is this value multiplied by kB * nH, and the dimensional specific heat per unit mass is this value multiplied by kB * nH * muH

Parameters

T [float | array] temperature in K

noSet [Boolean] if True, the value of cv stored in the class is not altered, but the calculated cv is still returned

Jmax [int] maximum J to be used in evaluating the rotational partition function; should be set to a value such that T << J(J+1) * thetaRot, there thetaRot = 85.3 K. Defaults to 40.

Returns

cv [float | array] value of cv

`computeDerived`(*nH*)

Compute the derived quantities mu, muH, qIon

Parameters

nH [float] volume density in H cm^-3

Returns Nothing

Remarks For the purposes of this procedure, we treat electrons as massless.

`computeEint`(*T*, *Jmax=40*)

Compute the dimensionless internal energy per H nucleus; the internal energy per H nucleus in K is this value multiplied by T, and the internal energy per H nucleus in erg is this value multiplied by kB * T

Parameters

T [float | array] temperature in K

Jmax [int] maximum J to be used in evaluating the rotational partition function; should be set to a value such that T << J(J+1) * thetaRot, there thetaRot = 85.3 K. Defaults to 40.

Returns

Eint [float | array] value of Eint

8.1.4 `despoticError`

class `despotic.`**`despoticError`**(*message*)

Class derived from Exception to handle exceptions raised by DESPOTIC-specific errors.

Parameters

message [string] the error message

8.1.5 `dustProp`

class `despotic.`**`dustProp`**

A class to hold paremeters describing the properties of dust grains.

Parameters None

Class attributes

sigma10 [float] dust opacity to thermal radiation at 10 K, in cm^2 H^-1

sigmaPE [float] dust opacity averaged over 8 - 13.6 eV, the range that dominates grain photoelectric heatin

sigmaISRF [float] dust opacity averaged the range that dominates grain starlight heating

Zd [float] dust abundance normalized to solar neighborhood value

beta [float] dust spectral index in the mm, sigma ~ nu^beta

alphaGD [float] grain-gas coupling coefficient

8.1.6 `emitter`

class `despotic.`**`emitter`**(*emitName*, *emitAbundance*, *extrap=True*, *energySkip=False*, *emitterFile=None*, *emitterURL=None*)

Class to store the properties of a single emitting species, and preform computations on those properties. Note that all quantities are stored in cgs units.

Parameters

emitName [string] name of this species

emitAbundance [float] abundance of species relative to H

emitterFile [string] name of LAMDA file containing data on this species; this option overrides the default

emitterURL [string] URL of LAMDA file containing data on this species; this option overrides the default

energySkip [Boolean] if True, this species is skipped when computing line cooling rates

extrap [Boolean] if True, collision rate coefficients for this species will be extrapolated to temperatures outside the range given in the LAMDA tables

Returns Nothing

Class attributes

name [string] name of emitting species

abundance [float] abundance of emitting species relative to H nuclei

data [class emitterData] physical data for this emitter

levPop [array, shape(nlev)] array giving populations of each level

levPopInitialized [Boolean] flag for whether levPop is initialized or uninitialized

escapeProb [array, shape (nlev, nlev)] 2d array giving escape probability for photons emitted in a line connecting the given level pair

escapeProbInitialized [Boolean] flag for whether escapeProb is initialized or uninitialized

energySkip [Boolean] flag that this species should be skipped when computing line cooling rates `luminosityPerH` (*rad*, *transition=None*, *total=False*, *thin=False*)

Return the luminosities of various lines, computed from the stored level populations and escape probabilities.

Parameters

- **rad** [class radiation] radiation field impinging on the cloud
- **transition** [list] A list containing two 1D arrays of integer type; transition[0] = array of upper states, transition[1] = array of lower states
- **total** [Boolean] if True, the routine returns a single float rather than an array; this float is the sum of the luminosities per H nucleus of all lines

Returns

- **lum** [array] luminosities per H in specified lines; by default

Raises despoticError, if the level populations are not initialized, or if the escape probabilities are not initialized and thin is not True

`opticalDepth` (*transition=None*, *escapeProbGeom='sphere'*)

Return the optical depths of various lines, computed from the stored escape probabilities.

Parameters

- **transition** [list] A list containing two 1D arrays of integer type; transition[0] = array of upper states, transition[1] = array of lower states
- **escapeProbGeom** ['sphere' | 'LVG' | 'slab'] sets problem geometry that will be assumed in calculating escape probabilities

Returns

- **tau** [array] optical depths in specified lines; by default

`setEscapeProb` (*thisCloud*, *transition=None*, *escapeProbGeom='sphere'*)

Compute escape probabilities from stored level populations

Parameters

- **thisCloud** [class cloud] a cloud object containing the physical and chemical properties of this cloud
- **transition** [list] list of transition for which to set escape probability; transition[0] = array of upper states, transition[1] = array of lower states
- **escapeProbGeom** ['sphere' | 'LVG' | 'slab'] sets problem geometry that will be assumed in calculating escape probabilities

Returns Nothing

`setLevPop` (*thisCloud*, *thin=False*, *noClump=False*, *diagOnly=False*)

Compute the level populations for this species using the stored escape probabilities

Parameters

- **thisCloud** [class cloud] a cloud object containing the physical and chemical properties of this cloud
- **thin** [Boolean] if True, the stored escape probabilities are ignored, and the cloud is assumed to be optically thin (equivalent to assuming all escape probabilities are 1)
- **noClump** [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity
- **diagOnly** [Boolean] if true, diagnostic information is returned, but no attempt is made to solve the equations or calculate the level popuplations (useful for debugging)

Returns

infoDict [dict] A dictionary containing a variety of diagnostic information

setLevPopEscapeProb(*thisCloud*, *escapeProbGeom='sphere'*, *noClump=False*, *verbose=False*, *reltol=1e-06*, *abstol=1e-08*, *maxiter=200*, *veryverbose=False*, *dampFactor=0.5*)

Compute escape probabilities and level populations simultaneously.

Parameters

thisCloud [class cloud] a cloud object containing the physical and chemical properties of this cloud

escapeProbGeom ['sphere' | 'LVG' | 'slab'] sets problem geometry that will be assumed in calculating escape probabilities

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

Returns

success: Boolean True if iteration converges, False if it does not

Additional Parameters

verbose [Boolean] if True, diagnostic information is printed

veryverbose [Boolean] if True, a very large amount of diagnostic information is printed; probably useful only for debugging

reltol [float] relative tolerance; convergence is considered to have occured when the absolute value of the difference between two iterations, divided by the larger of the two results being differences, is less than reltol

abstol [float] absolute tolerance; convergence is considered to have occured when the absolute value of the difference between two iterations is less than abstol

maxiter [int] maximum number of iterations to allow

dampFactor [float] a number in the range (0, 1] that damps out changes in level populations at each iteration. A value of 1 means no damping, while a value of 0 means the level populations never change.

Remarks Convergence occurs when either the relative or the absolute tolerance condition is satisfied. To disable either relative or absolute tolerance checking, just set the appropriate tolerance <= 0. However, be warned that in many circumstances disabling absolute tolerances will gaurantee non-convergence, because truncation errors tend to produce large relative residuals for high energy states whose populations are very low, and no amount of iterating will reduce these errors substantially.

setLevPopLTE(*temp*)

Set the level populations of this species to their LTE values

Parameters

temp [float] temperature in K

Returns Nothing

setThin()

Set the escape probabilities for this species to unity

Parameters None

Returns Nothing

8.1.7 `emitterData`

class `despotic.`**`emitterData`**(*emitName*, *emitterFile=None*, *emitterURL=None*, *extrap=True*, *noRefresh=False*)

Class to store the physical properties of a single emitting species, and preform computations on those properties. Note that all quantities are stored in cgs units.

Parameters

emitName [string] name of this species

emitterFile [string] name of LAMDA file containing data on this species; this option overrides the default

emitterURL [string] URL of LAMDA file containing data on this species; this option overrides the default

extrap [Boolean] if True, collision rate coefficients for this species will be extrapolated to temperatures outside the range given in the LAMDA tables

noRefresh [Boolean] if True, the routine will not attempt to automatically fetch updated versions of files from the web

Class attributes

name [string] name of emitting species

lamdaFile [string] name of file from which species was read

molWgt [float] molecular weight of species, in units of H masses

nlev [int] number of energy levels of the species

levEnergy [array, shape(nlev)] energies of levels

levTemp [array, shape(nlev)] energies of levels in K (i.e. levEnergy / kB)

levWgt [array shape(nlev)] degeneracies (statistical weights) of levels

nrad [int] number of radiative transition this species has

radUpper [integer array, shape (nrad)] array containing upper states for radiative transitions

radLower [integer array, shape (nrad)] array containing lower states for radiative transitions

radFreq [array, shape (nrad)] array of frequencies of radiative transitions

radTemp [array, shape (nrad)] same as radFreq, but multiplied by h/kB to give units of K

radTUpper [array, shape (nrad)] array of temperatures (E/kB) of upper radiative states

radA [array, shape (nrad)] array of Einstin A coefficients of radiative transitions

partners [dict] listing collision partners; keys are partner names, values are objects of class collPartner

EinsteinA [array, shape (nlev, nlev)] 2d array of nlev x nlev giving Einstein A's for radiative transitions connecting each level pair

freq [array, shape (nlev, nlev)] 2d array of nlev x nlev giving frequency of radiative transitions connecting each level pair

dT [array, shape (nlev, nlev)] 2d array of nlev x nlev giving energy difference between each level pair in K; it is positive for i > j, and negative for i < j

wgtRatio [array, shape (nlev, nlev)] 2d array giving ratio of statistical weights of each level pair

extrap [Boolean] if True, collision rate coefficients for this emitter are allowed to be extrapolated off the data table

`collRateMatrix`(*nH*, *comp*, *temp*)
This routine computes the matrix of collision rates (not rate coefficients) between every pair of levels.

Parameters

nH [float] number density of H nuclei

comp [class composition] bulk composition of the gas

temp [float] gas kinetic temperature

Returns

q [array, shape (nlev, nlev)] array in which element ij is the rate of collisional transitions from state i to state j, in s^-1

`partFunc`(*temp*)
Compute the partition function for this species at the given temperature.

Parameters

temp [float | array] gas kinetic temperature

Returns

Z [float | array] the partition function Z(T) for this species

`readLamda`(*fp*, *extrap=False*)
Read a LAMDA-formatted file

Parameters

fp [file] pointer to the start of a LAMDA-formatted file

extrap [Boolean] if True, collision rate coefficients for this species will be extrapolated to temperatures outside the range given in the LAMDA tables

Returns Nothing

8.1.8 `radiation`

class `despotic.`**`radiation`**
A class describing the radiation field affecting a cloud.

Parameters None

Class attributes

TCMB [float] the temperature of the CMB, in K

TradDust [float] the temperature of the dust IR background field, in K

ionRate [float] the primary ionization rate due to cosmic rays and x-rays, in s^-1 H^-1

chi [float] strength of the ISRF, normalized to the solar neighborhood value

fdDilute [float] dilution factor for the dust radiation field; a value of 0 means infinite dilution, so the dust does not contribute to the photon occupation number, while 1 means zero dilution, so the dust contributes as a full blackbody radiation field at temperature TradDust

`ngamma`(*Tnu*)
Return the photon occupation number from the CMB and dust radiation fields, equal to 1 / [exp(-h nu / k T_CMB) - 1] + fdDilute * 1 / [exp(-h nu / k T_radDust) - 1]

Parameters

Tnu [float | array] frequency translated into K, i.e. frequency times h/kB

Returns

ngamma [float | array] photon occupation number

8.1.9 `zonedcloud`

class `despotic.`**`zonedcloud`**(*fileName=None*, *colDen=None*, *AV=None*, *nZone=16*, *geometry='sphere'*, *verbose=False*)

A class consisting of an interstellar cloud divided into different column density / extinction zones.

Parameters

fileName [string] name of file from which to read cloud description

colDen [array] Array of column densities marking zone centers

AV [array] Array of visual extinction values (in mag) marking zone centers; AV is converted to column density using a V-band cross section equal to 0.4 * sigmaPE; this argument ignored if colDen is not None

nZone [int] Number of zones into which to divide the cloud, from 0 to the maximum column density found in the cloud description file fileName; ignored if colDen or AV is not None

geometry ['sphere' | 'slab'] geometry to assume for the cloud, either 'sphere' (onion-like) or 'slab' (layer cake-like)

verbose [Boolean] print out information about the cloud as we read it

`abundances`

Returns abundances of all emitting species, mass-weighted over cloud

`abundances_zone`

Return abundances of all emitting species in all zones

`addEmitter`(*emitName*, *emitAbundance*, *emitterFile=None*, *emitterURL=None*, *energySkip=False*, *extrap=True*)

Add an emitting species

Pamameters

emitName [string] name of the emitting species

emitAbundance [float or listlike] abundance of the emitting species relative to H; if this is listlike, it must have the same number of elements as the number of zones

emitterFile [string] name of LAMDA file containing data on this species; this option overrides the default

emitterURL [string] URL of LAMDA file containing data on this species; this option overrides the default

energySkip [Boolean] if set to True, this species is ignored when calculating line cooling rates

extrap [Boolean] if set to True, collision rate coefficients for this species will be extrapolated to temperatures outside the range given in the LAMDA table. If False, no extrapolation is perfomed, and providing temperatures outside the range in the table produces an error

Returns Nothing

`chemabundances`

Returns abundances of all species in the chemical network, mass-weighted over the zonedcloud

chemabundances_zone
Return abundances of all emitting species in all zones

dEdt (*c1Grav=0.0*, *thin=False*, *LTE=False*, *fixedLevPop=False*, *noClump=False*, *escapeProbGeom=None*, *PsiUser=None*, *sumOnly=False*, *dustOnly=False*, *gasOnly=False*, *dustCoolOnly=False*, *dampFactor=0.5*, *verbose=False*, *overrideSkip=False*, *zones=False*)
Return instantaneous values of heating / cooling terms

Parameters

c1Grav [float] if this is non-zero, the cloud is assumed to be collapsing, and energy is added at a rate Gamma_grav = c1 mu_H m_H cs^2 sqrt(4 pi G rho)

thin [Boolean] if set to True, cloud is assumed to be opticall thin

LTE [Boolean] if set to True, gas is assumed to be in LTE

fixedLevPop [Boolean] if set to True, level populations and escape probabilities are not recomputed, so the cooling rate is based on whatever values are stored

escapeProbGeom ['sphere' | 'slab' | 'LVG'] sets problem geometry that will be assumed in calculating escape probabilities; ignored if the escape probabilities are already initialized; if left as None, escapeProbGeom = self.geometry

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

dampFactor [float] damping factor to use in level population calculations; see emitter.setLevPopEscapeProb

PsiUser [callable] A user-specified function to add additional heating / cooling terms to the calculation. The function takes the cloud object as an argument, and must return a two-element array Psi, where Psi[0] = gas heating / cooling rate, Psi[1] = dust heating / cooling rate. Positive values indicate heating, negative values cooling, and units are assumed to be erg s^-1 H^-1.

sumOnly [Boolean] if true, rates contains only four entries: dEdtGas and dEdtDust give the heating / cooling rates for the gas and dust summed over all terms, and maxAbsdEdtGas and maxAbsdEdtDust give the largest of the absolute values of any of the contributing terms for dust and gas

gasOnly [Boolean] if true, the terms GammaISRF, GammaDustLine, LambdaDust, and PsiUserDust are omitted from rates. If both gasOnly and sumOnly are true, the dict contains only dEdtGas

dustOnly [Boolean] if true, the terms GammaPE, GammaCR, LambdaLine, GamamDLine, and PsiUserGas are omitted from rates. If both dustOnly and sumOnly are true, the dict contains only dEdtDust. Important caveat: the value of dEdtDust returned in this case will not exactly match that returned if dustOnly is false, because it will not contain the contribution from gas line cooling radiation that is absorbed by the dust

dustCoolOnly [Boolean] as dustOnly, but except that now only the terms LambdaDust, PsiGD, and PsiUserDust are computed

overrideSkip [Boolean] if True, energySkip directives are ignored, and cooling rates are calculated for all species

zones [Boolean] if True, heating and cooling rates are returned for each zone; if False, the values returned are mass-weighted over the entire cloud

Returns

rates [dict] A dict containing the values of the various heating and cooling rate terms; all quantities are in units of erg s^-1 H^-1, and by convention positive = heating, negative = cooling; for dust-gas exchange, positive indicates heating of gas, cooling of dust. By default these quantities are mass-weighted over the entire cloud, but if zones is True then they are returned as arrays for each zone

Elements of the dict are as follows by default, but can be altered by the additional keywords listed below:

GammaPE [float or array] photoelectric heating rate

GammaCR [float or array] cosmic ray heating rate

GammaGrav [float or array] gravitational contraction heating rate

LambdaLine [dict or array] cooling rate from lines; dictionary keys correspond to species in the emitter list, values give line cooling rate for that species

PsiGD [float or array] dust-gas energy exchange rate

GammaDustISRF [float or array] dust heating rate due to the ISRF

GammaDustCMB [float or array] dust heating rate due to CMB

GammaDustIR [float or array] dust heating rate due to IR field

GammaDustLine [float or array] dust heating rate due to absorption of line radiation

LambdaDust [float or array] dust cooling rate due to thermal emission

PsiUserGas [float or array] gas heating / cooling rate from user-specified function; only included if PsiUser != None

PsiUserDust [float or array] gas heating / cooling rate from user-specified function; only included is PsiUser != None

lineLum(*emitName*, *LTE=False*, *noClump=False*, *transition=None*, *thin=False*, *intOnly=False*, *TBOnly=False*, *lumOnly=False*, *escapeProbGeom=None*, *dampFactor=0.5*, *noRecompute=False*, *abstol=1e-08*, *verbose=False*)

Return the frequency-integrated intensity of various lines, summed over all zones

Parameters

emitName [string] name of the emitter for which the calculation is to be performed

LTE [Boolean] if True, and level populations are unitialized, they will be initialized to their LTE values; if they are initialized, this option is ignored

noClump [Boolean] if set to True, the clumping factor used in estimating rates for n^2 processes is set to unity

transition [list of two arrays] if left as None, luminosity is computed for all transitions; otherwise only selected transitions are computed, with transition[0] = array of upper states transition[1] = array of lower states

thin [Boolean] if True, the calculation is done assuming the cloud is optically thin; if level populations are uninitialized, and LTE is not set, they will be computed assuming the cloud is optically thin

intOnly: Boolean if true, the output is simply an array containing the frequency-integrated intensity of the specified lines; mutually exclusive with TBOnly and lumOnly

TBOnly: Boolean if True, the output is simply an array containing the velocity-integrated brightness temperatures of the specified lines; mutually exclusive with intOnly and lumOnly

lumOnly: Boolean if True, the output is simply an array containing the luminosity per H nucleus in each of the specified lines; mutually eclusive with intOnly and TBOonly

escapeProbGeom ['sphere' | 'slab' | 'LVG'] sets problem geometry that will be assumed in calculating escape probabilities; ignored if the escape probabilities are already initialized; if left as None, escapeProbGeom = self.geometry

dampFactor [float] damping factor to use in level population calculations; see emitter.setLevPopEscapeProb

noRecompute [False] if True, level populations and escape probabilities are not recomputed; instead, stored values are used

Returns res : list or array

if intOnly, TBOnly, and lumOnly are all False, each element of the list is a dict containing the following fields:

'freq' [float] frequency of the line in Hz

'upper' [int] index of upper state, with ground state = 0 and states ordered by energy

'lower' [int] index of lower state

'Tupper' [float] energy of the upper state in K (i.e. energy over kB)

'Tex' [float] excitation temperature relating the upper and lower levels

'intIntensity' [float] frequency-integrated intensity of the line, with the CMB contribution subtracted off; units are erg cm^-2 s^-1 sr^-1

'intTB' [float] velocity-integrated brightness temperature of the line, with the CMB contribution subtracted off; units are K km s^-1

'lumPerH' [float] luminosity of the line per H nucleus; units are erg s^-1 H^-1

'tau' [float] optical depth in the line, not including dust

'tauDust' [float] dust optical depth in the line

if intOnly, TBOnly, or lumOnly are True: res is an array containing the intIntensity, TB, or lumPerH fields of the dict described above

`mass`(*edge=False*)
Returns the mass in each zone.

Parameters

edge [Boolean] if True, the value returned gives the cumulative mass at each zone edge, starting from the outer edge; otherwise the value returned is the mass of each zone

Returns

mass [array] mass of each zone (if edge is False), or cumulative mass to each zone edge (if edge is True)

Remarks if the geometry is 'slab', the masses are undefined, and this return returns None

`radius`(*edge=False*)
Return the radius of each zone.

Parameters

edge [Boolean] if True, the value returned is the radii of the zone edges; otherwise it is the radii of the zone centers

Returns

rad [array] radii of zone centers (default) or edges (if edge is True)

Remarks if the geometry is 'slab', the values returned are the depths into the slab rather than the radii

setChemEq(***kwargs*)
Set the chemical abundances for a cloud to their equilibrium values, computed using a specified chemical network.

Parameters

kwargs [dict] these arguments are passed through to the corresponding function for each zone

Returns

success [Boolean] True if the calculation converges, False if it does not

Remarks if the key escapeProbGeom is not in kwargs, then escapeProbGeom will be set to self.geometry

setDustTempEq(***kwargs*)
Set Td to equilibrium dust temperature at fixed Tg

Parameters

kwargs [dict] these arguments are passed through to the corresponding function for each zone

Returns

success [Boolean] True if dust temperature calculation converged, False if not

setGasTempEq(***kwargs*)
Set Tg to equilibrium gas temperature at fixed Td

Parameters

kwargs [dict] these arguments are passed through to the corresponding function for each zone

Returns

success [Boolean] True if the calculation converges, False if it does not

Remarks if the key escapeProbGeom is not in kwargs, then escapeProbGeom will be set to self.geometry

setTempEq(***kwargs*)
Set Tg and Td to equilibrium gas and dust temperatures

Parameters

kwargs [dict] these arguments are passed through to the corresponding function for each zone

Returns

success [Boolean] True if the calculation converges, False if it does not

Remarks if the key escapeProbGeom is not in kwargs, then escapeProbGeom will be set to self.geometry

setVirial(*alphaVir=1.0*, *NTonly=False*)
This routine sets the velocity dispersion in all zones to the virial value

Parameters

alphaVir [float] virial ratio to be used in computation; defaults to 1

NTonly [Boolean] if True, the virial ratio is computed considering only the non-thermal component of the velocity dispersion

Returns Nothing

8.2 despotic functions

8.2.1 `fetchLamda`

`despotic.`**`fetchLamda`**(*inputURL*, *path=None*, *fileName=None*)

Routine to download LAMDA files from the web.

Parameters

inputURL [string] URL of LAMDA file containing data on this species; if this does not begin with "http://", indicating it is a URL, then this is assumed to be a filename within LAMDA, and a default URL is appended

path [string] relative or absolute path at which to store the file; if not set, the current directory is used; if the specified path does not exist, it is created

fileName [string] name to give to file; if not set, defaults to the same as the name in LAMDA

Returns

fname [string] local file name to which downloaded file was written; if URL cannot be opened, None is returned instead

8.2.2 `lineProfLTE`

`despotic.`**`lineProfLTE`**(*emdat*, *u*, *l*, *R*, *denProf*, *TProf*, *vProf=0.0*, *sigmaProf=0.0*, *offset=0.0*, *TCMB=2.73*, *vOut=None*, *vLim=None*, *nOut=100*, *dv=None*, *mxstep=10000*)

Return the brightness temperature versus velocity for a specified line, assuming the level populations are in LTE. The calculation is done along an infinitely thin pencil beam.

Parameters

em [class emitterData] emitterData object describing the emitting species for which the computation is to be made

u [int] upper state of line to be computed

l [int] lower state of line to be computed

R [float] cloud radius in cm

denProf [float or callable] If denProf is a float, this give the density in particles cm^-3 of the emitting species, which is taken to be uniform. denProf can also be a function giving the density as a function of radius; see remarks below for details.

TProf [float | callable] same as denProf, but giving the temperature in K

vProf [float | callable] same as vProf, but giving the bulk radial velocity in cm/s; if omitted, bulk velocity is set to 0

sigmaProf [float | callable] same as denProf, but giving the non-thermal velocity dispersion in cm/s; if omitted, non-thermal velocity dispersion is set to 0

offset [float] fractional distance from cloud center at which measurement is made; 0 = at cloud center, 1 = at cloud edge; valid values are 0 - 1

vOut [sequence] sequence of velocities (relative to line center at 0) at which the output is to be returned

vLim [sequence (2)] maximum and minimum velocities relative to line center at which to compute TB

nOut [int] number of velocities at which to output

dv [float, optional] velocity spacing at which to produce output

TCMB [float] CMB temperature used as a background to the cloud, in K. Defaults to 2.73.

mxstep [int] maximum number of steps in the ODE solver; default is 10,000

Returns

TB [array] brightness temperature as a function of velocity (in K)

vOut [array] velocities at which TB is computed (in cm s^-1)

Raises despoticError is the specified upper and lower state have no radiative transition between them, or if offset is not in the range 0 - 1

Remarks The functions denProf, TProf, vProf, and sigmaProf, if specified, should accept one floating argument, and return one floating value. The argument r is the radial position within the cloud in normalized units, so that the center is at r = 0 and the edge at r = 1. The return value should be the density, temperature, velocity, or non-thermal velocity dispesion at that position, in cgs units.

8.2.3 `refreshLamda`

`despotic.`**`refreshLamda`**(*path=None*, *cutoffDate=None*, *cutoffAge=None*, *LamdaURL=None*)

Refreshes LAMDA files by fetching new ones from the web

Parameters

path [string] path to the local LAMDA database; defaults to getting this information from the environment variable DESPOTIC_HOME

cutoffDate [class datetime.date or class datetime.datetime] a date or datetime specifying the age cutoff for updating files; files older than cutoffDate are updated, newer ones are not

cutoffAge [class datetime.timedelta] a duration between the present instant and the point in the past separating files that will be updated from files that will not be

LamdaURL [string] URL where LAMDA is located; defaults to the default value in fetchLamda

Returns Nothing

Remarks If the user sets both a cutoff age and a cutoff date, the date is used. If neither is set, the default cutoff age is 6 months.

8.3 despotic.chemistry classes

8.3.1 `abundanceDict`

class `despotic.chemistry.`**`abundanceDict`**(*specList*, *x*)

An abundanceDict object is a wrapper around a numpy array of abundances, and maps between human-readable chemical names (e.g. CO) and numeric indices in the array. Elements can be queried and addressed using a dict-like interface (e.g. if abd is an abundanceDict object, one could do abd['CO'] = 1.0e-4), but the underlying data structure can be manipulated with the speed and flexibility of a numpy array. In particular, one can perform arithmetic operations such as addition on abundance dicts, and they are simply applied to the underyling numpy array using the usual numpy operator rules.

This mapping between species names and array indices is created when the dict is first initialized, and is immutable thereafter. Thus operations that modify the keys in a dict are disallowed for abundanceDict objects.

Parameters

specList [list] list of species names for this abundanceDict; each list element must be a string

x [array of rank 1 or 2] array of abundances; the length of the first dimension of x must be equal to the length of specList

clear()
raises an error, since abundanceDicts are immutable

copy()
copy works just as for an ordinary dict

has_key(*key*)
has_key works just as for an ordinary dict

index(*spec*)

Parameters

spec [string | iterable] if this is a string, the method returns the index of that chemical species; if it is an iterable, the iterable must contain strings, and the method returns an array containing the indices of all species in the iterable

Returns

index [int | array] indices of the input species; if spec is a string, this is an int; otherwise it is an array of ints

Raises KeyError, if spec or any of its elements is not in the species list

keys()
keys works just as for an ordinary dict

pop(*key*)
raises an error, since abundanceDicts are immutable

popitem()
raises an error, since abundanceDicts are immutable

values()
values returns a list of numpy arrays corresponding to the rows of x

8.3.2 chemNetwork

class despotic.chemistry.**chemNetwork**(*cloud=None*, *info=None*)
This is a purely abstract class that defines the methods that all chemistry networks are required to implement. Chemistry networks should be derived from it, and should override its methods. Attempting to instantiate this directly will lead to an error.

Parameters

cloud [class cloud] a cloud object to which this network should be attached

info [dict] a dict of additional information to be passed to the network on instantiation

Class attributes

specList [list] list of strings giving the names of the species being treated in the chemical network

x [array] array of abundances of the species in specList

cloud [class cloud] a cloud object to which this chemical network is attached; can be None

abundances
The current abundances of every species in the chemical network, stored as an abundanceDict.

applyAbundances (*addEmitters=False*)
This method writes abundances from the chemical network back to the cloud to which this network is attached.

Parameters

addEmitters [bool] if True, and the network contains emitters that are not part of the parent cloud, then the network will attempt to add them using cloud.addEmitter. Otherwise this routine will change the abundances of whatever emitters are already attached to the cloud, but will not add new ones.

Returns: Nothing

dxdt (*xin*, *time*)
This routine returns the time rate of change of the abundances for all species in the network.

Parameters

xin [array] array of starting abundances

time [float] current time in sec

Returns

dxdt [array] the time derivative of all species abundances

8.3.3 `cr_reactions`

class despotic.chemistry.**cr_reactions** (*specList*, *reactions*, *sparse=False*)
The cr_reactions class is used to handle computation of cosmic ray-induced reaction rates. In addition to the constructor, the class has only a single method: dxdt, which returns the reaction rates.

Parameters

specList: listlike of string List of all chemical species in the full reaction network, including those that are derived from conservation laws instead of being computed directly

reactions [list of dict] A list listing all the reactions to be registered; each entry in the list must be a dict containing the keys 'spec' 'stoich', and 'rate', which list the species involved in the reaction, the stoichiometric factors for each species, and the reaction rate per primary CR ionization, respectively. Sign convention is that reactants on the left hand side have negative stoichiometric factors, those on the right hand side have positive factors.

sparse [bool] If True, the reaction rate matrix is represented as a sparse matrix; otherwise it is a dense matrix. This has no effect on results, but depending on the chemical network it may lead to improved execution speed and/or reduced memory usage.

Examples To list the reaction cr + H -> H+ + e-, the dict entry should be:

```
    { 'spec' : ['H', 'H+', 'e-'], 'stoich' : [-1, 1, 1],
      'rate' : 1.0 }
```

dxdt (*x*, *n*, *ionrate*)
This function returns the time derivative of the abundances x for a given cosmic ray ionization rate.

Parameters

x [array(N)] array of current species abundances

n [float] number density of H nuclei; only used if some reactions have multiple species on the LHS, otherwise can be set to any positive value

ionrate [float] cosmic ray primary ionization rate

Returns

dxdt: array(N) rate of change of all species abundances

8.3.4 `NL99`

class `despotic.chemistry.`**`NL99`**(*cloud=None*, *info=None*)

This class the implements the chemistry network of Nelson & Langer (1999, ApJ, 524, 923).

Parameters

cloud [class cloud] a DESPOTIC cloud object from which initial data are to be taken

info [dict] a dict containing additional parameters

Remarks The dict info may contain the following key - value pairs:

'xC' [float] the total C abundance per H nucleus; defaults to 2.0e-4

'xO' [float] the total H abundance per H nucleus; defaults to 4.0e-4

'xM' [float] the total refractory metal abundance per H nucleus; defaults to 2.0e-7

'sigmaDustV' [float] V band dust extinction cross section per H nucleus; if not set, the default behavior is to assume that sigmaDustV = 0.4 * cloud.dust.sigmaPE

'AV' [float] total visual extinction; ignored if sigmaDustV is set

'noClump' [bool] if True, the clumping factor is set to 1.0; defaults to False

`AV`

visual extinction in mag

`NH`

column density of H nuclei

`abundances`

abundances of all species in the chemical network

`applyAbundances`(*addEmitters=False*)

This method writes the abundances produced by the chemical network to the cloud's emitter list.

Parameters

addEmitters [Boolean] if True, emitters that are included in the chemical network but not in the cloud's existing emitter list will be added; if False, abundances of emitters already in the emitter list will be updated, but new emiters will not be added to the cloud

Returns Nothing

Remarks If there is no cloud associated with this chemical network, this routine does nothing and silently returns.

`cfac`

clumping factor; cannot be set directly, calculated from temp and sigmaNT

`chi`

ISRF strength, normalized to solar neighborhood value

`dxdt`(*xin*, *time*)

This method returns the time derivative of all abundances for this chemical network.

Parameters

xin [array(10)] current abundances of all species

time [float] current time; not actually used, but included as an argument for compatibility with odeint

Returns

dxdt [array(10)] time derivative of x

extendAbundances(*xin=None*)
Compute abundances of derived species not directly followed in the network.

Parameters

xin [array] abundances of species directly tracked in the network; if left as None, the abundances stored internally to the network are used

Returns

x [array] abundances, including those of derived species

ionRate
primary ionization rate from cosmic rays and x-rays

nH
volume density of H nuclei

sigmaNT
non-thermal velocity dispersion

temp
gas kinetic temperature

xHe
He abundance

8.3.5 NL99_GC

class despotic.chemistry.**NL99_GC**(*cloud=None*, *info=None*)
This class the implements the CO chemistry network of Nelson & Langer (1999, ApJ, 524, 923) coupled to the H2 chemistry network of Glover & MacLow (2007, ApJS, 169, 239), as combined by Glover & Clark (2012, MNRAS, 421, 9).

Parameters

cloud [class cloud] a DESPOTIC cloud object from which initial data are to be taken

info [dict] a dict containing additional parameters

Remarks The dict info may contain the following key - value pairs:

'xC' [float] the total C abundance per H nucleus; defaults to 2.0e-4

'xO' [float] the total H abundance per H nucleus; defaults to 4.0e-4

'xM' [float] the total refractory metal abundance per H nucleus; defaults to 2.0e-7

'Zd' [float] dust abundance in solar units; defaults to 1.0

'sigmaDustV' [float] V band dust extinction cross section per H nucleus; if not set, the default behavior is to assume that sigmaDustV = 0.4 * cloud.dust.sigmaPE

'AV' [float] total visual extinction; ignored if sigmaDustV is set

'noClump' [bool] if True, the clumping factor is set to 1.0; defaults to False

'sigmaNT' [float] non-thermal velocity dispersion

'temp' [float] gas kinetic temperature

AV
visual extinction in mag

NH
column density of H nuclei

Zd
dust abundance normalized to the solar neighborhood value

abundances
abundances of all species in the chemical network

applyAbundances (*addEmitters=False*)
This method writes the abundances produced by the chemical network to the cloud's emitter list.

Parameters

addEmitters [Boolean] if True, emitters that are included in the chemical network but not in the cloud's existing emitter list will be added; if False, abundances of emitters already in the emitter list will be updated, but new emiters will not be added to the cloud

Returns Nothing

Remarks If there is no cloud associated with this chemical network, this routine does nothing and silently returns.

cfac
clumping factor; cannot be set directly, calculated from temp and sigmaNT

chi
ISRF strength, normalized to solar neighborhood value

dxdt (*xin*, *time*)
This method returns the time derivative of all abundances for this chemical network.

Parameters

xin [array(12)] current abundances of all species

time [float] current time; not actually used, but included as an argument for compatibility with odeint

Returns

dxdt [array(12)] time derivative of x

extendAbundances (*xin=None*, *outdict=False*)
Compute abundances of derived species not directly followed in the network.

Parameters

xin [array] abundances of species directly tracked in the network; if left as None, the abundances stored internally to the network are used

outdict [bool] if True, the values are returned as an abundanceDict; if False, they are returned as a pure numpy array

Returns

x [array | abundanceDict] abundances, including those of derived species

ionRate
primary ionization rate from cosmic rays and x-rays

mu (*xin=None*)
Return mean particle mass in units of H mass

Parameters

xin [array] Chemical composition for which computation is to be done; if left as None, current chemical composition is used

Returns

mu [float] Mean mass per free particle, in units of H mass

nH
volume density of H nuclei

sigmaNT
non-thermal velocity dispersion

xHe
He abundance

8.3.6 `photoreactions`

class `despotic.chemistry.`**`photoreactions`**(*specList*, *reactions*, *sparse=False*)
The photoreactions class is used to handle computation of photon-induced reaction rates. Generally, it returns reaction rates for any reaction of the form gamma + ... -> ... with a rate that scales as the ISRF strength, parameterized in units of the Habing (1968) field, multiplied by dust and gas shielding factors. In addition to the constructor, the class has only a single method: dxdt, which returns the reaction rates.

Parameters

specList: listlike of string List of all chemical species in the full reaction network, including those that are derived from conservation laws instead of being computed directly

reactions [list of dict] A list listing all the reactions to be registered; each entry in the list must be a dict containing the keys:

- **'spec'** [list] list of strings giving the species involved in the reaction
- **'stoich'** [list] list of int stoichiometric factor for each species
- **'rate'** [float] reaction rate per target in a chi = 1 radiation field
- **'av_fac'** [float] optical depth per unit A_V
- **'shield_fac'** [(optional) callable] callable to compute the shielding factor; see the dxdt method for an explanation of how to specify its arguments

Reaction rates per target are given by chi * rate * shield_fac * exp(-av_fac * A_V)

sparse [bool] If True, the reaction rate matrix is represented as a sparse matrix; otherwise it is a dense matrix. This has no effect on results, but depending on the chemical network it may lead to improved execution speed and/or reduced memory usage.

8.3.7 `reaction_matrix`

class `despotic.chemistry.`**`reaction_matrix`**(*specList*, *reactions*, *sparse=False*)
This class provides a generic driver for computing rates of change of chemical species from chemical reactions. This class does the work of mapping from the reaction rates to rates of change in species abundances.

Parameters

specList: listlike of string List of all chemical species in the full reaction network, including those that are derived from conservation laws instead of being computed directly

reactions [list of dict] A list listing all the reactions to be registered; each entry in the list must be a dict containing the keys 'spec' and 'stoich', which list the species involved in the reaction and the stoichiometric factors for each species, respectively. Sign convention is that reactants on the left hand side have negative stoichiometric factors, those on the right hand side have positive factors.

sparse [bool] If True, the reaction rate matrix is represented as a sparse matrix; otherwise it is a dense matrix. This has no effect on results, but depending on the chemical network it may lead to improved execution speed and/or reduced memory usage.

Examples To describe the reaction C + O -> CO, the correct dict entry is:

```
{ 'spec' : ['C', 'O', 'CO'], 'stoich' : [-1, -1, 1] }
```

To describe the reaction H + H -> H2, the dict is:

```
{ 'spec' : ['H', 'H2'], 'stoich' : [-2, 1] }
```

`dxdt` (*x*, *n*, *ratecoef*)
This returns the rate of change of species abundances given a set of rate coefficients.

Parameters

x [array(N_species)] array of current species abundances

n [float] number density of H nuclei

ratecoef [array(N_reactions)] rate coefficients for each reaction; reaction rate per unit volume = ratecoef * product of densities of reactants; dxdt = reaction rate / unit volume / n

Returns

dxdt [array(N)] rate of change of all species abundances

8.4 despotic.chemistry functions

8.4.1 `chemEvol`

`despotic.chemistry.`**`chemEvol`**(*cloud*, *tFin*, *tInit=0.0*, *nOut=100*, *dt=None*, *tOut=None*, *network=None*, *info=None*, *addEmitters=False*, *evolveTemp='fixed'*, *isobaric=False*, *tempEqParam=None*, *dEdtParam=None*)
Evolve the abundances of a cloud using the specified chemical network.

Parameters

cloud [class cloud] cloud on which computation is to be performed

tFin [float] end time of integration, in sec

tInit [float] start time of integration, in sec

nOut [int] number of times at which to report the temperature; this is ignored if dt or tOut are set

dt [float] time interval between outputs, in sec; this is ignored if tOut is set

tOut [array] list of times at which to output the temperature, in s; must be sorted in increasing order

network [chemical network class] a valid chemical network class; this class must define the methods __init__, dxdt, and applyAbundances; if None, the existing chemical network for the cloud is used

info [dict] a dict of additional initialization information to be passed to the chemical network class when it is instantiated

addEmitters [Boolean] if True, emitters that are included in the chemical network but not in the cloud's existing emitter list will be added; if False, abundances of emitters already in the emitter list will be updated, but new emiters will not be added to the cloud

evolveTemp ['fixed' | 'gasEq' | 'fullEq' | 'evol'] how to treat the temperature evolution during the chemical evolution; 'fixed' = treat tempeature as fixed; 'gasEq' = hold dust temperature fixed, set gas temperature to instantaneous equilibrium value; 'fullEq' = set gas and dust temperatures to instantaneous equilibrium values; 'evol' = evolve gas temperature in time along with the chemistry, assuming the dust is always in instantaneous equilibrium

isobaric [Boolean] if set to True, the gas is assumed to be isobaric during the evolution (constant pressure); otherwise it is assumed to be isochoric; note that (since chemistry networks at present are not allowed to change the mean molecular weight), this option has no effect if evolveTemp is 'fixed'

tempEqParam [None | dict] if this is not None, then it must be a dict of values that will be passed as keyword arguments to the cloud.setTempEq, cloud.setGasTempEq, or cloud.setDustTempEq routines; only used if evolveTemp is not 'fixed'

dEdtParam [None | dict] if this is not None, then it must be a dict of values that will be passed as keyword arguments to the cloud.dEdt routine; only used if evolveTemp is 'evol'

Returns

time [array] array of output times, in sec

abundances [class abundanceDict] an abundanceDict giving the abundances as a function of time

Tg [array] gas temperature as a function of time; returned only if evolveTemp is not 'fixed'

Td [array] dust temperature as a function of time; returned only if evolveTemp is not 'fixed' or 'gasEq'

Raises despoticError, if network is None and the cloud does not already have a defined chemical network associated with it

8.4.2 `setChemEq`

`despotic.chemistry.`**`setChemEq`**(*cloud*, *tEqGuess=None*, *network=None*, *info=None*, *addEmitters=False*, *tol=1e-06*, *maxTime=1e+16*, *verbose=False*, *smallabd=1e-15*, *convList=None*, *evolveTemp='fixed'*, *isobaric=False*, *tempEqParam=None*, *dEdtParam=None*, *maxTempIter=50*)

Set the chemical abundances for a cloud to their equilibrium values, computed using a specified chemical netowrk.

Parameters

cloud [class cloud] cloud on which computation is to be performed

tEqGuess [float] a guess at the timescale over which equilibrium will be achieved; if left unspecified, the code will attempt to estimate this time scale on its own

network [chemNetwork class] a valid chemNetwork class; this class must define the methods __init__, dxdt, and applyAbundances; if None, the existing chemical network for the cloud is used

info [dict] a dict of additional initialization information to be passed to the chemical network class when it is instantiated

addEmitters [Boolean] if True, emitters that are included in the chemical network but not in the cloud's existing emitter list will be added; if False, abundances of emitters already in the emitter list will be updated, but new emiters will not be added to the cloud

evolveTemp ['fixed' | 'iterate' | 'iterateDust' | 'gasEq' | 'fullEq' | 'evol'] how to treat the temperature evolution during the chemical evolution:

- 'fixed' = treat tempeature as fixed
- 'iterate' = iterate between setting the gas temperature and chemistry to equilibrium
- 'iterateDust' = iterate between setting the gas and dust temperatures and the chemistry to equilibrium
- 'gasEq' = hold dust temperature fixed, set gas temperature to instantaneous equilibrium value as the chemistry evolves
- 'fullEq' = set gas and dust temperatures to instantaneous equilibrium values while evolving the chemistry network
- 'evol' = evolve gas temperature in time along with the chemistry, assuming the dust is always in instantaneous equilibrium

isobaric [Boolean] if set to True, the gas is assumed to be isobaric during the evolution (constant pressure); otherwise it is assumed to be isochoric; note that (since chemistry networks at present are not allowed to change the mean molecular weight), this option has no effect if evolveTemp is 'fixed'

tempEqParam [None | dict] if this is not None, then it must be a dict of values that will be passed as keyword arguments to the cloud.setTempEq, cloud.setGasTempEq, or cloud.setDustTempEq routines; only used if evolveTemp is not 'fixed'

dEdtParam [None | dict] if this is not None, then it must be a dict of values that will be passed as keyword arguments to the cloud.dEdt routine; only used if evolveTemp is 'evol'

tol [float] tolerance requirement on the equilibrium solution

convList [list] list of species to include when calculating tolerances to decide if network is converged; species not listed are not considered. If this is None, then all species are considered in deciding if the calculation is converged.

smallabd [float] abundances below smallabd are not considered when checking for convergence; set to 0 or a negative value to consider all abundances, but beware that this may result in false non-convergence due to roundoff error in very small abundances

maxTempIter [int] maximum number of iterations when iterating between chemistry and temperature; only used if evolveTemp is 'iterate' or 'iterateDust'

verbose [Boolean] if True, diagnostic information is printed as the calculation proceeds

Returns

converged [Boolean] True if the calculation converged, False if not

Raises despoticError, if network is None and the cloud does not already have a defined chemical network associated with it

Remarks The final abundances are written to the cloud whether or not the calculation converges.

8.4.3 Shielding functions

`despotic.chemistry.shielding.`**`fShield_H2_DB`**(*NH2*, *sigma*)

This function returns the shielding factor for H2 photodissociation as a function of the H2 column density and velocity dispersion, based on the approximation formula of Draine & Bertoldi (1996)

Parameters

NH2 [float | array] H2 column density in cm^-2

sigma [float | array] velocity dispersion in cm s^-1

Returns

fShield [float | array] the shielding factor for the input NH2 and sigma

`despotic.chemistry.shielding.`**`fShield_CO_vDB`**`(`*NCO*, *NH2*, *order=1*`)`

This function returns the shielding factor for CO photodissociation as a function of CO and H2 column densities, based on the model of van Dishoeck & Black (1987)

Parameters

NCO [float | array] CO column density in cm^-2

NH2 [float | array] H2 column density in cm^-2

order [1 | 2 | 3] order of spline interpolation on van Dishoeck & Black's table; 1 is the safest choice, but 2 and 3 are also provided, and may be more accurate in some ranges

Returns

fShield [array] the shielding factor for the input NCO and NH2 values

Raises ValueError if order is not 1, 2, or 3

CHAPTER NINE

ACKNOWLEDGEMENTS

DESPOTIC was primarily written by Mark Krumholz, but it contains or relies on the contributions of a number of others.

- Desika Narayanan has done more than anyone else to field test the code and discover bugs.
- DESPOTIC grew out of a code initially written in collaboration with Todd Thompson
- Adam Ginsburg provided useful advice and aided in early testing.
- Floris van der Tak has assisted in field testing the code, and helped improve the quality of the documentation.
- DESPOTIC would not be possible without the Leiden Atomic and Molecular Database, which is maintained by Floris van der Tak, Ewine van Dishoeck, and John Black, and which was initially established by Feredic Schöier

Mark Krumholz's work writing DESPOTIC was funded in part by the US National Science Foundation (grant AST-0955300) and NASA (grant NNX09AK31G). Continued development is supported by the Australian Research Council (grant DP160100695).

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search