User's Guide for DESPOTIC v. 1.1.0

Mark Krumholz

May 3, 2013

Contents

1 License and Citations

This is a guide for users of the DESPOTIC software package. DESPOTIC is distributed under the terms of the GNU General Public License v. 3. A copy of the license notification is included in all the source files.

If you use DESPOTIC in any published work, please cite the paper Krumholz, M. R., 2013, "DESPOTIC – A New Software Library to Derive the Energetics and SPectra of Optically Thick Interstellar Clouds", submitted to *Monthly Notices of the Royal Astronomical Society*, arXiv:1304.2404.

In addition, if you make use of any of DESPOTIC's capabilities that take advantage of data from the Leiden Atomic and Molecular Database, please cite Schöier, F. L., van der Tak, F. F. S., van Dishoeck, E. F., and Black, J. H., 2005, "An atomic and molecular database for analysis of submillimetre line observations", *Astronomy & Astrophysics*, 432, 369, and please cite the individual data sources for particular LAMDA files. These sources may be found at the LAMDA website.

2 Installation

This section describes several options for how to get DESPOTIC.

2.1 Downloading from Google Code

To download the full DESPOTIC distribution, just type

```
svn checkout http://despotic.googlecode.com/svn/trunk/ despotic-read-only
```

This will create a directory despotic, which will contain the subdirectories cloudfiles, despotic, dist, doc, and examples. The DESPOTIC package is in the despotic directory, while the other directories contain sample cloud files, a distribution that can be installed as a site package, this User's Guide, and some example Python scripts that use DESPOTIC.

At this point you can start using DESPOTIC immediately, but you might want to either set the environment variable $PYTHONPATH to the despotic subdirectory, or you might want to install DESPOTIC as a site package. If you do neither of these, you will only be able to run DESPOTIC when you are in the directory you just downloaded.

To install DESPOTIC as a site package, simply cd into the despotic subdirectory and type

```
python setup.py install
```

Depending on how your python installation is configured, you may need to prepend the command above with sudo.

2.2 Downloading from the Python Package Index

You can also download and the DESPOTIC package from the Python Package Index, at https://pypi.python.org/pypi/DESPOTIC/. This distribution includes only the DESPOTIC source, not any of the documentation or example files. The easiest way to install from PPI is using pip, the Python Installer Package. If you have pip installed, you can download DESPOTIC simply by typing

```
pip install despotic
```

Depending on how your python installation is configured, you may need to prepend the command above with sudo.

2.3 Specifying the Location of LAMDA

DESPOTIC uses the environment variable $DESPOTIC_HOME as a location where the locally-cached version of the Leiden Atomic and Molecular Database (LAMDA) should be located (see Section 7). It is recommended that you set this environment variable to the directory containing the DESPOTIC distribution you just downloaded. However, this is not required. If this environment variable is not set, DESPOTIC will just use the current working directory to store LAMDA files unless you tell it otherwise.

2.4 Requirements and Dependencies

`DESPOTIC` requires `sciPy >= 0.11.0`.

3 Quickstart Example

As with any Python program, DESPOTIC can be used either interactively or non-interactively. For the purposes of the Quickstart we will assume the commands are being issued in interactive mode, but they will work equally well non-interactively.

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
```

This command puts the cloud class into the current namespace.

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 DESPOTIC cloud file. The format for these files is described in Section 5. 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 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. In this case, the By default DESPOTIC operations are silent.

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.

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 dicts, 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 $J = 1 - 0$ 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 $J = 1 - 0$ line is 79 K km s^{-1}.

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']
```

4 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 Section 8. 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
```

4.1 Computing 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 following the directions in Section 5, 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 $n_{\rm H} = 10^2$ cm^{-3}, a column density of $N_{\rm H} = 10^{22}$ cm^{-2}, a non-thermal velocity dispersion of $\sigma_{\rm NT} = 2.0 \times 10^5$ cm s^{-1}, a gas temperature of $T_g = 10$ K, and a composition that is 0.1 ortho-H_2 molecules per H nucleus, 0.4 para-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.

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 `ilnes`, is a sequence of Python dicts, 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 erg cm^{-2} s^{-1} Hz^{-1}, `intTB` is the velocity-integrated brightness temperature (again subtracting off the CMB) in K km s^{-1}, and `lumPerH` is the rate of energy emission in the line per H nucleus in the cloud, in erg s^{-1}. This is a partial list of what the dict contains; see Section 8.1.5 for a full listing.

Once the data have 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]
plot(freq, TB, 'o')
```

4.2 Computing 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 containing the requisite information (Section 5), 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, all in CGS units, 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, all in erg s^{-1} H^{-1}. 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']
```

4.3 Computing 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.

4.4 Computing Time-Dependent Thermal 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:

```
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 Section 8.1.10 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 clouds, and one may interrogate their states (e.g. calculating their line emission) using the usual routines.

4.5 Computing Chemical Equilibria

As of version 1.1.0, `DESPOTIC` can also compute the chemical abundances in clouds using chemistry networks. There are a vast number of possible chemical networks, but at present `DESPOTIC` only includes the reduced carbon-oxygen chemistry network of Nelson & Langer (1999, *Astrophysical Journal*, 524, 923), which computes most of the important reactions involving carbon and oxygen. (`DESPOTIC` provides a generic chemical network class and drivers so that it is straightforward to expand to include other chemical networks; see Section 6.)

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 the next section) that specifies a different chemical network.

4.6 Computing Time-Dependent Chemical Evolution

`DESPOTIC` can also calculate time-dependent chemical evolution. This is accomplished through the method `cloud.chemEvol`. At with `cloud.tempEvol`, 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)
```

Note that the chemical network is not specified here, which is fine assuming that one has previously executed the `mycloud.setChemEq(network=NL99)` statement in the previous section, which assigns the NL99 network to the cloud. If one has not executed that statement, then the keyword `network=NL99` must be added to the call to `mycloud.chemEvol`.

The output quantity `abd` here is an object of class `abundanceDict`, which is a specialized dict for handling chemical abundances. 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

```
plot(t, abd['CO'])
plot(t, abd['C'])
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
```

4.7 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 co.

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*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), 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:

```
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 Section 8.9 for details.

4.8 Escape Probability Geometries

DESPOTIC currently supports three possible geometries that can be used when computing escape probabilities, and which are controlled by the `escapeProbGeom` optional argument to most DESPOTIC functions. This optional argument, if included, must be set equal to one of the three strings `'sphere'` (the default), `'slab'`, or `'LVG'`. In the examples provided in the previous sections, the methods `cloud.lineLum`, `cloud.dEdt`, `cloud.setTempEq`, `cloud.setGasTempEq`, `cloud.setDustTempEq`, and `cloud.tempEvol` all accept the optional argument `escapeProbGeom`.

5 DESPOTIC Cloud Files

5.1 Overall Structure

DESPOTIC cloud files contain descriptions of clouds that can be read by the cloud class, using either the class constructor or the read method; see Section 8.1 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. The full set of allowed keys and their functions is listed in Table 1. 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[1]. 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).

5.2 Emitters

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

```
emitter = name abundance [extrapolate] [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 Section 7. 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.

[1] In v1.0.4, `H2OPR` and `XH2` are implemented as python properties, and will automatically update `comp.xpH2` and `comp.xoH2` when changed interactively

Key	Units	Description
Physical properties		
`nH`	cm^{-3}	Volume density of H nuclei
`colDen`	cm^{-3}	Column density of H nuclei, averaged over cloud 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 coupling coefficient
`sigmaD10`	$\mathrm{cm}^2\ \mathrm{H}^{-1}$	Dust cross section per H to thermal radiation at 10 K
`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 the ISRF
`betaDust`	-	Dust spectral index in the infrared
`Zdust`	-	Dust abundance relative to Milky Way
Radiation field		
`TCMB`	K	Temperature of the cosmic microwave background
`TradDust`	K	Temperature of the dust-trapped IR radiation field
`ionRate`	$\mathrm{s}^{-1}\ \mathrm{H}^{-1}$	Primary ionization rate due to cosmic rays and x-rays
`chi`	-	ISRF strength, normalized to Solar neighborhood value
Chemical composition		
`emitter`	-	See Section 5.2

Table 1: List of allowed cloud file keys and their meanings.

Thus LAMDA refers to $^{28}Si^{16}O$ (the molecule composed of the most common isotopes) as sio, $^{29}Si^{16}O$ as 29sio, and $^{12}C^{18}O$ 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 Section 8.7.

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 `extrapolate` and `energySkip` change how `DESPOTIC` performs computations with that species. If `extrapolate` is omitted, `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 `extrapolate` is specified, `DESPOTIC` will instead extrapolate off the table by assuming that the downward collision rate coefficient varies as a powerlaw in temperature, with the powerlaw slope and normalization determined by the two closest table elements. The optional item `energySkip` specifies that a species should be ignored when computing heating and cooling rates via the `cloud.dEdt` method. 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`.

6 Chemistry and Chemical Networks

The chemistry capabilities in DESPOTIC, which are new as of version 1.1.0, are located in the despotic.chemistry sub-package. This package defines a generic class to describe chemical reaction networks, and two procedures that can operate on chemical networks and their associated clouds to compute chemical equilibria and time-dependent chemical evolution. These capabilities are add-ons to the capabilities of earlier versions of DESPOTIC, and versions 1.1.0 retains full backwards compatibility with earlier versions.

6.1 Operations on Chemical Networks: Time Evolution and Equilibria

In its most generic form, a chemical network is defined by a list of N chemical species, a set of abundances $\mathbf{x}$ for those species (defined relative to H nuclei, like all other abundances in DESPOTIC), and a function $f(\mathbf{x}, \mathbf{p})$ that gives the time rate of change of those abundances $d\mathbf{x}/dt = f(\mathbf{x}, \mathbf{p})$. The function f generally depends on the instantaneous abundances $\mathbf{x}$, and may also depend on any number of other parameters $\mathbf{p}$. Examples of quantities that might enter $\mathbf{p}$ include but are not limited to the ambient radiation field (for photochemical reactions), the cosmic ray ionization rate, abundances of species that are not explicitly included in the network, and the gas density and temperature. Given these definitions, 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 $d\mathbf{x}/dt = f(\mathbf{x}, \mathbf{p})$ from t_0 to t_1, where f is a known function that is defined by the chemical network. 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.

- Find an equilibrium set of abundances $\mathbf{x}_{\rm eq}$ such that $d\mathbf{x}/dt = f(\mathbf{x}_{\rm eq}, \mathbf{p}) = 0$. 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 f. DESPOTIC handles this problem by explicitly integrating the ODE $d\mathbf{x}/dt = f(\mathbf{x}, \mathbf{p})$ 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 class contains a wrapper around this routine, which allows it to be called to operate on a specific instance of cloud.

Full details of these modules are given in Sections 8.12 and 8.13.

6.2 The NL99 Network

As of version 1.1.0, `DESPOTIC` only ships with a a single-predefined chemistry network: 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 I, C^+, CH_x, CO, HCO^+, H_3^+, He^+, O I, OH_x, M, and M^+. Several of these are "super-species" that agglomerate several distinct species with similar reaction rates and pathways, including CH_x (where $x = 1 - 4$), OH_x (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 $CO + h\nu \rightarrow C + O$, 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 $N_H/2$, 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

$$A_V = 0.4\sigma_{PE}(N_H/2).$$

This ratio of V-band to 1000 Å 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 via

$$\begin{aligned} N_{H_2} &= x_{H_2} N_H/2 \\ N_{CO} &= x_{CO} N_H/2, \end{aligned}$$

which amounts to assuming that the CO and H_2 are uniformly distributed. Note that the NL99 network explicitly assumes $x_{H_2} = 0.5$, as no reactions involving atomic H are included. These column densities are then used to find a shielding factor by interpolating the tabulated values of van Dishoeck & Black (1998, *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 abundances 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) OH_x is divided evenly between OH and OH_2; (2) the abundances of the para- and ortho- states of a given species are equal; (3) the abundances ratios of all isotopomers of a species remain fixed as reactions occur, so, for example, the final abundance ratio of $C^{18}O$ to $C^{16}O$ as computed by the chemical network is always the same as the initial one.

6.3 Implementing New Chemical Networks via the `chemNetwork` Class

DESPOTIC implements chemical networks through the abstract base class `chemNetwork`, which is defined by module `despotic.chemistry.chemNetwork`. 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 `NL99` class for an example of how to derive a new chemical network class from `chemNetwork`.

The required non-callable attributes of a chemistry network class `cn` are as follows:

- `cn.specList` is a list of strings the describes the chemical species included in the network. The names in `specList` can be arbitrary, and are not used for any purpose other than providing human-readable labels on outputs.

- `cn.x` is 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` is 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.

The required callable attributes are

- `cn.__init__(self, cloud=None, info=None)` 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)` 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)` 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` instances 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.

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 these methods will automatically perform chemical calculations using the input network.

Finally, `chemNetwork` defines a property `abundances`, which 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. For example, to print the abundance of CO in a chemical network, and then set the abundance of C^+ to zero, one can simply type

```
print cn.abundances['CO']
cn.abundances['C+'] = 0.0
```

This save the user the trouble of remembering the numerical indices for CO and C^+ in a given network, and provides a means for external programs to access the abundances of particular species in a manner that is independent of the mapping between species name and index number, which might vary between two chemical networks that both contain the same species. Full details of the `abundanceDict` class are given in Section 8.17.

7 Atomic and Molecular Data

This section describes how DESPOTIC handles atomic and molecular data, both in terms of its local cache of LAMDA files and its internal representation of them.

7.1 The Local Database

DESPOTIC uses atomic and molecular data in the format specified by the Leiden Atomic and Molecular Database, described at http://home.strw.leidenuniv.nl/~moldata/. When DESPOTIC downloads a file from LAMDA, either by automatically guessing the name or if a URL is manually specified (see Section 5.2), it stores 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 Section 7.2.)

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.

7.2 Keeping the Database Up to Date

The data in LAMDA is 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 (Section 5.2) 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 Section 8.11.

7.3 How DESPOTIC Deals with Atomic and Molecular Data Internally

When it is running, DESPOTIC maintains a list of emitting species for which data have been read within the emitter module (Section 8.6). 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. First, since data on level structure, collision rates, etc. (everything stored in the `emitterData` class – Section 8.7) 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. Second, 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'. Finally, maintenance of a central emitter list affects how deepcopy and pickling operations operate on emitters. See Section 8.6 for details.

Name	Type	Meaning
nH	float	Number density of H nuclei
colDen	float	Column density of H nuclei
sigmaNT	float	Non-thermal velocity dispersion
dVdr	float	Radial velocity gradient
Tg	float	Gas temperature
Td	float	Dust temperature
comp	class composition	Bulk composition
dust	class dustprop	Dust properties
rad	class radiation	Radiation field properties
emitters	dict	emitters in the cloud; keys are emitter name, values are of class emitter

Table 2: Non-callable attributes of the cloud class.

8 Full Description of Despotic Modules

Below is a full listing of all the modules included in DESPOTIC. Except where noted otherwise, modules define a single class of the same name, and all the functions provided by that module are associated with that class. Note that all DESPOTIC classes and functions include a docstring, so their call signatures and return values can be read using the standard Python help utility.

8.1 cloud

The cloud module defines the class cloud. The non-callable attributes of this class are listed in Table 2. The callable attributes are discussed in the following sections.

8.1.1 cloud.__init__

Call signature:

```
cloud.__init__(self, fileName=None, verbose=False)
```

This initializes a cloud object. The optional argument fileName is a string that specifies the name of a despotic cloud file in the format specified in Section 5. The optional argument verbose is a Boolean that, if True, specifies that verbose output should be printed as a file is read. This procedure returns nothing.

8.1.2 cloud.read

Call signature:

```
cloud.read(self, fileName, verbose=False)
```

This reads a `despotic` cloud file in the format specified in Section 5 and uses its contents to set the properties of the calling `cloud` object. The argument `fileName` is a string that gives the name of the file to be read. The optional argument `verbose` is a Boolean that, if True, specifies that verbose output should be printed as a file is read. This procedure returns nothing.

8.1.3 `cloud.addEmitter`

Call signature:

```
cloud.addEmitter(self, emitName, emitAbundance, emitterFile=None, \
                       emitterURL=None, energySkip=False, \
                       extrap=False):
```

This routine adds an emitter to the `emitters` dict. The argument `emitName` is a string that becomes the key corresponding to this emitter in the `emitter` dict. The argument `emitAbundance` gives the abundance of the emitting species relative to H nuclei. The optional arguments `emitterFile` and `emitterURL` are strings specifying the name of a file or URL from which the LAMDA-formatted file containing data on the emitting species is to be read. These are treated exactly as specifications of file and URL in a `DESPOTIC` cloud file; see Section 5.2. Similarly, optional arguments `energySkip` and `extrap` are Booleans that are equivalent to the keys `extrapolation` and `energySkip` in a `DESPOTIC` cloud file. This procedure returns nothing.

8.1.4 `cloud.setVirial`

Call signature:

```
cloud.setVirial(self, alphaVir=1.0, setColDen=False, setnH=False)
```

This routine sets the non-thermal velocity dispersion, column density, or volume density to the value required to produce a specified virial ratio. By default, `colDen` and `nH` are left unchanged, and `sigmaNT` for the cloud is altered. The optional argument `alphaVir` is a float specifies the value of the virial ratio used in the computation, with a default value of 1.0. The optional arguments `setColDen` and `setnH` are Booleans that specify, respectively, that `sigmaNT` and `nH` should be fixed and `colDen` altered, or `sigmaNT` and `colDen` fixed and `nH` altered, so as to produce the desired virial ratio. This procedure returns nothing.

8.1.5 `cloud.lineLum`

Call signature:

```
lum = cloud.lineLum(self, emitName, LTE=False, noClump=False, \
                    transition=None, thin=False, intOnly=False, \
                    TBOnly=False, lumOnly=False, \
                    escapeProbGeom='sphere', \
                    noRecompute=False)
```

This routine returns the luminosities of all the lines for a specified emitter. The argument `emitName` is a string giving the name of the emitter whose lines are to be calculated. The optional arguments `LTE`, `thin`, and `noClump` are Booleans that, if True, specify that the level populations are to be set to their LTE values, that they are to be computed assuming that the cloud is optically thin, and that they are to be computed using a clumping factor of unity, respectively. The optional argument `noRecompute` is a Boolean that, if True, causes the level populations not to be recalculated at all; instead, stored values are used to compute the luminosities. The optional argument `escapeProbGeom` is a string specifying which geometry to assume when calculating escape probabilities (see Section 4.8). The optional argument `transition` is a sequence of two arrays, the first of which specifies a list of upper states and the second of which specifies a list of lower states. The routine will then compute only the lines corresponding to transitions between those two states. The default behavior is to perform the calculation for all transitions with non-zero Einstein A coefficients.

The return value `lum` is by default returns a sequence of dicts, but this behavior can be altered by the optional keywords `intOnly`, `TBOnly`, and `lumOnly`; these are all Booleans. If none of them are True, the routine returns a sequence of dicts, each corresponding to one line. Each dict contains the following entries: `freq` is the line frequency in Hz, `upper` is the index of the upper level, `lower` is the index of the lower level, `TUpper` is the energy of the upper state in K, `Tex` is the excitation temperature describing the relative populations of the upper and lower levels for this line, `tau` is the gas optical depth in the line, `tauDust` is the dust optical depth in the line, `lumPerH` is the total rate of energy emission per H nucleus in the line, in units of erg s^{-1} H^{-1}, `intIntensity` is the frequency-integrated intensity of the line with the CMB subtracted off, in erg cm^{-2} s^{-1} sr^{-1}, and `intTB` is the velocity-integrated brightness temperature of the line, in K km s^{-1}. If `intOnly`, `TBOnly`, or `lumOnly` are True, instead of returning a dict, the routine returns only an array corresponding to the entries `intIntensity`, `intTB`, or `lumPerH` in the default dict.

8.1.6 `cloud.dEdt`

Call signature:

```
rates = cloud.dEdt(self, 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)
```

This procedure returns the rates of heating and cooling for the cloud. When called, by default this routine computes the level populations of all emitting species not marked with `energySkip` using the `emitters.setLevPopEscapeProb` method (see Section 8.6.8).

By default the return value `rates` is a dict containing the following keys and values: `GammaPE` is the gas photoelectric heating rate; `GammaCR` is that cosmic ray / x-ray ionization heating rate; `GammaGrav` is the gas heating rate from adiabatic gravitational compression; `GammaDustISRF` is the dust heating rate from the interstellar radiation field; `GammaDustCMB` is the dust heating rate from the CMB; `GammaDustIR` is the dust heating rate from the background IR radiation field; `GammaDLine` is the dust heating rate due to absorption of line radiation; `PsiGD` is the gas-dust energy exchange rate, with a positive value indicating net transfer of energy from dust to gas; `LambdaDust` is the dust cooling rate due to thermal emission; finally, `LambdaLine` is a dict containing the cooling rate due to all emitting species, with the name of the species as the dict key and the cooling rate as its value. All quantities are given in units of erg s^{-1} H^{-1}.

The optional arguments have the following functions: `c1Grav` is a float specifying the constant C_1 used to compute the gravitational heating rate, with a default value of 0 indicating no gravitational heating; `LTE` is a Boolean that, if True, causes the level populations to be set to their LTE values; `thin` is a Boolean that, if True, causes the level populations to be computed assuming that the cloud is optically thin; `fixedLevPop` is a Boolean that, if True, causes the line heating and cooling rates to be computed using the currently-stored level populations for all emitters, rather than recomputing those level populations; `noClump` is a Boolean that, if True, causes the clumping factor to be set to 1.0 for purposes of computing the level populations; `escapeProbGeom` is a string specifying which geometry to assume when calculating escape probabilities (see Section 4.8); `PsiUser` is a callable giving a user-specified heating rate for the gas and dust (see below for details); `sumOnly` is a Boolean that, if True, specifies that only the sum of the heating and cooling terms should be returned, not the individual ones; `dustOnly` is a Boolean that, if True, specifies that only the dust heating and cooling terms should be computed; `gasOnly` is a Boolean that, if True, specifies that only the gas heating and cooling terms should be computed; `dampFactor` is a float that gives the damping factor used in the level population calculations; `overrideSkip` specifies that line heating and cooling rates should be computed even from species marked by `energySkip`; finally, `verbose` causes status messages to be printed as the computation proceeds.

A number of optional arguments can alter the contents of the dict that is returned. If `PsiUser` is not `None`, this will cause the keys `PsiUserGas` and `PsiUserDust` to be added, giving the heating / cooling rate produced by the user function. If `sumOnly` is True, then the returned dict contains only four entries, `dEdtGas`, `maxAbsdEdtGas`, `dEdtDust`, and `maxAbsdEdtDust`. These give, respectively, the net rate of heating / cooling for gas, the largest of the absolute values of any of the heating and cooling terms for gas, and the corresponding two properties for dust. If `gasOnly` is True, then `GammaISRF`, `GammaDLine`,

LambdaDust and PsiUserDust are omitted from the dict. If dustOnly is True, then GammaPE, GammaCR, LambdaLine, and GammaDLine are omitted from the dict. If dustCoolOnly is True, then GammaDustISRF, GammaDustCMB, and GammaDustIR are omitted as well. If both gasOnly and sumOnly are set, then the dict contains only dEdtGas and maxAbsdEdtGas, and similarly if both dustOnly or dustCoolOnly and sumOnly are set. Note that, if dustOnly or dustCoolOnly are set together with sumOnly, the summed heating and cooling rate for dust includes only those terms that would have been included in the dict has sumOnly not been set.

The optional argument PsiUser can be used to specify an additional heating and cooling function. This argument must be set equal to a callable, which takes one object of class cloud as an argument, and returns a two-element array. The first element is the gas heating / cooling rate (with positive indicating heating), and the second is the dust heating / cooling rate, both in erg s^{-1} H^{-1}. If this argument is not None when dEdt is called, then dEdt will call the specified callable, passing it the calling cloud object as its argument.

8.1.7 cloud.setTempEq

Call signature:

```
success = cloud.setTempEq(self, c1Grav=0.0, thin=False, noClump=False, \
                      LTE=False, Tinit=None, fixedLevPop=False, \
                      escapeProbGeom='sphere', PsiUser=None, \
                      verbose=False, tol=1e-4)
```

This procedure sets the cloud gas and dust temperature Tg and Td to their equilibrium values, such that the net rates of heating and cooling for both gas and dust are 0 to within a specified tolerance (see below). The optional arguments c1Grav, LTE, thin, fixedLevPop, escapeProbGeom, PsiUser, and verbose behave the same as for the routine dEdt (see Section 8.1.6). The optional argument Tinit is a sequence of two floats, which specifies initial guesses for the gas and dust temperatures to be used in the iterative solver. If it is left equal to None, then the currently-stored gas and dust temperatures are used, or 10 K if the currently-stored values are 0.

The tol argument specifies the tolerance within which the heating and cooling rates are to be set to zero. The residual for this solve is defined as the larger of dEdtGas / maxAbsdEdtGas and dEdtDust / maxAbsdEdtDust, where these are the values returned by the cloud.dEdt function (see Section 8.1.6); they represent the net heating / cooling rate of gas normalized to the largest of the absolute values of any of the individual terms contributing to heating and cooling, and vice versa. *Note that the numerators of these terms are evaluated at the current temperatures, but the denominator is evaluated at the initially-guessed temperatures.* The tolerance is defined in this way in order to guarantee that the residual is monotonically decreasing as one approaches the correct solution, which need not be the case if both the numerator and denominator are evaluated at the current temperature. The temperature calculation is considered converged when this residual falls below tol.

This procedure returns True if the iteration converges, and False if it fails to converge.

8.1.8 `cloud.setDustTempEq`

Call signature:

```
success = cloud.setDustTempEq(self, PsiUser=None, Tdinit=None, \
                              noLines=False, noClump=False, \
                              verbose=False)
```

This function is identical to `cloud.setTempEq` (Section 8.1.7) except that the gas temperature `Tg` is held fixed and only the dust temperature `Td` is altered. The optional argument `Tdinit` is a float giving an initial guess for the dust temperature. The optional argument `noLines` specifies that the line heating of the dust should be ignored in this computation; this can be advantageous because in many situations line heating of the dust is negligible, and omitting line heating makes the calculation run much faster. All other optional arguments, and the return value, are the same as for `cloud.setTempEq`.

This procedure returns nothing.

8.1.9 `cloud.setGasTempEq`

Call signature:

```
success = cloud.setGasTempEq(self, c1Grav=0.0, thin=False, noClump=False, \
                             LTE=False, Tginit=None, fixedLevPop=False, \
                             escapeProbGeom='sphere', PsiUser=None, verbose=False)
```

This function is identical to `cloud.setTempEq` (Section 8.1.7) except that the dust temperature `Td` is held fixed and only the gas temperature `Tg` is altered. The optional argument `Tginit` is a float giving an initial guess for the dust temperature. All other optional arguments, and the return value, are the same as for `cloud.setTempEq`.

8.1.10 `cloud.tempEvol`

Call signature:

```
out, time = cloud.tempEvol(self, 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, verbose=False)
```

This function calculates the evolution of the gas temperature of the cloud over a specified time period, assuming that the dust temperature and level populations reach equilibrium instantaneously. The argument `tFin` is a float giving the time, in seconds, at which to end the integration. The optional argument `tInit` specifies the time at which integration

starts; the gas temperature `cloud.Tg` initially stored in the cloud is taken to be at this initial time. The optional argument `isobaric` is a Boolean that, if True, specifies that cooling is to be computed isobarically; the default is isochoric.

The optional arguments `nOut`, `dt`, and `tOut` control the times at which data are output; `nOut` is an integer that gives the number of equally-spaced time intervals between `tFin` and `tInit` at which to output. Alternately, `dt` is float that gives the time interval (in seconds) between output times, or `tOut` is a sequence specifying exact output times in seconds. If `tOut` is specified, it must be sorted in increasing order. The optional arguments `c1Grav`, `noClump`, `thin`, `LTE`, `fixedLevPop`, `escapeProbGeom`, `PsiUser`, and `verbose` have the same meaning as for the routine `cloud.dEdt` (see Section 8.1.6).

This function returns two sequences. The second of these, `time` is always a sequence of floats, and it contains the list of output times. The first sequence, `out` is by default a sequence of floats containing the calculated gas temperatures in K at the times given in `time`. If the optional argument `fullOutput` is True, then `out` is instead a sequence of cloud objects, each of which is a deep copy of the full state of the cloud at the specified output time.

Name	Type	Meaning
xHI	float	H I abundance per H nucleus
xoH2	float	ortho-H_2 abundance per H nucleus
xpH2	float	para-H_2 abundance per H nucleus
xHe	float	He abundance per H nucleus
xe	float	free electron abundance per H nucleus
xHplus	float	H^+ abundance per H nucleus
mu	float	mean mass per free particle, in units of m_H
mhH	float	mean mass per H nucleus, in units of m_H
qIon	float	energy added to the gas per primary ionization
cv	float	specific heat per H nucleus at constant volume

Table 3: Non-callable attributes of the composition class.

8.2 composition

The composition module defines the composition class, which stores information and methods related to the bulk chemical composition of gas. The non-callable attributes of this class are listed in Table 3, and the callable attributes are described in the following sections.

8.2.1 composition.computeDerived

Call signature:

```
composition.computeDerived(self, nH)
```

This routine set the quantities composition.mu, composition.muH, and composition.qIon based on the stored chemical composition, and the input volume density. This is specified by the argument nH, which must be a float. This procedure returns nothing.

8.2.2 composition.computeCv

Call signature:

```
cv = computeCv(self, T, noSet=False, Jmax=40)
```

This routine computes the specific heat per H nucleus $c_{v,\mathrm{H}}$ from the stored gas composition and the input temperature. The argument T is a float or an array of floats that specifies the temperature. The optional argument noSet is a Boolean. If it is False, the computed value of $c_{v,\mathrm{H}}$ is stored as composition.cv; if it is True, the computed value is returned by not stored. The optional argument Jmax specifies the maximum rotational quantum number to consider when computing the specific heat. Accurate results require that $T \ll J(J+1)\theta_{\mathrm{rot}}$, where $\theta_{\mathrm{rot}} = 85.3$ K is the rotational constant of H_2 in K. This procedure returns the computed value of $c_{v,\mathrm{H}}$. If T is a float, the returned value is a float as well. If T is an array, the returned value is an array of the same shape.

Name	Type	Meaning
`sigma10`	float	Dust opacity per H nucleus to 10 K thermal radiation, in cm^{-2} H^{-1}
`sigmaPE`	float	Dust opacity per H nucleus to $8 - 13.6$ eV photons, in cm^{-2} H^{-1}
`sigmaISRF`	float	Dust opacity per H nucleus to ISRF photons that heat dust grains, in cm^{-2} H^{-1}
`Zd`	float	Dust abundance normalized to the Milky Way value
`beta`	float	Dust spectral index in the mm, $\sigma \propto \nu^{\beta}$
`alphaGD`	float	Grain-gas collisional coupling coefficient, in cm^3 s^{-1} $\mathrm{K}^{-3/2}$

Table 4: Non-callable attributes of the `dustProp` class.

8.3 dustProp

The `dustProb` module defines the `dustProp` class, which stores properties of the dust. The non-callable attributes in the class are described in Table 4. This class has no callable attributes.

Name	Type	Meaning
`TCMB`	float	CMB temperature
`TradDust`	float	Temperature of the dust thermal radiation field
`ionRate`	float	Primary cosmic-ray / X-ray ionization rate, in s^{-1} per H nucleus
`chi`	float	ISRF strength, normalized to the Solar neighborhood value

Table 5: Non-callable attributes of the `radiation` class.

8.4 radiation

The `radiation` module defines the `radiation` class, which describes the radiation field around a cloud. The non-callable attributes of the class are listed in Table 5, and the callable attributes are listed below.

8.5 `radiation.__init__`

Call signature:

```
radiation.__init__(self)
```

This procedure creates a `radiation` object with its parameters initialized to reasonable Milky Way defaults: CMB temperature of 2.73 K, dust radiation field temperature of 0, $\chi = 1$, and primary ionization rate 2×10^{-17} s^{-1} H^{-1}.

8.5.1 `radiation.ngamma`

Call signature:

```
ngamma = radiation.ngamma(self, Tnu)
```

This routine computes the photon occupation number of the CMB, $1/[\exp(-h\nu/k_B T_{\rm CMB}]$. The argument `Tnu` is a float or array of floats, which specifies the value of $h\nu/k_B$ at which the computation is to be performed. The return value is a float or array of floats with the same shape as `Tnu`.

Name	Type	Meaning
`name`	string	Name of emitting species
`abundance`	float	Abundance relative to H nuclei
`data`	class `emitterData`	Physical data on this species
`levPop`	array(nlev) of float	Fractional populations of all levels
`levPopInitialized`	Boolean	Are level populations initialized?
`escapeProb`	array(nlev, nlev) of float	Escape probability for radiative transitions between each level pair
`escapeProbInitialized`	Boolean	Are escape probabilities initialized?
`energySkip`	Boolean	Should this emitter be skipped when computing heating / cooling rates?

Table 6: Non-callable attributes of the `emitter` class.

8.6 emitter

The `emitter` module defines the class `emitter`, and also stores a dict `knownEmitterData`. This dict contains an entry for every `emitter` class object for which a corresponding `emitterData` object (see Section 8.7) has been created. The keys are the names of the `emitterData` objects, and the values are the corresponding `emitterData` objects. This list is maintained so that `DESPOTIC` need not have a duplicate copy of the `emitterData` for every instance of an emitter, and so that the LAMDA file need not be re-read every time an instance of a given emitting species is instantiated. Instead, different `emitter` objects just contain shallow copies of a single master copy on the `knownEmitterData` dict.

The `emitter` class stores information about emitting species, and provides methods for performing computations on those species. It is distinct from the `emitterData` class in that `emitterData` stores only physical constants (e.g. energies of quantum levels and Einstein coefficients for radiative transitions between them) that do not depend on the physical conditions in a given cloud. In contrast, the `emitter` class stores information that does vary from cloud to cloud. The non-callable attributes of the class are described in Table 6, and the callable attributes are described in the following sections.

8.6.1 `emitter.__init__`

Call signature:

```
emitter.__init__(self, emitName, emitAbundance, extrap=False, \
                 energySkip=False, emitterFile=None, emitterURL=None)
```

This method initializes an object of type `emitter`. The first argument `emitName` is a string that gives the name of the emitting species, which will be stored as `emitter.name`. The second argument, `emitAbundance`, is a float that gives the abundance of the species relative to H nuclei; this will be stored as `emitter.abundance`. The optional argument `extrap` is a Boolean that specifies whether collision rates for this emitter can be extrapolated past the range provided in the LAMDA file. The optional argument `energySkip` is a Boolean that specifies that, if True, specifies that this species is unimportant for heating or cooling, and thus can be skipped when computing heating and cooling rates. The optional arguments `emitterFile` and `emitterURL` are identical to the arguments of the same name in `cloud.addEmitter` (see Section 8.1.3).

This routine returns nothing.

The behavior of this routine with respect to the dict `knownEmitterData` stored in the `emitter` module is as follows. When `emitter.__init__` is called, before attempting to read data from a LAMDA file, it first checks if the input `emitName` matches any of the keys in `knownEmitterData`. If a match is found, the `data` attribute of this instance of `emitter` is set equal to a shallow copy of the corresponding `emitterData` in `knownEmitterData`, and no files are read. If no match is found, on the other hand, this routine attempts to read the data for this species from a LAMDA file. If successful, the routine creates an `emitterData` instances from that LAMDA file, stores it in `knownEmitterData` with a key equal to the input `emitName`, and then sets `emitter.data` equal to a shallow copy of that `emitterData`.

8.6.2 `emitter.__deepcopy__`

Call signature:

```
emitter.__deepcopy__(self, memo={})
```

This method provides a custom deep copy operation for the `emitter` class. A custom deep copy is required because of the way emitter data is handled. Since the `data` attribute of this class should always be a shallow copy of an element in the `knownEmitterData` master list, we want to ensure that the `data` attribute is never deep-copied. This method implements that operation, such that a deep copy of an `emitter` will contain a deep copy of all its internal data except `data`, which will continue to be a shallow copy of an element in `knownEmitterData`. The optional argument `memo` is the standard dict used by the built-in `copy` module.

This procedure returns nothing.

8.6.3 `emitter.__getstate__` and `emitter.__setstate__`

Call signatures:

```
odict = emitter.__getstate__(self)
```

and

```
emitter.__setstate__(self, idict)
```

This pair of routines provides the functionality necessary for pickling of `emitter` objects. Custom behavior is required for two reasons. First, the `data` attribute is of class `emitterData`, and `emitterData` objects in turn contain one or more instances of the `collPartner` class. The attribute `collPartner.colRateInterp` is a sequence of callables, and thus cannot be pickled. Second, as with the deep copy operation, we wish to ensure that pickling respects the paradigm that physical data on emitting species is carried in a master dict `knownEmitterData`, all instances of the same emitting species only contain shallow copies of elements of that list.

To implement this functionality, the `emitter.__getstate__` routine returns a dict containing all of the attributes of the `emitter` except `emitter.data`, and with two added entries that store `emitter.data.extrap` and `emitter.data.lamdaFile`. The `emitter.__setstate__` routine takes as an argument a dict (presumably read from a pickle file) containing these two extra attributes. It uses them to re-read the LAMDA file, re-calculate the interpolating functions stored in the instances of `collPartner.colRateInterp`, and re-set the value of `emitter.data.extrap`. It then deletes the two extraneous entries from the dict of the `emitter`.

8.6.4 `emitter.setLevPopLTE`

Call signature:

```
emitter.setLevPopLTE(self, temp)
```

This procedure sets `emitter.levPop` to the values expected for a species in local thermodynamic equilibrium at temperature `temp`. The argument `temp` is a float. The procedure returns nothing.

8.6.5 `emitter.setThin`

Call signature:

```
emitter.setThin(self)
```

This procedure sets all escape probabilities in the array `emitter.escapeProb` to 1.0. It turns nothing.

8.6.6 `emitter.setLevPop`

Call signature:

```
infoDict = emitter.setLevPop(self, thisCloud, thin=False, noClump=False, \
                      diagOnly=False, verbose=False)
```

This procedure calculates the level populations for this emitter in statistical but not necessarily thermodynamic equilibrium, using the stored escape probabilities, and stores them in `emitter.levPop`. The argument `thisCloud` is the `cloud` object for which this computation is to be performed. The optional argument `thin` is a Boolean that, if True, specifies that the escape probabilities for all transitions are to be assumed to be unity for the purposes of this calculation; if it is False, the current value of `emitter.escapeProb` is used instead. The optional argument `noClump`, if True, specifies that the clumping factor should be set to unity; if False, the clumping factor is computed normally from the cloud velocity dispersion and sound speed. The optional argument `verbose`, if True, causes status messages to be printed as the calculation is performed. Finally, the argument `diagOnly` causes the level populations not to be altered, but all the diagnostic information to be computed normally.

In addition to setting the value of `emitter.levPop` and `emitter.levPopInitialized`, this routine returns a dict containing diagnostic information for the computation. The dict contains the following entries: `qNoClump` is a matrix of collisional transition rates between states before a clumping factor is applied, with element ij giving the collision rate for transitions from state i to state j; `ngamma` is a matrix of photon occupation numbers from the CMB, with element ij giving the CMB photon occupation number at the frequency corresponding to transitions from state i to state j; `inRateCoef` is a matrix giving the total rate coefficient (in s^{-1}) for transitions into state i from state j, including both radiative and collisional processes; `m` is the matrix that multiplies the vector of level populations in the statistical equilibrium equations (equation 58 of the `DESPOTIC` paper); `levDel` is a list of levels that have been deleted from the computation to reduce the matrix condition number (see Appendix C of the `DESPOTIC` paper), and is included in the dict only if level reduction is required; `inRateCoefRed` and `mRed` are the values of `inRateCoef` and `m` after level reduction.

8.6.7 `emitter.setEscapeProb`

Call signature:

```
emitter.setEscapeProb(self, thisCloud, transition=None, \
                          escapeProbGeom='sphere')
```

This procedure calculates escape probabilities from the currently-stored level populations and stores them in `emitter.escapeProb`. The argument `thisCloud` is a `cloud` object for which this computation is to be performed. The optional argument `transition` may be `None`, or it may be a sequence of two sequences of int. If it is `None`, then escape probabilities are set for all transitions. If it is a sequence of two sequences, then transition[0] is interpreted as a sequence of upper states, and transition[1] as a sequence of lower states; the escape probability is set only for transitions between the specified states. The optional argument `escapeProbGeom` specifies the geometry to be assumed in computing escape probabilities (see Section 4.8). This routine returns nothing.

8.6.8 emitter.setLevPopEscapeProb

Call signature:

```
success = emitter.setLevPopEscapeProb(self, thisCloud, \
                                      escapeProbGeom='sphere', \
                                      noClump=False, verbose=False, \
                                      reltol=1e-6, abstol=1e-8, \
                                      maxiter=200, veryverbose=False, \
                                      dampFactor=0.5)
```

This procedure simultaneously solves for `emitter.levPop` and `emitter.escapeProb` using an iterative procedure, as outlined in the `DESPOTIC` paper. The argument `thisCloud` is the `cloud` object for which this computation is to be performed. The optional argument `escapeProbGeom` gives the geometry to be assumed in the escape probability computation (see Section 4.8). The optional argument `noClump` is a Boolean that, if True, sets the clumping factor to unity. The optional arguments `verbose` and `veryverbose` cause some and a great deal, respectively, of diagnostic information to be printed as the computation proceeds. The optional argument `dampFactor` is a float that gives the damping factor used when iterating. The optional arguments `abstol` and `reltol` give the absolute and relative tolerances used to define the criteria for convergence, as described in the `DESPOTIC` paper. The calculation is considered converged when either the relative or absolute error falls below the specified tolerance. Finally, the optional argument `maxiter` is an int that specifies the maximum number of iterations to perform. The routine returns `True` if either the absolute or relative error falls below the input tolerance before the maximum number of iterations is reached, and False otherwise.

8.6.9 emitter.opticalDepth

Call signature:

```
tau = opticalDepth(self, transition=None, escapeProbGeom='sphere')
```

This procedure returns the optical depth corresponding to the escape probabilities `emitter.escapeProb`. The optional argument `transition` may be `None`, or it may be a sequence of two sequences of int. If it is `None`, then optical depths are computed for all transitions in the list of transitions in `emitter.data`. If it is a sequence of two sequences, then transition[0] is interpreted as a sequence of upper states, and transition[1] as a sequence of lower states; the optical depth is computed only for transitions between the specified states. The optional argument `escapeProbGeom` gives the geometry to be assumed in the escape probability computation (see Section 4.8). This procedure returns an array giving the optical depths of the specified transitions.

8.6.10 `emitter.luminosityPerH`

Call signature:

```
lum = emitter.luminosityPerH(self, rad, transition=None, total=False, \
                             thin=False)
```

This procedure returns the luminosity per H nucleus, in erg s^{-1}, that is emitted in lines from this species. These quantities are computed from `emitter.levPop` and `emitter.escapeProb`. The argument `rad` is an object of class radiation that describes the radiation field impinging on the cloud. The optional argument `thin` is a Boolean; if True, the escape probability is assumed to be unity for all transitions when performing this computation. The optional argument `transition` may be `None`, or it may be a sequence of two sequences of int. If it is `None`, then luminosities are computed for all transitions in the list of transitions in `emitter.data`. If it is a sequence of two sequences, then transition[0] is interpreted as a sequence of upper states, and transition[1] as a sequence of lower states; the luminosity is computed only for transitions between the specified states.

By default, the routine returns an array giving the luminosity per H nucleus in each line. The optional argument `total` is a Boolean, and, if True, the routine instead returns a single float that is equal to the sum of the luminosities of all transitions.

8.6.11 `emitter.setExtrap`

Call signature:

```
emitter.setExtrap(self, extrap)
```

This procedure changes whether extrapolation is allowed for this species; the value of `emitter.data.extrap` is set equal to the value of the argument `extrap`. This procedure returns nothing.

8.7 emitterData

The `emitterData` module defines the `emitterData` class, which stores physical data on emitting species. One instance of the `emitterData` class is created each time a distinct emitting species is created. The non-callable attributes of `emitterData` objects are listed in Table 7. The callable attributes are listed below.

8.7.1 `emitterData.__init__`

Call signature:

```
emitterData.__init__(self, emitName, emitterFile=None, emitterURL=None, \
                     extrap=False, noRefresh=False)
```

This routine initializes `emitterData` objects. The optional arguments `emitterFile`, `emitterURL`, and `extrap` are identical to the arguments of the same name for the `cloud.addEmitter` routine (see Section 8.1.3). The `noRefresh` optional argument is a Boolean. If it is False, before reading data from a file stored locally, the routine will check the age of the file. If it is older than six months, the routine will attempt to download a new copy of the data from LAMDA. If `noRefresh` is True, this behavior is suppressed and no new data will be downloaded regardless of the age of the file.

This routine returns nothing.

8.7.2 `emitterData.collRateMatrix`

Call signature:

```
rate = emitterData.collRateMatrix(self, nH, comp, temp)
```

This routine returns the rate of collisional transitions between every pair of levels. The argument `nH` is the volume density of H nuclei in cm^{-3}. The argument `comp` is an object of class composition (see Section 8.2) that describes the gas bulk chemical composition. The argument `temp` is a float that gives the gas kinetic temperature. The routine returns an array `rate` of shape (`nlev`, `nlev`) in which element ij gives the rate of collisional transitions from state i to state j in units of s^{-1}, not including any enhancement due to clumping.

8.7.3 `emitterData.partFunc`

Call signature:

```
Z = emitterData.partFunc(self, temp)
```

This routine computes the partition function $Z(T)$ for the species. The argument `temp` is a temperature, can be either a float or an array of floats. The return value `Z` gives the partition function, and is either a float or an array of floats of the same shape as the input temperature array.

Name	Type	Meaning
`name`	string	Name of emitting species
`lamdaFile`	string	Name of the file from which the data for this species was read
`molWgt`	float	Molecular weight of species in units of $m_{\rm H}$
`nlev`	int	Number of energy levels for this species
`levEnergy`	array(nlev) of float	Energies of levels (in erg)
`levTemp`	array(nlev) of float	Energies of levels (in K)
`nrad`	int	Number of radiative transitions for this species
`radUpper`	array(nrad) of int	Upper levels of radiative transitions
`radLower`	array(nrad) of int	Lower levels of radiative transitions
`radFreq`	array(nrad) of float	Frequencies of radiative transitions (in Hz)
`radTemp`	array(nrad) of float	Energy differences of radiative transitions (in K)
`radTUpper`	array(nrad) of float	Energies of radiative transition upper levels (in K)
`radA`	array(nrad) of float	Einstein A coefficients of radiative transitions
`partners`	dict	Dict of collision partners; keys are partner names, values are of class `collPartner`
`EinsteinA`	array(nlev, nlev) of float	Matrix of Einstein A values connecting every pair of states
`freq`	array(nlev, nlev) of float	Matrix of frequencies for transitions connecting every pair of states
`dT`	array(nlev, nlev) of float	Matrix of energy differences between every pair of states (in K)
`wgtRatio`	array(nlev, nlev) of float	Matrix of ratios of statistical weights for every pair of states
`extrap`	Boolean	Can collision rates for this species be extrapolated?

Table 7: Non-callable attributes of the `emitterData` class.

Name	Type	Meaning
`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) of float	Array of temperatures at which collision rate coefficients are tabulated
`colUpper`	array(ntrans) of int	Array of upper states for collisional transitions
`colLower`	array(ntrans) of int	Array of lower states for collisional transitions
`colRate`	array(ntrans, ntemp) of float	Table of downward transition rate coefficients
`colRateInterp`	sequence(ntrans) of functions	Functions giving the rate coefficient for each transition interpolated in temperature

Table 8: Non-callable attributes of the `collPartner` class.

8.8 collPartner

The `collPartner` module defines the `collPartner` class, a class that stores information about collision partners for emitting species. Instances of `collPartner` are created when instances of `emitterData` are created, and the corresponding LAMDA files are read. The non-callable attributes of a `collPartner` object are listed in Table 8. The callable attributes of the class are listed in the following sections.

8.8.1 `collPartner.__init__`

Call signature:

```
collPartner.__init__(self, fp, nlev, extrap=False)
```

This method initializes a `collPartner` object. The argument `fp` is a file object, and it must point to the section of a LAMDA-formatted file at which the information about a particular collision partner beings. The argument `nlev` is an integer that gives the number of levels in the emitter for which this is a collision partner. The optional argument `extrap` is a Boolean that specifies whether extrapolation is allowed for this emitter. If `False`, no extrapolation in temperature outside the range found in the LAMDA data table will be allowed; if `True`, extrapolation will be performed, assuming the rate coefficient is a powerlaw in temperature. This function returns nothing.

When this function is called, data on this collision partner is read from the LAMDA-formatted file pointed to by `fp`, and `fp` is advanced to the next collision partner, or to file end if this is the last collision partner listed. The routine initializes the sequence of interpolating functions `colRateInterp`, so that, after this function returns, `colRateInterp[n]` is a function that takes as an argument the natural log of the temperature, and returns the collision rate coefficient at that temperature, as determined by linearly interpolating in log temperature on the input data table.

8.8.2 `collPartner.colRateMatrix`

Call signature:

```
k = collPartner.colRateMatrix(self, temp, levWgt, levTemp)
```

This routine calculates collision rate coefficients between levels by interpolating on the stored data table. The argument `temp` is a float that gives the gas kinetic temperature in K, the argument `levWgt` is an array giving the statistical weights of the levels, and `levTemp` is an array giving the energies of the levels (measured in K). The routine returns a matrix of size `nlev` $\times$ `nlev` that contains the collision rate coefficient for transitions between states at the specified gas temperature. Element ij of the matrix is the rate coefficient from state i to state j.

8.8.3 `collPartner.colRates`

Call signature:

```
rates = colRates(self, temp, transition=None)
```

This routine calculates collision rate coefficients by interpolating on the stored data table. The argument `temp` is a float or an array of floats that gives the gas kinetic temperature in K. The optional argument `transition` specifies which transitions are to be computed. If it is `None`, then the rates for all transitions are computed; if it is an array of integers of shape $(2, N)$ then elements $(0, i)$ and $(1, i)$ are interpreted as the identifying the lower and upper levels of the transition(s) whose rate coefficients are to be computed. The return value `rates` is an array of interpolated collision rate coefficients. If `temp` is a float then it is an array of shape N, where N is the number of transitions that have been computed; if `temp` is an array of floats, then `rates` is of shape $(N, \texttt{len(temp)})$.

8.9 lineProfLTE

The `lineProfLTE` module provides the routine `lineProfLTE`, which is capable of computing the brightness temperature as a function of velocity for clouds in LTE. The call signature is:

```
TB, v = 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)
```

The first argument `emdat` is an object of class `emitterData`, which gives the data for the emitter whose line is to be computed. The arguments `u` and `l` are integers giving the upper and lower quantum state for the line whose profiles is to be computed. The argument `R` is the cloud radius in cm. The arguments `denProf`, `TProf`, `vProf`, and `sigmaProf` are all either floats or callables. If they are floats, they specify the constant value of the number density of emitting molecules, gas kinetic temperature, bulk radial velocity, and non-thermal velocity dispersion within the cloud, in CGS units. If they are callables, they must accept as an argument a single float in the range $0 - 1$, which gives the radial position normalized to the cloud radius `R`. They must return a single float that gives the number density of emitters, gas kinetic temperature, bulk radial velocity, or non-thermal velocity dispersion at that radius, again in CGS units.

The optional argument `offset` is a float in the range $0 - 1$ that specifies the offset of the line of sight from the cloud's projected center, normalized to the cloud radius. The optional argument `TCMB` gives the cosmic microwave background temperature.

The optional arguments `vOut`, `vLim`, `nOut`, and `dv` provide mechanisms to control the velocities at which the brightness temperature of the line is computed. Here `vOut` is a sequence that specifies the exact velocities at which the computation should be performed, `vLim` gives minimum and maximum velocities for the computation, `nOut` specifies the number of velocities at which to output, evenly spaced between `vLim[0]` and `vLim[1]`, and `dv` specifies the spacing in velocity between outputs. Setting more than the minimum number of these parameters required to specify a set of velocities will produce unpredictable behavior. If `vLim` and `vOut` are both `None`, the velocity range over which the computation is performed is from $-5\sigma_{\rm tot}(R) - |v(R)|$ to $5\sigma_{\rm tot}(R) + |v(R)|$, where $\sigma_{\rm tot}$ is the total velocity dispersion including both thermal and non-thermal contributions, and v is the bulk velocity. All quantities are in CGS units.

Finally, the optional argument `mxstep` gives the maximum number of steps to allow in the ODE integrator before returning with an error. The default value is sufficient for most computations, but clouds with exceptionally rapidly-varying profiles of one or more quantities may require larger values.

The function returns two arrays `TB` and `v`. The latter gives the velocities at which the brightness temperature has been computed (in cm $\rm s^{-1}$), and the former gives the brightness temperature at those velocities (in K).

8.10 fetchLamda

The `fetchLamda` module provides the routine `fetchLamda`, a utility for downloading Leiden Atomic and Molecular Database (LAMDA) files. The call signature is:

```
fname = fetchLamda(inputURL, path=None, fileName=None)
```

When this function is run, it downloads a single file and stores it locally. The argument `inputURL` is a string that specifies the file to be downloaded. It may either be an absolute URL, beginning with "http://", or a relative URL. If it is a relative URL, then the absolute URL is constructed by prepending `$DESPOTIC_LAMDAURL/datafiles/`, where `$DESPOTIC_LAMDAURL` is an environment variable, or `http://home.strw.leidenuniv.nl/~moldata/datafiles/` if the environment variable is not set. The optional argument `path` is a string giving a relative or absolute path where the downloaded file is to be stored. If it is `None`, the file is stored in the current working directory. If the specified path does not exist, it is created. Finally, the optional argument `fileName` is the name to give to the file. If it is `None`, then the local file name will be the same as the name of the file that is downloaded.

This routine returns the name of the file created if it succeeds, or `None` if the download is unsuccessful. Note that failed downloads or writes of files do not raise errors, they simply cause `None` to be returned.

8.11 refreshLamda

The `refreshLamda` module provides the routine `refreshLamda`, a utility for manually updating the local cache of the Leiden Atomic and Molecular Database (LAMDA). The call signature is:

```
refreshLamda(path=None, cutoffDate=None, cutoffAge=None, LamdaURL=None)
```

By default this routine examines every file in the directory `$DESPOTIC_HOME/LAMDA`, where `$DESPOTIC_HOME` is an environment variable; if this environment variable is unset, it checks the directory `LAMDA` relative to the current working directory. The optional argument `path` is a string that specifies the path where files are located, and overrides these defaults it is set to something other than `None`. For each file in the target directory, it checks the date of last modification, flagging those files that are older than a specified age or cutoff date. By default files older than 6 months are flagged, but the user can manually specify a different cutoff age or date using the optional arguments `cutoffAge` or `cutoffDate`; cutoff ages must be objects of the class `datetime.timedelta`, and cutoff dates must be objects of the class `datetime.datetime or \verb`datetime.date=. If any files are flagged, the routine then attempts to download a replacement for that file using the `fetchLamda` routine (see Section 8.10). It searches for the replacement at a URL given by `LamdaURL` if that optional argument is set to a string, or at the default location searched by `refreshLamda` if it is `None`.

This procedure returns nothing.

8.12 chemEvol

8.13 setChemEq

8.14 chemNetwork

8.15 NL99

8.16 shielding

8.17 abundanceDict

The `abundanceDict` module defines the `abundanceDict` class, a class that provides a dict-like interface to chemical abundances. An instance of this class is created via the method `abundanceDict.__init__(self, specList, x)` where `specList` is a list of strings giving the names of all the species, and `x` is a numPy array of rank >= 1 whose first dimension must be of the same size as the number of elements in `specList`.

Once instantiated, `abundanceDict` behaves exactly as would a dict whose keys are `specList` and whose corresponding values are the elements of `x`; all the methods available for dict are available for `abundanceDict`, including `__getitem__`, `__setitem__`, `__iter__`, `__len__`, `__contains__`, `keys`, `values`, `has_key`, and `copy`. In addition, `abundanceDict` defines a `__repr__` method that provides a nicely-formatted printed version of the information. The only difference from a standard dict is that the list of keys is immutable once the `abundanceDict` is instantiated, so that the dict methods `__delitem__`, `clear`, `pop`, `popitem`, and `update` are all disallowed, as is the use of `__setitem__` with any key that does not match one of the keys in the `abundanceDict` at the time it was instantiated.

9 Revision History

1. Version 1.1.0, 5/2013 – added chemistry capabilities; also numerous minor bug fixes
2. Version 1.0.4, 4/2013 – fixed a bug with reading LAMDA files, added `XH2` and `H2OPR` as `cloud` properties
3. Version 1.0.3, 4/2013 – fixed a bug introduced in version 1.0.2; also added section on dependencies to the manual; no functional changes
4. Version 1.0.2, 4/2013 – revised the way residuals are calculated in `cloud.setTempEq`, `cloud.setDustTempEq`, and `cloud.setGasTempEq` to provide a higher probability of convergence when the initial guess is very far from the correct answer.
5. Version 1.0.1, 4/2013 – added license statements to all source files.
6. Version 1.0, 4/2013 – initial release