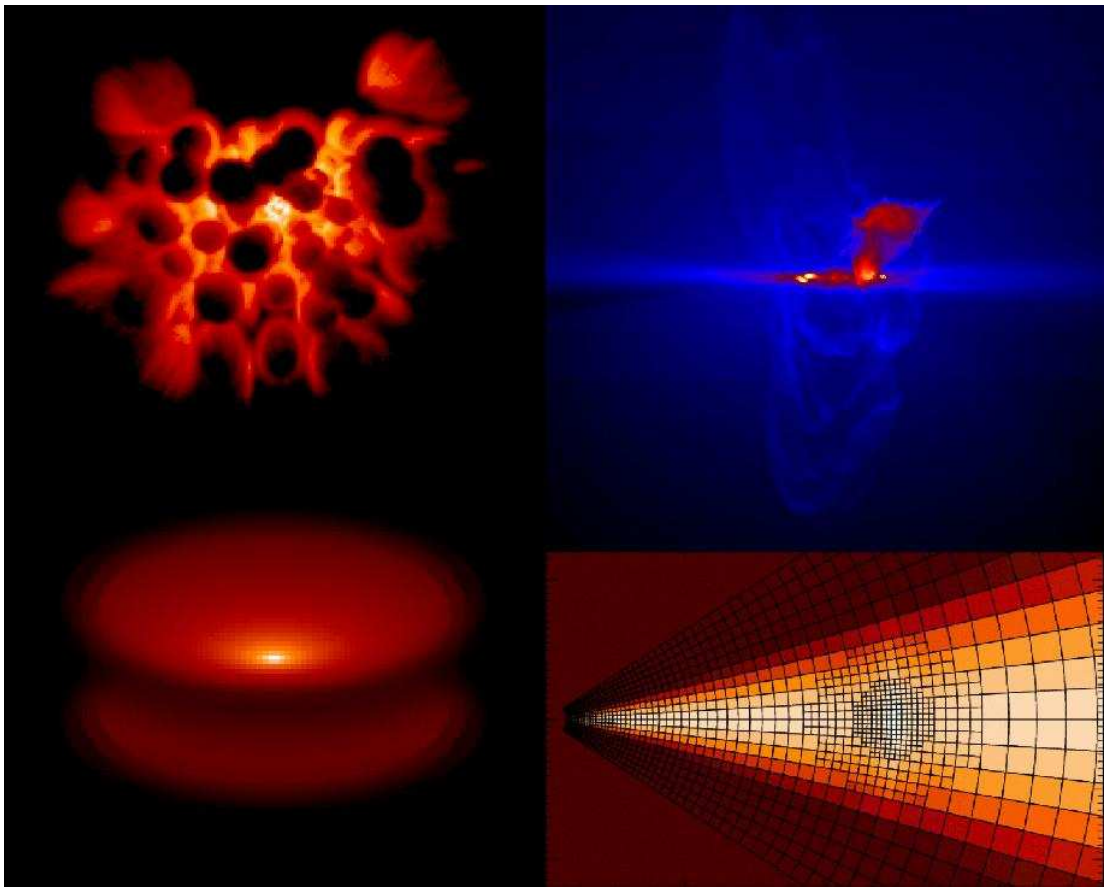
Manual for RADMC-3D
Version 0.26
Manual Version 0.26-1

C.P. Dullemond

with substantial contributions from:
A. Juhasz, R. Shetty

and testing by:
Rainer Rolffs, Laszlo Szucs



Front page images: *Simulations with RADMC-3D. Upper-left: A simple model of a clumpy dust "torus" around an active galactic nucleus. Upper-right: A simulation of high-mass star formation by Thomas Peters (ITA, University of Heidelberg, 2010) using the FLASH MHD code. Lower-left: A simple 2-D axisymmetric model of a protoplanetary disk seen in scattered light. Lower-right: An example of the kind of grid refinement possible with RADMC-3D.*

Contents

Chapter 1

Introduction, copyright and disclaimer

1.1 Introduction

RADMC-3D is a software package for astrophysical radiative transfer calculations in arbitrary 1-D, 2-D or 3-D geometries. It is mainly written for continuum radiative transfer in dusty media, but also includes modules for gas line transfer and gas continuum transfer.

RADMC-3D is a new incarnation of an older software package called RADMC. The original RADMC package was written in Fortran 77 and was only for axially symmetric problems in spherical coordinates. Because it was written in Fortran 77, the arrays had a fixed maximum size, so whenever a new grid was necessary, the code had to be recompiled. RADMC was also ageing in many other ways, in the sense that it used input formats that stemmed from the very early developing phase, and were not particularly practical. Also, RADMC's limitation to axisymmetric configurations and rigid gridding made it not capable of dealing with more complex 3-D models that are now becoming ever more mainstream. For these reasons I decided to make a huge make-over of the code, or more precise: to build a new incarnation of RADMC, called RADMC-3D, almost completely from scratch.

At the moment RADMC-3D is still in the development phase, but is is already reasonably mature. Here is a list of current and planned features. Those features that are now already working are marked with [+], while those which are not yet (!!) built in are marked with [-]. Those that are currently being developed are marked with [.] and those that are ready, but are still in the testing phase are marked with [t].

- Coordinate systems:
  - [+] Cartesian coordinates (3-D)
  - [+] Spherical coordinates (1-D, 2-D and 3-D)
- Gridding systems (regular and adaptive mesh refinement grids are available for cartesian *and* spherical coordinates):
  - [+] Regular
  - [+] Adaptive Mesh Refinement: oct-tree style
  - [+] Adaptive Mesh Refinement: layered ('patch') style
  - [-] Delaunay gridding *[To be implemented on request]*
  - [-] Voronoi gridding *[To be implemented on request]*
- Radiation mechanisms:
  - [+] Dust continuum, thermal emission
  - [t] Dust continuum scattering:
    - [+] ...in isotropic approximation
    - [t] ...with full anisotropy

[-] Dust quantum heated grains *[To be implemented on request]*

[-] Polarized light *[To be implemented on request]*

[t] Gas line transfer (LTE)

[-] Gas line transfer (non-LTE: LVG)

[-] Gas line transfer (non-LTE: full transfer)

[t] Gas line transfer with user-defined populations

[+] Gas continuum opacity and emissivity sources

- Radiation netto sources for continuum:

  [+] Discrete stars positioned at will

  [t] Continuous 'starlike' source

  [-] Continuous 'dissipation' source

  [t] External 'interstellar radiation field'

- Imaging options:

  [+] Easy-to-use IDL front-end widget interface for imaging

  [+] Observer from 'infinite' distance

  [+] Zoom-in at will

  [+] Flux-conserving imaging, i.e. pixels are recursively refined

  [+] A movie-making tool

  [+] Multiple wavelengths in a single image

  [+] Local observer with perspective view (for PR movies!)

- Spectrum options:

  [+] SED spectrum (spectrum on 'standard' wavelength grid)

  [+] Spectrum on any user-specified wavelength grid

  [+] Spectrum of user-specified sub-region (pointing)

  [t] Specification of size and shape of a primary 'beam' for spectra

- User flexibility:

  [+] Free model specification via tabulated input files

  [+] Easy special-purpose compilations of the code (optional)

- Front-end IDL packages:

  [+] Example model setups

  [+] Image viewing GUI (graphical user interface)

- Miscellaneous:

  [+] Stars can be treated as point-sources or as spheres

  [+] Option to calculate the mean intensity $J_\nu(\vec{x})$ in the model

  [-] Support for parallel computing

1.2 Copyright and disclaimer

The use of this software is free of charge. However, it is not allowed to distribute this package without prior consent of the lead author (C.P. Dullemond). Please refer any interested user to the web site of this software where the package is available, which is currently:

http://www.mpia.de/homes/dullemon/radmc-3d/index.php

IMPORTANT NOTICE 1: I/We reject all responsibility for the use of this package. The package is provided as-is, and we are not responsible for any damage to hardware or software, nor for incorrect results that may result from the software. The user is fully responsible for any results from this code, and we strongly recommend thorough testing of the code before using its results in any scientific papers.

IMPORTANT NOTICE 2: Any publications which involve the use of this software must mention the name of this software package and cite the accompanying paper once it is published (Dullemond et al. in prep), or before that the above mentioned web site.

IMPORTANT NOTICE 3: If you use this software, you may want to notify the lead author (C.P. Dullemond) so that you are put on an email list. This ensures that you are always up to date with major bug reports and major updates. This mail list is only used for important enough news, so you will not be flooded with emails and you can always unsubscribe.

Chapter 2

Quick-Start

In general I recommend reading the manual fully, but it is often useful to get a quick impression of the package with a quick-start. To make your first example model, this is what you do:

1. When you read this you have probably already unzipped this package. You should find, among others, a `src/` directory and a `examples/` directory. Go into the `src/` directory.
2. Edit the `src/Makefile` file, and make sure to set the `FF` variable to the Fortran-90 compiler you have installed on your system.
3. Type `make`. If all goes well, this should compile the entire code and create an executable called `radmc3d`.
4. Type `make install`. If all goes well this should try to create a link to `radmc3d` in your `~/bin/` directory. If this directory does not exist, it will ask to make one.
5. Make sure to have the `~/bin/` directory in your path. If you use, for instance, the `tcsh` shell, you do this by setting the `path` variable: `set path = ( ~/bin $path )` in your `~/.tcshrc` file. If you change these things you may have to open a new shell to make sure that the shell now recognizes the new path.
6. Check if the executable is OK by typing `radmc3d` in the shell. You should get a small welcoming message by the code.
7. Now enter the directory `examples/run_simple_1/`.
8. Copy all standard IDL (see Section 3.2 about IDL) routines from the `../../idl/` directory into the current directory by typing in the tcsh or bash shell `cp ../../idl/*.pro ./`. *NOTE: This is a quick-and-dirty way of using the IDL routines, only meant to get the stuff running quickly without going through the somewhat more involved IDL routines installation procedure described in Section 4.4. For the the proper use of the RADMC-3D package, it is recommended to follow the procedures described in Section 4.4.*
9. Enter IDL.
10. Type (in IDL) `.r problem_setup.pro`, and after that `exit` to exit IDL again.
11. Type `radmc3d mctherm`. This should let the code do a Monte Carlo run. You should see `Photon nr 1000`, followed by `Photon nr 2000`, etc until you reach `Photon nr 1000000`. The Monte Carlo modeling for the dust temperatures has now been done.
12. Go into IDL again and type `.r viewimage.pro` followed by `viewimage`. This should bring an image viewer on the screen and show what the simple model looks like when rendered at some angle[1]. The model is very simple: a spherical blob, so do not expect to see much in this simple example.

[1] Note: on some systems there is an apparent problem with the communication pipe between `radmc3d` and IDL which causes things to freeze. Try typing `viewimage,/nochild` in that case, which should fix the problem, although the viewer may then be substantially slower. I am working on figuring out how the problem can be fixed, but have so far been not succesful.

If you experience troubles with the above steps, and you cannot fix it, please read the next chapters for more details.

Tip: If the code unexpectedly quits or freezes while using `viewimage`*, please have a look at the file* `radmc3d.out` *which contains the messages that RADMC-3D outputs. This may give hints what went wrong. Note that this file is only written if RADMC-3D is used in child mode, which is the case when it is spawned from viewimage. Otherwise this output will be written to screen. Also, when viewimage is called with the option* `,/nochild` *the output will also be written to screen instead of the file* `radmc3d.out`.

Chapter 3

Overview of the RADMC-3D package

3.1 Introduction

The RADMC-3D code is written in fortran-90 and should compile with most f90 compilers without problems. It needs to be compiled only once for each platform. *Note that the code is developed for Unix-based systems such as linux machines or Mac OS X machines. It may also work on Windows machines, but this is not guaranteed, and throughout this manual a Unix-based machine is assumed, with a csh, tcsh or bash shell. User-level knowledge of Unix-like operating systems is required.*

The executable is called `radmc3d` and it performs all the model calculations of the RADMC-3D package, for instance the Monte Carlo simulations, ray-tracing runs (images, spectra), etc. There is also a set of useful subroutines written in the IDL[1] language to use the `radmc3d` code, but `radmc3d` can also run without IDL. In that case the user will have to write his/her own pre- and post-processing subroutines in e.g. python or other data processing languages.

3.2 Requirements

This package runs under linux/unix/MacOSX, but has not been tested under Windows. The following pre-installed software is required:

- `make` or `gmake`
  This is the standard tool for compiling packages on all Unix/Linux-based systems.
- `perl`
  This is a standard scripting language available on most or all Unix/Linux-based systems. If you are in doubt: type `which perl` to find the location of the `perl` executable. See `http://www.perl.org/` for details on perl, should you have any problems. But on current-day UNIX-type operating systems perl is nearly always installed in the `/usr/bin/` directory.
- `A fortran-90 compiler`
  Preferably the `gfortran` compiler (which the current installation assumes is present on the system). Web site: `http://gcc.gnu.org/fortran/`. Other compilers may work, but have not been tested yet.
- `The IDL package (Interactive Data Language)`
  `IDL` is a software package similar to MatLab, and it is not free. While RADMC-3D can be used without IDL, all examples and all post-processing scripts are written in IDL, so it would require the user to rewrite them into other languages (fortran, c, c++, perl, python or whatever). The website for IDL is: `http://www.ittvis.com/`. If `IDL` is not present on your system, and your system administrators cannot install this package due to lack of funding, you can also use an open source clone called `GDL` (Gnu Data Language) which can be readily downloaded from the web (`http://gnudatalanguage.sourceforge.net/`).

[1] IDL is a commercial data processing package used frequently among astrophysicists. See `http://www.ittvis.com/idl/` for more information.

This GDL package misses some libraries and features, but the RADMC-3D code can be used with GDL with the exception that the Graphical User Interfaces of RADMC-3D (such as viewimage.pro) cannot be used.

Note that the Monte Carlo code RADMC-3D itself is in Fortran-90. Only the creation of the input files (and hence the problem definition) and the analysis of the output files is done in IDL. The user is of course invited to use other ways to create the input files for RADMC-3D if he/she is not able to use IDL. Therefore IDL are not strictly required for the use of this code.

3.3 The archive, how to unzip it, and what it contains

The package of RADMC-3D is packed in a zip archive called `radmc-3d_v*.**_##.##.##.zip` where the `*.**` is the version number and `##.##.##` is the date of this version in dd.mm.yy format. To unpack on a linux, unix or Mac OS X machine you type:

```
unzip <this archive file>
```

i.e. for example for radmc-3d_v0.07_27.07.09.zip you type

```
unzip radmc-3d_v0.07_29.07.09.zip
```

A directory `radmc-3d` is created which has the following subdirectory structure:

```
radmc-3d/
   src/
   idl/
   examples/
      run_simple_1/
      run_simple_1_userdef/
      run_simple_1_userdef_refined/
      .
      .
      .
   manual/
```

The first directory, `src/`, contains the fortran-90 source code for RADMC-3D. The second directory, `idl/`, contains a set of IDL routines that are useful for model preparation and post-processing. The third directory contains a series of example models. The fourth directory contains this manual.

3.4 Units: RADMC3D uses CGS

The RADMC-3D package is written such that all units are in CGS (length in cm, time in sec, frequency in Hz, energy in erg, angle in steradian). There are exceptions:

- Wavelength is usually written in micron
- Sometimes angles are in degrees (internally in radian, but input as degrees)

Chapter 4

Compilation and installation of radmc3d

Although the RADMC-3D package contains a lot of different software, the main code is located in the `src/` directory, and is written in Fortran-90. The executable is `radmc3d`. Here we explain how to compile the fortran-90 source codes and create the executable `radmc3d`.

4.1 Compiling the code with 'make'

To compile the code, enter the `src/` directory in your shell (we assume a tcsh shell here, but bash or other Unix-shells are also fine). You now *may* need to edit the `Makefile` in this directory using your favorite text editor and replace the line

```
FF = gfortran
```

with a line specifying your own compiler. If, of course, you use gfortran, you can keep this line. But if you use, e.g., ifort, then replace the above line by

```
FF = ifort
```

If you save this file, and you are back in the shell, you can compile the radmc3d code by typing

```
make
```

in the shell. If all goes well, you have now created a file called `radmc3d` in the `src/` directory.

4.2 The install.perl script

If instead of typing just 'make' you type

```
make install
```

(or you first type 'make' and then 'make install', it's the same), then in addition to creating the executable, it also automatically executes a perl script called `install.perl` (located also in the `src/` directory) that installs the code in such a way that it can be conveniently used in any directory. What it does is:

1. It checks if a `bin/` directory is present in your home directory (i.e. a `~/bin/` directory). If not, it asks if you want it to automatically make one.
2. It checks if the `~/bin/` directory is in the 'path' of the currently used shell. This is important to allow the computer to look for the program 'radmc3d' in the `~/bin/` directory. If you use a csh or tcsh shell, then you can add the following line to your ~/.tcshrc file:

```
set path=($HOME/bin $path)
```

3. It creates a file `radmc3d` in this ∼`/bin/` directory with the correct executable permissions. This file is merely a dummy executable, that simply redirects everything to the true `radmc3d` executable located in your current `src/` directory. When you now open a new shell, the path contains the ∼`/bin/` directory, and the command `radmc3d` is recognized. You can also type `source ∼/.tcshrc` followed by `rehash`. This also makes sure that your shell recognizes the `radmc3d` command.
4. It checks if a `idl/` subdirectory exists in the above mentioned `bin` directory, i.e. a ∼`/bin/idl/` directory. If not, it asks if you want it to automatically create one.
5. If yes, then it will copy all the files ending with `.pro` in the `idl/` directory of the distribution to that ∼`/bin/idl/` directory. This is useful to allow you to make an `IDL_PATH` entry to allow idl to find these idl scripts automatically (see Section 4.4).

Note that this perl script installs the code only for the user that installs it. A system-wide installation is, in my view, not useful, because the code package is not very big and it should remain in the control of the user which version of the code he/she uses for each particular problem.

If all is 'normal', then the `perl.install` script described here is called automatically once you type `make install` following the procedure in Section 4.1.

Before the installation is recognized by your shell, you must now either type `rehash` in the shell or simply open a new shell.

How do you know that all went OK? If you type `radmc3d` in the shell the RADMC-3D code should now be executed and give some comments. It should write:

```
============================================================
          WELCOME TO RADMC-3D: A 3-D CONTINUUM RT SOLVER

           This is the 3-D version of the 2-D RADMC code
                 (c) 2008 Cornelis Dullemond
============================================================

 Nothing to do... Use command line options to generate action:
   mctherm        : Do Monte Carlo simul of thermal radiation
   mcscat         : Do Monte Carlo simul only for scattering
   spectrum       : Make continuum spectrum
   image          : Make continuum image
```

on the screen (or for newer versions of RADMC-3D perhaps some more or different text). This should also work from any other directory.

4.3 What to do if this all does not work?

In case the above compilation and installation does not work, here is a proposed procedure to do problem hunting:

- First, answer the following questions:
  - Did you type `make install` in the `src/` directory? I mean, did you not forget the `install` part?
  - Did you put ∼`/bin/` in your path (see above)?
  - If you just added ∼`/bin/` to your path, did you follow the rest of the procedure (either closing the current shell and opening a new shell or typing the `source` and `rehash` commands as described above)?

  If this does not help, then continue:
- Close the shell, open a new shell.
- Go to the RADMC-3D `src/` directory.

- Type `./radmc3d`. This should give the above message. If not, then make sure that the compilation went right in the first place:
  - Type `rm -f radmc3d`, to make sure that any old executable is not still present.
  - Type `make clean`. This should return the sentence `OBJECT and MODULE files removed.`
  - Then type `make`. This should produce a set of lines, each representing a compilation of a module, e.g. `gfortran -c -O2 ./amr_module.f90 -o amr_module.o`, etc. The final line should be something like `gfortran -O2 main.o rtglobal_module.o montecarlo_module.o dust_module.o quantum_module.o mathroutines_module.o ioput_module.o stars_module.o amr_module.o amrray_module.o constants_module.o camera_module.o lines_module.o namelist_module.o userdef_module.o gascontinuum_module.o -o radmc3d`. If instead there is an error message, then do the following:
    - Check if the compiler used (by default `gfortran`) is available on your computer system.
    - If you use an other compiler, check if the compiler options used are recognized by your compiler.
  - Check if the executable `radmc3d` is now indeed present. If it is not present, then something must have gone wrong with the compilation. So then please check the compilation and linking stage again carefully.

  If you followed all these procedures, but you still cannot get even the executable in the `src/` directory to run by typing (in the `src/` directory) `./radmc3d` (don't forget the dot slash!), then please contact the author.
- At this point I assume that the previous point worked. Now go to another directory (any one), and type `radmc3d`. This should also give the above message. If not, but the `radmc3d` executable was present, then apparently the shell path settings are wrong. Do this:
  - Check if, in the current directory (which is now not `src/`) there is by some accident another copy of the executable `radmc3d`. If yes, please remove it.
  - Type `which radmc3d` to find out if it is recognized at all, and if yes, to which location it points.
  - Did you make sure that the shell path includes the `~/bin/` directory, as it should? Otherwise the shell does not know where to find the `~/bin/radmc3d` executable (which is a perl link to the `src/radmc3d` executable).
  - Does the file `~/bin/radmc3d` perl file exist in the first place? If no, check why not.
  - Type `less ~/bin/radmc3d` and you should see a text with first line being `#!/usr/bin/perl` and the second line being someting like `system("/Users/user1/radmc-3d/version_0.12/src/radmc3d @ARGV");` where the `/Users/user1` should of course be the path to your home directory, in fact to the directory in which you installed RADMC-3D.

If this all brings you no further, please first ask your system administrators if they can help. If not, then please contact the author.

4.4 Installing the IDL analysis tools

In the package there is a directory containing a whole series of analysis tools for analyzing the results of RADMC-3D. They are highly recommended, but not essential for using RADMC-3D. These tools are described in detail in Chapter 14.

The tools are written in IDL and you can find them in the `idl/` directory. To use them in a convenient way one must let IDL know where to find these routines. Since the install.perl script described above copies all these files to the `~/bin/idl/` directory, it is advisable to put that directory as the `IDL_PATH` instead of the local `idl` directory. The reason is that if you have multiple versions of RADMC-3D on your system, you always are assured that IDL finds the idl routines belonging to the latest installation of RADMC-3D (note: only assured if that latest compilation was done with `make install`).

In IDL here are two ways how you can make sure that IDL automatically finds the RADMC-3D scripts:

1. Under Unix/Linux/MacOSX you can set the `IDL_PATH` directly in your `.cshrc` or `.tcshrc` or `.bashrc` file. For example: in `.tcshrc` (if you use the tcsh shell) you can write:

```
setenv IDL_PATH "/myhomedirectory/bin/idl:/Applications/itt/idl70/lib:\
 /Applications/itt/idl70/lib/iTools:\
 /Applications/itt/idl70/lib/iTools/framework:\
 /Applications/itt/idl70/lib/iTools/components:\
 /Applications/itt/idl70/lib/iTools/ui_widgets"
```

(where `myhomedirectory` should be replaced by your home directory name). Note that the entire IDL default path also has to be added, as it is done here, otherwise most of IDL libraries are not working anymore. The disadvantage of this method is that if IDL adds further directories to its default path in the future, you would have to add them by hand here.

2. You can set the `IDL_PATH` in a more elegant way directly from within IDL with the command:

   ```
   PREF_SET, 'IDL_PATH', '/myhomedirectory/bin/idl:<IDL_DEFAULT>',/COMMIT
   ```

   (where `myhomedirectory` should be replaced by your home directory name). Of course, you do not want to have to type this line every time you start up IDL. So you can make a startup script that IDL executes every time it is started or reset. The way to do this is:

   (a) Make a script file, e.g. called `.idl_startup` in your home directory (Note: by starting the name with a "." it will remain invisible under Unix/Linux unless you type `ls -a`), containing the above line (i.e. containing `PREF_SET, 'IDL_PATH', '/myhomedirectory/bin/idl:<IDL_DEFAULT>',/COMMIT`).

   (b) In your `.tcshrc` or `.bashrc` file in your home directory set the `IDL_STARTUP` environment variable to `/myhomedirectory/.idl_startup`. For `.tcshrc` this works by adding a line `setenv IDL_STARTUP /myhomedirectory/.idl_startup`.

   If all goes well, if you now start IDL you should be able to have access to the IDL routines of RADMC-3D directly. To test this, try typing `.r viewimage` in IDL. If this gives an error message that `viewimage.pro` cannot be found, then please ask your system administrators how to solve this.

NOTE: You can also ignore all of this, and not copy any of the IDL routines to this central location, and instead simply copy all the `*.pro` files of the `idl/` directory that you use to the local model directory (see Section 5.6 for what we mean with 'model directory'). Or you could, in IDL, give the full path to each of the files. But these solutions are a lot messier.

4.5 Making special-purpose modified versions of RADMC-3D (optional)

For most purposes it should be fine to simply compile the latest version of RADMC-3D once-and-for-all, and simply use the resulting `radmc3d` executable for all models you make. Normally there is no reason to have to modify the code, because models can be defined quite flexibly by preparing the various input files for RADMC-3D to your needs. So if you are an average user, you can skip to the next subsection without problem.

But sometimes there *is* a good reason to want to modify the code. For instance to allow special behavior for a particular model. Or for a model setup that is simply easier made internally in the code rather than by preparing large input files. One can imagine some analytic model setup that might be easier to create internally, so that one can make use of the full AMR machinery to automatically refine the grid where needed. Having to do so externally from the code would require you to set up your own AMR machinery, which would be a waste of time.

The problem is that if the user would modify the central code for each special purpose, one would quickly lose track of which modification of the code is installed right now.

Here is how this problem is solved in RADMC-3D:

- For most purposes you can achieve your goals by only editing the file `userdef_module.f90`. This is a set of standard subroutines that the main code calls at special points in the code, and the user can put anything he/she wants into those subroutines. See Chapter 13 for more information about these standard subroutines. This method is the safest way to create special-purpose codes. It means (a) that you know that your modification cannot do much harm unless you make really big blunders, because these subroutines are meant to be modified, and (b) you have all your modifications *only* in one single file, leaving the rest of the code untouched.

- You can create a *local* version of the code, without touching the main code. Suppose you have a model directory `run_mymodel` and for this model you want to make a special-purpose version of the code. This is what you do:

  1. Copy the Makefile from the `src/` directory into `run_mymodel`.
  2. Copy the `.f90` file(s) you want to modify from the `src/` directory into `run_mymodel`. Usually you only want to modify the `userdef_module.f90` file, but you can also copy any other file if you want.
  3. In the `run_mymodel/Makefile` replace the `SRC = .` line with `SRC = XXXXXX`, where `XXXXXX` should be the *full* path to the `src/` directory. An example line is given in the Makefile, but is commented out.
  4. In the `run_mymodel/Makefile` make sure that all the `.f90` files that should remain as they are have a `$(SRC)/` in front of the name, and all the `.f90` files that you want to modify (and which now have a copy in the `run_mymodel` directory) have a `./` in front of the name. By default all `.f90` files have `$(SRC)/` in front of the name, except the `userdef_module.f90` file, which has a `./` in front of the name because that is the file that is usually the one that is going to be edited by you.
  5. Now edit the local `.f90` files in the `run_mymodel` directory in the way you want. See Chapter 13 for more details.
  6. Now *inside* the `run_mymodel` directory you can now type `make` and you will create your own local `radmc3d` executable. NOTE: Do not type `make install` in this case, because it should remain a local executable, only inside the `run_mymodel` directory.
  7. If you want (though this is not required) you can clean up all the local `.o` and `.mod` files by typing `make clean`, so that your `run_mymodel` directory is not filled with junk.
  8. You can now use this special purpose version of `radmc3d` by simply calling on the command line: `./radmc3d`, with any command-line options you like. Just beware that, depending on the order in which you have your paths set (in tcsh or bash) typing just `radmc3d` *may* instead use the global version (that you may have created in the `src/` directory with `make install`). So to be sure to use the *local* version, just put the `./` in front of the `radmc3d`.

Note: In chapter 13 there is more information on how to set up models internally in the code using the method described here.

Note: You can use `make clean` to remove all the .o and .mod files from your model directory, because they can be annoying to have hanging around. By typing `make cleanmodel` you remove, in addition to the .o and .mod files, also all model input and output files, with the exception of dust opacity or molecular data files (because these latter files are usually not created locally by the `problem_setup.pro` script). By typing `make cleanall` you remove everything *except* the basic files such as the `Makefile`, any `.f90` files, any `.pro` files, the dust opacity or molecular data files and `README` files.

Chapter 5

Basic structure and functionality

RADMC-3D is a very versatile radiative transfer package with many possibilities. As a consequence it is a rather complex package. However, we have tried to keep it still as easy as possible to use as a first-time user. We tried to do so by keeping many of the sophisticated options "hidden" and having many default settings already well-chosen. The idea is that one can already use the code at an entry level, and then gradually work oneself into the more fancy options.

RADMC-3D is a general-purpose package, so there are no 'built-in' models inside the `radmc3d` executable[1]. For instance, if you want to model a protoplanetary disk, then you would have to design the grid and density structure of the disk on this grid yourself. To make it easier for the user, we have provided several IDL-scripts as examples. Among these examples is indeed a protoplanetary disk model. So this is as close as we go to 'built-in' models: we provide, for some cases, already well-developed example models that you, the user, can use out-of-the-box, or that you can adapt to your needs.

In this chapter we give an overview of the rough functionality of the code in its simplest form: ignoring all the hidden fancy options and possibilities. For the details we then refer to the chapters ahead.

5.1 Radiative processes

Currently RADMC-3D handles the following radiative processes:

- *Dust thermal emission and absorption*
  RADMC-3D can compute spectra and images in dust continuum. The dust temperature must be known in addition to the dust density. In typical applications you will know the dust density distribution, but not the dust temperature, because the latter is the results of a balance between radiative absorption and re-emission. So in order to make spectra and images of a dusty object we must first calculate the dust temperature consistently. This can be done with RADMC-3D by making it perform a "thermal Monte Carlo" simulation (see Chapter 6). This can be a time-consuming computation. But once this is done, RADMC-3D writes the resulting dust temperatures out to the file `dust_temperature.dat`, which it can then later use for images and spectra. We can then call RADMC-3D again with the command to make an image or a spectrum (see Chapter 6). To summarize: a typical dust continuum radiative transfer calculation goes in two stages:
  1. A thermal Monte Carlo simulation with RADMC-3D to compute the dust temperatures.
  2. A spectrum or image computation using ray-tracing with RADMC-3D.
- *Dust scattering*
  Dust scattering is automatically included in the thermal Monte Carlo simulations described above, as well as in the production of images and spectra. For more details, consult Chapter 6.
- *Gas atomic/molecular lines*
  RADMC-3D can compute spectra and images in gas lines (see Chapter 7). The images are also known

[1] Except if you insert one yourself using the userdef module, see Chapter 13.

as "channel maps". To compute these, RADMC-3D must know the population densities of the various atomic/molecular levels. For now there are two options how to let RADMC-3D know these values:

- Tell RADMC-3D to assume that the molecules or atoms are in "Local Thermodynamic Equilibrium" (LTE), and specify the gas temperature at each location to allow RADMC-3D to compute these LTE level populations. *Note that in principle one is now faced with the same problem as with the dust continuum: we need to know the gas temperature, which we typically do not know in advance.* However, computing the gas temperature self-consistently is very difficult, because it involves many heating and cooling processes, some of which are very complex. That's why most line radiative transfer codes assume that the user gives the gas temperature as input. We do so as well. If you like, you can tell RADMC-3D to use the (previously calculated) dust temperature as the gas temperature, for convenience.
- Deliver RADMC-3D an input file with all the level populations that you have calculated youself using some method.
- Tell RADMC-3D to compute the level populations according to some simple local non-LTE prescription such as the Sobolev approximation ("Large Velocity Gradient method") or the Escape Probability Method. (This is still under development!)

Currently RADMC-3D does not have a full non-local non-LTE computation method implemented. The reason is that it is very costly, and for many applications presumably not worth the computational effort. But we are working on a full non-LTE mode. Stay tuned!

- *Gas continuum opacities*
  We are currently working on implementing gas continuum opacities as well. Again we are faced with the question how to compute the gas temperature. For now we simply require you to specify the gas temperature yourself.

Remark: We are thinking of methods to compute gas temperatures self-consistently in some special situations. Stay tuned...

5.2 Coordinate systems

With RADMC-3D you can specify your density distribution in two coordinate systems:

- *Cartesian coordinates*
  The simplest coordinate system is the Cartesian coordinate system (x, y, z). For now each model must be 3-D (i.e. you must specify the densities and other quantities as a function of x, y and z). But in the near future we plan to also include the possibility of 1-D plane-parallel models.
- *Spherical coordinates*
  You can also specify your model in spherical coordinates (r, θ, ϕ). These coordinates are related to the cartesian ones by:

$$x = r\sin\theta\cos\phi \tag{5.1}$$
$$y = r\sin\theta\sin\phi \tag{5.2}$$
$$z = r\cos\theta \tag{5.3}$$

  This means that the spatial variables (density, temperature etc) are all specified as a function of (r, θ, ϕ). However, the location of the stars, the motion and direction of photon packages etc. are still given in cartesian coordinates (x, y, z). In other words: any function of space $f(\vec{x})$ will be in spherical coordinates $f(r, \theta, \phi)$, but any point-like specification of position $\vec{x}$ will be given as Cartesian coordinates $\vec{x} = (x, y, z)$. This hybrid method allows us to do all physics in cartesian coordinates: photon packages or rays are treated always in cartesian coordinates, and so is the physics of scattering, line emission etc. Only if RADMC-3D needs to know what the local conditions are (dust temperature, gas microturbulence, etc) RADMC-3D looks up which coordinates (r, θ, ϕ) belong to the current (x, y, z) and looks up the value of the density, microturbulence etc. at that location in the (r, θ, ϕ) grid. And the same is true if RADMC-3D updates or calculates for instance the dust temperature: it will compute the (r, θ, ϕ) belong to the current (x, y, z) and update the temperature in the

cell belonging to (r, θ, ϕ). For the rest, all the physics is done in the Cartesian coordinate system. This has the major advantage that we do not need different physics modules for cartesian and spherical coordinates. Most parts of the code don't care which coordinate system is used: they will do their own work in Cartesian coordinates. When using spherical coordinates, please read Section 10.2.

5.3 The spatial grid

To specify the density or temperature structure (or any other spatial variable) as a function of spatial location we must have a grid. There are two basic types of grids:

- *Structured grids (AMR grids)*
  The standard gridding is a simple rectangular grid.
  - *Cartesian coordinates:* When cartesian coordinates are used, this simply means that each cell is defined as $x_l < x < x_r$, $y_l < y < y_r$ and $z_l < z < z_r$, where l and r stand for the left and right cell walls respectively.
  - *Spherical coordinates:* When spherical coordinates are used, this simply means that each cell is defined as $r_l < r < r_r$, $\theta_l < \theta < \theta_r$ and $\phi_l < \phi < \phi_r$. Note therefore that the shape of the cells in spherical coordinates is (in real space) curved. For spherical coordinates the following four modes are available:
    - *1-D Spherical symmetry:* All spatial functions depend only on r.
    - *2-D Axial symmetry:* All spatial functions depend only on r and θ.
    - *2-D Axial symmetry with mirror symmetry:* All spatial functions depend only on r and θ, where the θ grid only covers the part above the $z = 0$ plane. Internally it is in this mode assumed that all quantities below the $z = 0$ plane are equal to those above the plane by mirror symmetry in the $z = 0$ plane. This saves a factor of two in computational effort for Monte Carlo calculations, as well as in memory useage. Note that also the resulting output files such as `dust_temperature.dat` will only be specified for $z > 0$.
    - *3-D:* All spatial functions depend on all three variables r, θ and ϕ.
    - *3-D with mirror symmetry:* All spatial functions depend on all three variables r, θ and ϕ, but like in the 2-D case only the upper part of the model needs to be specified: the lower part is assumed to be a mirror copy.

    When using spherical coordinates, please read Section 10.2.

  In all cases these structured grids allow for oct-tree-style grid refinement, or its simplified version: the layer-style grid refinement. See Section A.2 and Chapter 10 for more information about the gridding and the (adaptive) mesh refinement (AMR).

- *Unstructured grids (for now: cartesian coordinates only)*
  For some applications it may be more convenient to specify spatial variables not on a structured grid, but on a semi-random set of points in 3-D space. RADMC-3D will (hopefully) soon feature a mode in which it can handle such a situation, and use a Delaunay triangulation to produce a *triangulation* on the basis of this set of points, thus creating an *unstructured grid*. Unfortunately, for now this mode is not yet ready. Stay tuned...

5.4 Computations that RADMC-3D can perform

The code RADMC-3D (i.e. the executable `radmc3d`) is *one* code for *many* actions. Depending on which command-line arguments you give, RADMC-3D can do various actions. Here is a list:

- *Compute the dust temperature:*
  With `radmc3d mctherm` you call RADMC-3D with the command of performing a thermal Monte Carlo simulation to compute the dust temperature under the assumption that the dust is in radiative equilibrium with its radiation field. This is normally a prerequisite for computing SEDs and images from dusty objects (see "computing spectra and images" below). The output file of this computation is `dust_temperature.dat` which contains the dust temperature everywhere in the model.

- *Compute a spectrum or SED:*
  With `radmc3d sed` you call RADMC-3D with the command of performing a ray-tracing computation to compute the spectral energy distribution (SED) for the model at hand. Typically you first need to have called `radmc3d mctherm` (see above) beforehand to compute dust temperatures (unless you have created the file `dust_temperature.dat` yourself because you have a special way of computing the dust temperature). With `radmc3d sed` the spectrum is computed for the wavelengths points given in the file `wavelength_micron.inp`, which is the same wavelength grid that is used for `radmc3d mctherm`. If you want to compute the spectrum at wavelength other than those used for the thermal Monte Carlo simulation, you should instead call `radmc3d spectrum`, and you have the full freedom to choose the spectral wavelengths points at will, and you can specify these in various ways described in Section 9.4. Most easily you can create a file called `camera_wavelength_micron.inp` (see Section A.12) and call RADMC-3D using `radmc3d spectrum loadlambda`. In all these cases the vantage point (where is the observer) can of course be set as well, see Section 6.2 and Chapter 9.

- *Compute an image:*
  With `radmc3d image` you call RADMC-3D with the command of performing a ray-tracing computation to compute an image. You must specify the wavelength(s) at which you want the image by, for instance, calling RADMC-3D as `radmc3d image lambda 10`, which makes the image at $\lambda = 10 \mu$m. But there are other ways by which the wavelength(s) can be set, see Section 9.4. In all these cases the vantage point (where is the observer) can of course be set as well, see Section 6.2 and Chapter 9.

- *Compute the local radiation field inside the model:*
  With `radmc3d mcmono` you call RADMC-3D with the command of performing a wavelength-by-wavlength monochromatic Monte Carlo simulation (at the wavelengths that you specify in the file `mcmono_wavelength_micron.inp`). The output file of this computation is `mean_intensity.out` which contains the mean intensity J_ν as a function of the (x, y, z) (cartesian) or (r, θ, ϕ) (spherical) coordinates at the frequencies $\nu_i \equiv 10^4 c/\lambda_i$ where λ_i are the wavelengths (in μm) specified in the file `mcmono_wavelength_micron.inp`. The results of this computation can be interesting for, for instance, models of photochemistry. But if you use RADMC-3D only for computing spectra and images, then you will not use this.

In addition to the above main methods, you can ask RADMC-3D to do various minor things as well, which will be described throughout this manual.

5.5 How a model is set up and computed: a rough overview

A radiative transfer code such as RADMC-3D has the task of computing synthetic images and spectra of a model that you specify. You tell the code what the dust and/or gas density distribution in 3-D space is and where the star(s) are, and the code will then tell you what your cloud looks like in images and/or spectra. That's basically it. That's the main task of RADMC-3D[2].

First you have to tell RADMC-3D what 3-D distribution of dust and/or gas you want it to model. For that you must specify a coordinate system (cartesian or spherical) and a spatial grid. For cartesian coordinates this grid should be 3-D (although there are exceptions to this), while for spherical coordinates it can be 1-D (spherical symmetry), 2-D (axial symmetry) or 3-D (no symmetry). RADMC-3D is (for most part) a cell-based code, i.e. your grid devides space in cells and you have to tell RADMC-3D what the average densities of dust and/or gas are in these cells.

The structure of the grid is specified in a file `amr_grid.inp` (see Section A.2). All the other data, such as dust density and/or gas density are specified in other files, but all assume that the grid is given by `amr_grid.inp`.

We can also specify the locations and properties of one or more stars in the model. This is done in the `stars.inp` (see Section A.7) file.

Now suppose we want to compute the appearance of our model in dust continuum. We will describe this in detail in Chapter 6, but let us give a very rough idea here. We write, in addition to the `amr_grid.inp` and `stars.inp` files, a file `dust_density.inp` which specifies the density of dust in each cell (see Section A.3). We also must write the main input file `radmc3d.inp` (see Section A.1), but we can leave it empty for now. We must give RADMC-3D a dust opacity table in the files `dustopac.inp` and for instance `dustkappa_silicate.inp` (see Section A.13).

[2] It can/will, of course, do much more, but that is described later in this manual.

And finally, we have to give RADMC-3D a table of discrete wavelengths in the file `wavelength_micron.inp` that it will use to perform its calculations on. We then call the `radmc3d` code with the keyword `mctherm` (see Chapter 6) to tell it to perform a Monte Carlo simulation to compute dust temperatures everywhere. RADMC-3D will write this to the file `dust_temperature.dat`. If we now want to make a spectral energy distribution, for instance, we call `radmc3d sed` (see Section 9.3) and it will write a file called `spectrum.out` which is a list of fluxes at the discrete wavelengths we specified in `wavelength_micron.inp`. Then we are done: we have computed the spectral energy distribution of our model. We could also make an image at wavelength 10 μm for instance with `radmc3d image lambda 10` (see Section 9.1). This will write out a file `image.out` containing the image data (see Section A.15).

As you see, RADMC-3D reads all its information from tables in various files. Since you don't want to make large tables by hand, you will have to write a little computer program that generates these tables automatically. You can do this in any programming language you want. But in the example models (see Section 5.7) we use the programming language IDL (see Section 3.2) for this. This is a very simple (BASIC-like) programming language that makes it easy to create the above input files. It is easiest to indeed have a look at the example models to see how this is (or better: can be) done.

We will explain all these things in much more detail below, and we will discuss also many other radiative transfer problem types. The above example is really just meant to give an impression of how RADMC-3D works.

5.6 Organization of model directories

The general philosophy of the RADMC-3D code package is the following. The core of everything is the fortran code `radmc3d`. This is the main code which does the hard work for you: it makes the radiative transfer calculations, makes images, makes spectra etc. Normally you compile this code just once-and-for-all (see Chapter 4), and then simply use the executable `radmc3d` for all models. There is an exception to this 'once-and-for-all' rule described in Section 4.5, but in the present chapter we will not use this (see Chapter 13 for this instead). So we will stick here to the philosophy of compiling this code once and using it for all models.

So how to set up a model? The trick is to present `radmc3d` with a set of input files in which the model is described in all its details. The procedure to follow is this:

1. The best thing to do (to avoid a mess) is to make a directory for *each model*: one model, one directory. Since `radmc3d` reads multiple input files, and also outputs a number of files, this is a good way to keep organized and we recommend it strongly. So if we wish to make a new model, we make a new directory, or copy an old directory to a new name (if we merely want to make small changes to a prior model).

2. In this directory we generate the input files according to their required format (see Chapter A). You can create these input files in any way you want. But since many of these input files will/must contain huge lists of numbers (for instane, giving the density at each location in your model), you will typically want to write some script or program in some language (be it either C, C++, Fortran, IDL, GDL, perl, python, you name it) that automatically creates these input files. *We recommend using IDL, because we provide examples and standard subroutines in the programming language IDL; see below for more details. But IDL is not a strict requirement for using RADMC-3D.* Section 5.7 describes how to use the example IDL scripts to make these input files with IDL.

3. When all the input files are created, and we make sure that we are inside the model directory, we call `radmc3d` with the desired command-line options (see Chapter C). This will do the work for us.

4. Once this is done, we can analyze the results by reading the output files (see Chapter A). To help you reading and analyzing these output files you can use a set of IDL routines that we created for the user (see Chapter 14 and Section 4.4). But here again, you are free to use any other plotting software and/or data postprocessing packages (many people favor python, for instance).

5.7 Running the example models

Often the fastest and easiest way to learn a code is simply to analyze and run a set of example models. They are listed in the `examples` directory. Each model occupies a separate directory. This is also the style we normally recommend: each model should have its own directory. Of course there are also exceptions to this rule, and the user is free to organize her/his data in any way he/she pleases. But in all the examples and throughout this manual each model has its own directory.

To run an example model, go into the directory of this model, and follow the directions that are written in the `README` file in each of these directories. *This is under the assumption that you have IDL installed on your system, and that you have a license for it.*

Let us do for instance `run_simple_1/`:

```
cd examples/run_simple_1
```

Now we must create all the input files for this model. These input files are all described in chapter A, but let us here just 'blindly' follow the example. In this example most (all except one) of the input files are created using an IDL script called `problem_setup.pro`. To execute this script, this is what you do:

```
idl
IDL> .r problem_setup.pro
IDL> exit
```

or in words: you go into IDL and *within the IDL prompt* you type `.r problem_setup.pro`, and if this is ready, you can leave IDL again (the latter is not required of course)[3]. This IDL script has now created a whole series of input files, all ending with the extension `.inp`. To see which files are created, type the following in the shell:

```
ls -l *.inp
```

There is one file that this example does not create, and that is the file `dustkappa_silicate.inp`. This is a file that contains the dust opacity in tabulated form. This is a file that you as the user should provide to the RADMC-3D code package. The file `dustkappa_silicate.inp` is merely an example, which is an amorphous spherical silicate grain with radius 0.1 micron. But see Section A.13 for more information about the opacities.

Now that the input files are created we must run `radmc3d`:

```
radmc3d mctherm
```

This tells RADMC-3D to do the thermal Monte Carlo simulation. This may take some time. When the model is ready, the prompt of the shell returns. To see what files have been created by this run of the code, type:

```
ls -l *.dat
```

You will find the `dust_temperature.dat` containing the dust temperature everywhere in the model. See again chapter A for details of these files. To create a spectrum:

```
radmc3d sed incl 45.
```

This will create a file `spectrum.dat`. To analyze these data you can use the IDL routines delivered with the code (see Chapter 14 and Section 4.4).

There is a file `Makefile` in the directory. This is here only meant to make it easy to clean the directory. Type `make cleanmodel` to clean all the output from the radmc3d code. Type `make cleanall` to clean the directory back to basics.

Let us now do for instance model `run_simple_1_userdef/`:

```
cd examples/run_simple_1_userdef
```

[3] Note that this does not work in IDL demo mode, which is what you get if you use IDL without a license, because files will be written, which is not possible in demo mode.

This is the same model as above, but now the grid and the dust density are set up *inside* `radmc3d`, using the file `userdef_module.f90` which is present in this directory. See Chapter 13 for details and follow the directions in the `README` file. In short: first edit the variable `SRC` in the `Makefile` to point to the `src/` directory. Then type `make`. Then go into IDL and run the `problem_setup.pro` script (which now only sets up the frequency grid, the star and the `radmc3d.inp` file and some small stuff). Now you can run the model.

Please read the README file in each of the example model directories. Everything is explained there, including how to make the relevant plots.

Chapter 6

Dust continuum radiative transfer

Many of the things related to dust continuum radiative transfer have already been said in the previous chapters. But here we combine these things, and expand with more in-depth information.

Most users simply want RADMC-3D to compute images and spectra from a model. This is done in a two-stage procedure:

1. First compute the dust temperature everywhere using the thermal Monte Carlo computation (Section 6.1).
2. Then making the images and/or spectra (Section 6.2).

You can then view the output spectra and images with the IDL tools or use your own plotting software.

Some expert users may wish to use RADMC-3D for something entirely different: to compute the local radiation field *inside* a model, and use this for e.g. computing photochemistry rates of a chemical model or so. This is described in Section 6.4.

You may also use the thermal Monte Carlo computation of the dust temperature to help estimating the *gas* temperature for the line radiative transfer. See Chapter 7 for more on line transfer.

6.1 The thermal Monte Carlo simulation: computing the dust temperature

RADMC-3D can compute the dust temperature using the Monte Carlo method of Bjorkman & Wood (2001, ApJ 554, 615) with various improvements such as the continuous absorption method of Lucy (1999, A&A 344, 282). Once a model is entirely set up, you can ask `radmc3d` to do the Monte Carlo run for you by typing in a shell:

```
radmc3d mctherm
```

if you use the standard `radmc3d` code, or

```
./radmc3d mctherm
```

if you have created a local version of `radmc3d` (see Section 4.5).

What the method does is the following: First all the netto sources of energy (or more accurately: sources of luminosity) are identified. The following net sources of energy can be included:

- *Stars:* You can specify any number of individual stars: their position, and their spectrum and luminosity (See Section A.7). This is the most commonly used source of luminosity, and as a beginning user we recommend to use only this for now.

- *Continuum stellar source:* For simulations of galaxies it would require by far too many individual stars to properly include the input of stellar light from the billions of stars in the galaxy. To overcome this problem you can specify a continuously spatially distributed source of stars. *NOTE: Still in testing phase.*
- *Viscous heating / internal heating:* Sometimes the dust grains acquire energy directly from the gas, for instance through viscous heating of the gas or adiabatic compression of the gas. This can be included as a spatially distributed source of energy. *NOTE: Still in progress... Not yet working.*

To compute the dust temperature we must have at least one source of luminosity, otherwise the equilibrium dust temperature would be everywhere 0.

The next step is that this total luminosity is divided into `nphot` packages, where `nphot` is 100000 by default, but can be set to any value by the user (see the file `radmc3d.inp` described in Section A.1). Then these photon packages are emitted by these sources one-by-one. As they move through the grid they may scatter off dust grains and thus change their direction. They may also get absorbed by the dust. If that happens, the photon package is immediately re-emitted in another direction and with another wavelength. The wavelength is chosen according to the recipe by Bjorkman & Wood (2001, ApJ 554, 615). The luminosity fraction that each photon package represents remains, however, the same. Each time a photon package enters a cell it increases the "energy" of this cell and thus increases the temperature of the dust of this cell. The recipe for this is again described by Bjorkman & Wood (2001, ApJ 554, 615), but contrary to that paper we increase the temperature of the dust always when a photon package enters a cell, while Bjorkman & Wood only increase the dust temperature if a discrete absorption event has taken place. Each photon package will ping-pong through the model and never gets lost until it escapes the model through the outer edge of the grid (which, for cartesianl coordinates, is any of the grid edges in x, y or z, and for spherical coordinates is the outer edge of r). Once it escapes, a new photon package is launched, until also it escapes. After all photon packages have been launched and escaped, the dust temperature that remains is the final answer of the dust temperature.

One must keep in mind that the temperature thus computed is an *equilibrium* dust temperature. It assumes that each dust grain acquires as much energy as it radiates away. This is for most cases presumably a very good approximation, because the heating/cooling time scales for dust grains are typically very short compared to any time-dependent dynamics of the system. But there might be situations where this may not be true: in case of rapid compression of gas, near shock waves or in extremely optically thick regions.

6.2 Making SEDs, spectra, images for dust continuum

You can use RADMC-3D for computing spectra and images in dust continuum emission. This is described in detail in Chapter 9. RADMC-3D needs to know not only the dust spatial distribution (the file `dust_density.inp`) but also the dust temperature (the file `dust_temperature.dat`). The latter is normally computed by RADMC-3D itself through the thermal Monte Carlo computation (see Section 6.1). But if you, the user, wants to specify the dust temperature at each location in the model youself, then you can simply create your own file `dust_temperature.dat` and skip the thermal Monte Carlo simulation and go straight to the creation of images or spectra.

The basic command to make a spectrum at the global grid of wavelength (specified in the file `wavelength_micron.inp`, see Section A.11) is:

```
radmc3d sed
```

You can specify the direction of the observer with `incl` and `phi`:

```
radmc3d sed incl 20 phi 80
```

which means: put the observer at inclination 20 degrees and ϕ-angle 80 degrees.

You can also make a spectrum for a given grid of wavelength (independent of the global wavelength grid). You first create a file `camera_wavelength_micron.inp`, which has the same format as `wavelength_micron.inp`. You can put any set of wavelengths in this file without modifying the global wavelength grid (which is used by the thermal Monte Carlo computation). Then you type

```
radmc3d spectrum loadlambda
```

and it will create the spectrum on this wavelength grid. More information about making spectra is given in Chapter 9.

For creating an image you can type

```
radmc3d image lambda 10
```

which creates an image at wavelength λ=10μm. More information about making images is given in Chapter 9.

Important note: To handle scattering of light off dust grains, the ray-tracing is preceded by a quick Monte Carlo run that is specially designed to compute the "scattering source function". This Monte Carlo run is usually *much* faster than the thermal Monte Carlo run, but must be done at each wavelength. It can lead, however, to slight spectral noise, because the random photon paths are different for each wavelength. See Section 6.5 for details.

6.3 Overview of input data for dust radiative transfer

In order to perform any of the actions described in Sections 6.1, 6.4 or 6.2, you must give RADMC-3D the following data:

- `amr_grid.inp`: The grid file (see Section A.2).
- `wavelength_micron.inp`: The global wavelength file (see Section A.11).
- `stars.inp`: The locations and properties of stars (see Section A.7).
- `dust_density.inp`: The spatial distribution of dust on the grid (see Section A.3).
- `dustopac.inp`: A file with overall information about the various species of dust in the model (see Section A.13). One of the main pieces of information here is (a) how many dust species are included in the model and (b) the tag names of these dust species (see `dustkappa_XXX.inp` below). The file `dust_density.inp` must contain exactly this number of density distributions: one density distribution for each dust species.
- `dustkappa_XXX.inp`: One or more dust opacity files (where `XXX` should in fact be a tag name you define, for instance `dustkappa_silicate.inp`). The labels are listed in the `dustopac.inp` file. ee Section A.13 for more information.
- `camera_wavelength_micron.inp (optional)`: This file is only needed if you want to create a spectrum at a special set of wavelengths (otherwise use `radmc3d sed`).
- `mcmono_wavelength_micron.inp (optional)`: This file is only needed if you want to compute the radiation field inside the model by calling `radmc3d mcmono` (e.g. for photochemistry).

Other input files could be required in certain cases, but you will then be asked about it by RADMC-3D.

6.4 Special-purpose feature: Computing the local radiation field

If you wish to use RADMC-3D for computing the radiation field *inside* the model, for instance for computing photochemical rates in a chemical model, then RADMC-3D can do so by calling RADMC-3D in the following way:

```
radmc3d mcmono
```

This computes the mean intensity J_ν as a function of the (x, y, z) (cartesian) or (r, θ, ϕ) (spherical) coordinates at frequencies $\nu_i \equiv 10^4 c/\lambda_i$ where λ_i are the wavelengths (in μm) specified in the file `mcmono_wavelength_micron.inp`. The results of this computation can be interesting for, for instance, models of photochemistry. But if you use RADMC-3D only for computing spectra and images, then you will not use this.

Note that if your model is very large, the computation of the radiation field on a large set of wavelength could easily overload the memory of the computer. However, often you are in the end not interested in the entire spectrum at each location, but just in integrals of this spectrum over some cross section. For instance, if you want to compute the degree to which dust shields molecular photodissociation lines in the UV, then you only need to compute the total photodissociation rate, which is an integral of the photodissociation cross section times the radiation field. In Section 13.3 it will be explained how you can create a userdef subroutine (see Chapter 13) that will do this for you in a memory-saving way.

There is an important parameter for this Monochromatic Monte Carlo that you may wish to play with:

- `nphot_mono`
  The parameter `nphot_mono` sets the number of photon packages that are used for the Monochromatic Monte Carlo simulation. It has as default 100000, but that may be too little for 3-D models. You can set this value in two ways:
  - In the `radmc3d.inp` file as a line `nphot_mono = 1000000` for instance.
  - On the command-line by adding `nphot_mono 1000000`.

6.5 More about scattering of photons off dust grains

Photons can not only be absorbed and re-emitted by dust grains: They can also be scattered. Scattering does nothing else than change the direction of propagation of a photon. Strictly speaking it may also slightly change its wavelength, if the dust grains move with considerable speed they may Doppler-shift the wavelength of the outgoing photon (which may be relevant, if at all, when dust radiative transfer is combined with line radiative transfer, see chapter 7), but this subtle effect is not treated in RADMC-3D. For RADMC-3D scattering is just the changing of direction of a photon.

6.5.1 Three modes of treating scattering

RADMC-3D has three main levels of treatment of scattering:

1. *No scattering:* If either the `dustkappa_XXX.inp` files do not contain a scattering opacity or scattering is switched off by setting `scattering_mode_max` to 0 in the `radmc3d.inp` file, then scattering is ignored. It is then assumed that the dust grains have zero albedo.
2. *Isotropic scattering:* If either the `dustkappa_XXX.inp` files do not contain information about the anisotropy of the scattering or anisotropic scattering is switched off by setting `scattering_mode_max` to 1 in the `radmc3d.inp` file, then scattering is treated as isotropic scattering. Note that this can be a bad approximation in certain cases.
3. *Full anisotropic scattering:* If the `dustkappa_XXX.inp` files contain the scattering opacity and information about the anisotropy, and `scattering_mode_max` to 2 or higher in the `radmc3d.inp` file (2 is the default, which will be used if no setting is specified of `scattering_mode_max`) then the full anisotropic scattering is treated. This is clearly the most physically correct.

So in summary: the dust opacity files themselves tell how detailed the scattering is going to be included. If no scattering information is present in these files, RADMC-3D has no choice but to ignore scattering. If they only contain scattering opacities but no phase information, then RADMC-3D will treat scattering in the isotropic approximation. Only if all scattering information is present in these input files, will RADMC-3D do the full thing. BUT even if this information is present, you can limit the realism of scattering by setting the `scattering_mode_max` to 1 or 0 in the file `radmc3d.inp`. This can be useful to speed up the calculations or be sure to avoid certain complexities of the full phase-function treatment of scattering.

At the moment there are some limitations to the full anisotropic scattering treatment:

- *Anisotropic scattering in 1-D and 2-D Spherical coordinates:*
  For 1-D and 2-D Spherical coordinates there is currently no possibility of treating anisotropic scattering in the image- and spectrum-making. The reason is that the scattering source function (see Section 6.5.4) must be stored in an angle-dependent way. However, in 2-D spherical coordinates, each cell is in fact a ring around the symmetry axis, and the angular dependence of the scattering source function would depend on which position along the ring the scattering takes place (with respect to the location of the observer). *This will be fixed hopefully later by storing the scattering source function for different angles in each cell; stay tuned!*

6.5.2 Scattering of photons in the Thermal Monte Carlo run

So how is scattering treated in practice? In the thermal Monte Carlo model (Section 6.1) the scattering has only one effect: it changes the direction of propagation of the photon packages whenever such a photon package experiences a scattering event. This may change the results for the dust temperatures subtly. In special cases it may even change the dust temperatures more strongly, for instance if scattering allows "hot" photons to reach regions that would have otherwise been in the shadow. It may also increase the optical depth of an object and thus change the temperatures accordingly. But this is all there is to it.

6.5.3 Scattering of photons in the Monochromatic Monte Carlo run

For the monochromatic Monte Carlo calculation (Section 6.4) the scattering has the same effect as for the thermal Monte Carlo model: it changes the direction of photon packages. In this way "hot" radiation may enter regions which would otherwise have been in a shadow. And by increasing the optical depth of regions, it may increase the local radiation field by the greenhouse effect or decrease it by preventing photons from entering it. As in the thermal Monte Carlo model the effect of scattering in the monochromatic Monte Carlo model is simply to change the direction of motion of the radiation field, but for the rest nothing differs to the case without scattering.

6.5.4 Scattered light in images and spectra: The "Scattering Monte Carlo" computation

For making images and spectra with the ray-tracing capabilities of RADMC-3D (see Section 6.2 and Chapter 9) the role of scattering is a much more complex one than in the thermal and monochromatic Monte Carlo runs. The reason is that the scattered radiation will eventually end up on your images and spectra. The ray-tracing transfer equation along each ray is:

$$\frac{dI_\nu}{ds} = j_\nu^{\rm therm} + j_\nu^{\rm scat} - (\alpha_\nu^{\rm abs} + \alpha_\nu^{\rm scat}) I_\nu \tag{6.1}$$

where $\alpha_\nu^{\rm abs}$ and $\alpha_\nu^{\rm scat}$ are the extinction coefficients for absorption and scattering. Let us assume, for convenience of notation, that we have just one dust species with density dstribution ρ, absorption opacity $\kappa_\nu^{\rm abs}$ and scattering opacity $\kappa_\nu^{\rm scat}$. We then have

$$\alpha_\nu^{\rm abs} \equiv \rho\kappa_\nu^{\rm abs} \tag{6.2}$$

$$\alpha_\nu^{\rm scat} \equiv \rho\kappa_\nu^{\rm scat} \tag{6.3}$$

$$j_\nu^{\rm therm} = \alpha_\nu^{\rm abs} B_\nu(T) \tag{6.4}$$

where $B_\nu(T)$ is the Planck function. The last equation is an expression of Kirchhoff's law. For isotropic scattering the scattering source function $j_\nu^{\rm scat}$ is given by

$$j_\nu^{\rm scat} = \alpha_\nu^{\rm scat} \frac{1}{4\pi} \oint I_\nu d\Omega \tag{6.5}$$

where the integral is the integral over solid angle. In this case $j_\nu^{\rm scat}$ does *not* depend on solid angle. For anisotropic scattering we must introduce the scattering phase function $\Phi(\Delta\Omega)$, where $\Delta\Omega$ is the angle between incoming and outgoing photon, and the scattering phase function is normalized to unity:

$$\frac{1}{4\pi} \oint \Phi(\Delta\Omega) = 1 \tag{6.6}$$

Then the scattering source function becomes:

$$j_\nu^{\rm scat}(\Omega') = \alpha_\nu^{\rm scat} \frac{1}{4\pi} \oint I_\nu(\Omega)\Phi(\Delta\Omega)d\Omega \tag{6.7}$$

which is angle-dependent. The angular dependence means: a photon package has not completely forgotten from which direction it came before hitting the dust grain.

If we want to make an image or a spectrum, then for each pixel we must integrate Eq. (6.1) along the 1-D ray belonging to that pixel. If we performed the thermal Monte Carlo simulation beforehand (or if we specified the dust temperatures by hand) we know the thermal source function through Eq. (6.4). But we have, at that point, no information yet about the scattering source function. The thermal Monte Carlo calculation *could* have also stored this function at each spatial point and each wavelength and each observer direction, but that would require gigantic amounts of memory (for a typical 3-D model it might be many Gbytes, going into the Tbyte regime). So in RADMC-3D the scattering source function is *not* computed during the thermal Monte Carlo run.

In RADMC-3D the scattering source function $j_\nu^{\rm scat}(\Omega')$ is computed *just prior to* the ray-tracing through a brief "Scattering Monte Carlo" run. This is done *automatically* by RADMC-3D, so you don't have to worry about this. Whenever you ask RADMC-3D to make an image (and if the scattering is in fact included in the model, see Section 6.5.1), RADMC-3D will automatically realize that it requires knowledge of $j_\nu^{\rm scat}(\Omega')$, and it will start a brief single-wavelength Monte Carlo simulation for computing $j_\nu^{\rm scat}(\Omega')$. This single-wavelength "Scattering Monte Carlo" simulation is relatively fast compared to the thermal Monte Carlo simulation, because photon packages can be destroyed by absorption. So photon packages do not bounce around for long, as they do in the thermal Monte Carlo simulation. This Scattering Monte Carlo simulation is in fact very similar to the monochromatic Monte Carlo model described in Section 6.4. While the monochromatic Monte Carlo model is called specifically by the user (by calling RADMC-3D with `radmc3d mcmono`), the Scattering Monte Carlo simulation is not something the user must specify him/her-self: it is automatically done by RADMC-3D if it is needed (which is typically before making an image or during the making of a spectrum). And while the monochromatic Monte Carlo model returns the mean intensity inside the model, the Scattering Monte Carlo simulation provides the raytracing routines with the scattering source function but does *not* store this function in a file.

You can see this happen if you have a model with scattering opacity included, and you make an image with RADMC-3D, you see that it prints `1000`, `2000`, `3000`, ... etc., in other words, it performs a little Monte Carlo simulation before making the image.

There is an important parameter for this Scattering Monte Carlo that you may wish to play with:

- `nphot_scat`
  The parameter `nphot_scat` sets the number of photon packages that are used for the Scattering Monte Carlo simulation. It has as default 100000, but that may be too little for 3-D models and/or cases where you wish to reduce the "streaky" features sometimes visible in scattered-light images when too few photon packages are used. You can set this value in two ways:
  - In the `radmc3d.inp` file as a line `nphot_scat = 1000000` for instance.
  - On the command-line by adding `nphot_scat 1000000`.
- `nphot_spec`
  The parameter `nphot_spec` is actually exactly the same as `nphot_scat`, but is used (and used only!) for the creation of spectra. The default is 10000, i.e. substantially smaller than `nphot_scat`. The reason for this separate parameter is that if you make spectra, you integrate over the image to obtain the flux (i.e. the value of the spectrum at that wavelength). Even if the scattered light image may look streaky, the integral may still be accurate. We can thus afford much fewer photon packages when we make spectra than when we make images, and can thus speed up the calculation of the spectrum. You can set this value in two ways:
  - In the `radmc3d.inp` file as a line `nphot_spec = 100000` for instance.
  - On the command-line by adding `nphot_spec 100000`.

  NOTE: It may be possible to get still very good results with even smaller values of `nphot_spec` than the default value of 10000. That might speed up the calculation of the spectrum even more in some cases. On the other hand, if you notice "noise" on your spectrum, you may want to increase `nphot_spec`. If you are

interested in an optimal balance between accuracy (high value of `nphot_spec`) and speed of calculation (low value of `nphot_spec`) then it is recommended to experiment with this value. If you want to be on the safe side, then set `nphot_spec` to a high value (i.e. set it to 100000, as `nphot_spec`).

WARNING: At wavelengths where the dominant source of photons is thermal dust emission but scattering is still important (high albedo), it cannot be excluded that the "scattering monte carlo" method used by RADMC-3D produces very large noise. Example: a very optically thick dust disk consisting of large grains (10 μm size), producing thermal dust emission in the near infrared in its inner disk regions. This thermal radiation can scatter off the large dust grains at large radii (where the disk is cold and where the only "emission" in the near-infrared is thus the scattered light) and thus reveal the outer disk in scattered light emerging from the inner disk. However, unless `nphot_scat` is huge, most thermally emitted photons from the inner disk will be emitted so deeply in the disk interior (i.e. below the surface) that they will be immediately reabsorbed and lost. This means that that radiation that does escape is extremely noisy. The corresponding scattered light source function at large radii is therefore very noisy as well, unless `nphot_scat` is taken to be huge. Currently no elegant solution is found, but maybe there will in the near future. Stay tuned...

6.5.5 Warning when using an-isotropic scattering

An important issue with anisotropic scattering is that if the phase function is very forward-peaked, then you may get problems with the *spatial* resolution of your model: it could then happen that one grid cell may be too much to the left to "beam" the scattered light into your line of sight, while the next grid point will be too much to the right. A proper treatment of strongly anisotropic scattering therefore requires also a good check of the spatial resolution of your model. There are, however, also two possible tricks (approximations) to prevent problems. They both involve slight modifications of the dust opacity files:

1. You can simply assure in the opacity files that the forward peaking of the phase function has some upper limit.
2. Or you can simply treat extremely forward-peaked scattering as no scattering at all (simply setting the scattering opacity to zero at those wavelengths).

Both "tricks" are presumably reasonable and will not affect your results, unless you concentrate in your modeling very much on the angular dependence of the scattering.

6.5.6 For experts: Some more background on scattering

The inclusion of the scattering source function in the images and spectra is a non-trivial task for RADMC-3D because of memory constraints. If we would have infinite random access memory, then the inclusion of scattering in the images and spectra would be relatively easy, as we could then store the entire scattering source function $j^{\rm scat}(x,y,z,\nu,\Omega)$ and use what we need at any time. But as you see, this function is a 6-dimensional function: three spatial dimensions, one frequency and one angular direction (which consists of two angles). For any respectable model this function is far too large to be stored. So nearly all the "numerical logistic" complexity of the treatment of scattering comes from various ways to deal with this problem. In principle RADMC-3D makes the choices of which method to use itself, so the user is not bothered with it. But depending on which kind of model the user sets up, the performance of RADMC-3D may change as a result of this issue.

So here are a few hints as to the internal workings of RADMC-3D in this regard. You do not have to read this, but it may help understanding the performance of RADMC-3D in various cases.

- *Scattering in spectra and multi-wavelength images*
  If no scattering is present in the model (see Section 6.5.1), then RADMC-3D can save time when making spectra and/or multi-wavelength images. I will then do each integration of Eq. (6.1) directly for all wavelengths at once before going to the next pixel. This saves some time because RADMC-3D then has to calculate the geometric stuff (how the ray moves through the model) just once for each ray. If, however, scattering is included, the scattering source function must be computed using the Scattering Monte Carlo computation. Since for large models it would be too memory consuming (in particular for 3-D models) to store this function for all positions *and* all wavelengths, it must do this calculation one-by-one for each wavelength, and calculate the

image for that wavelength, and then go off to the next wavelength. This means that for each ray (pixel) the geometric computations (where the ray moves through the model) has to be redone for each new wavelength. This may slow down the code a bit.

- *Anisotropic scattering and multi-viewpoint images*
  Suppose we wish to look at an object at one single wavelength, but from a number of different vantage points. If we have *isotropic* scattering, then we need to do the Scattering Monte Carlo calculation just once, and we can make multiple images at different vantage points with the same scattering source function. This saves time, if you use the "movie" mode of RADMC-3D (Section 9.11). However, if the scattering is anisotropic, then the source function would differ for each vantage point. In that case the scattering source function must be recalculated for each vantage point. There is, deeply hidden in RADMC-3D, a way to compute scattering source functions for multiple vantage points within a single Scattering Monte Carlo run, but for the moment this is not yet activated.

6.6 Polarization, Stokes vectors and full phase-functions

As of version 0.26 RADMC-3D can deal with polarization of continuum radiation. Radiative transfer of polarized radiation is a relatively complex issue. A good and extensive review on the details of polarization is given in the book by Mishchenko, Travis & Lacis, "Scattering, Absorption and Emission of Light by Small Particles", 2002, Cambridge University Press (also electronically available on-line). For some discussions on how polarization can be built in into radiative transfer codes, see e.g. Wolf, Voshchinnikov & Henning (2002, A&A 385, 365).

When we wish to include polarization in our model we must follow not just the intensity I of light (or equivalently, the energy E of a photon package), but the full Stokes vector (I, Q, U, V) (see review above for definitions, or any textbook on radiation processes). If a photon scatters off a dust grain, then the scattering angular probability density function depends not only on the scattering angle μ, but also on the input state of polarization, i.e. the values of (I, Q, U, V). And the output polarization state will be modified. In addition to this, for some circumstances the thermally emitted light is already polarized to begin with, and differently polarized radiation will be absorbed with different extinction coefficients. Moreover, even if we would not be interested in polarization at all, but we *do* want to have a correct scattering phase function, we need to treat polarization, because a first scattering wil polarize the photon, which will then have different angular scattering probability in the next scattering event. Normally these effects are very small, so if we are not particularly interested in polarization, one can usually ignore this effect without too high a penalty in reliability. But if one wants to be accurate, there is no way around a full treatment of the (I, Q, U, V), even if the end-result polarization state is of no particular interest to the user of the code.

Interaction between polarized radiation with matter happens through so-called Müller matrices, which are 4×4 matrices that can be multiplied by the (I, Q, U, V) vector. More on this later.

It is important to distinguish between two situations:

1. The simplest case (and fortunately applicable in many cases) is if all dust particles are **randomly oriented**, and there is **no preferential helicity** of the dust grains (i.e. for each particle shape there are equal numbers of particles with that shape and with its mirror copy shape). This is also automatically true if all grains are spherically symmetric. In this case the problem of polarized radiative transfer simplifies in several ways:

   - The scattering Müller matrix simplifies, and contains only 6 independent matrix elements. Moreover, these matrix elements depend only on a single angle: the scattering angle θ, and of course on the wavelength. This means that the amount of information is small enough that these Müller matrix elements can be stored in computer memory in tabulated form.
   - The total scattering cross section is independent of the input polarization state. Only the angular dependence (i.e. in which direction the photon will scatter) depends on the input polarization state.
   - The absorption cross section is the same for all components of the (I, Q, U, V)-vector. In other words: the absorption Müller matrix is the usual scalar absorption coefficient times the unit matrix.

   The last two points assure that most of the structure of the RADMC-3D code for non-polarized radiation can remain untouched. Only for computing the new direction and polarization state of a photon after a scattering event in the Monte Carlo module, as well as for computing the scattering source function in the Monte Carlo

module (for use in the camera module) we must do extra work. Thermal emission and thermal absorption remain the same, and computing optical depths remains also the same.

2. A (much!) more complex situation arises if dust grains are **non-spherical** and are somehow **aligned due to external forces**. For instance, particles tend to align themselves in the interstellar medium if strong enough magnetic fields are present. Or particles tend to align themselves due to the combination of gravity and friction if they are in a planetary/stellar atmosphere. Here are the ways in which things become more complex:

   - All the scattering Müller matrix components will become non-zero and independent. We will thus get 16 independent variables.
   - The matrix elements will depend on four angles, of which one can, in some cases, be removed due to symmetry (e.g. if we have gravity, there is still a remaining rotational symmetry; same is true of particles are aligned by a $\vec{B}$-field; but if both gravity and a $\vec{B}$-field are present, this symmetry may get lost). It will in most practical circumstances not be possible to precalculate the scattering Müller matrix beforehand and tabulate it, because there are too many variables. The matrix must be computed on-the-fly.
   - The total scattering cross section now *does* depend on the polarization state of the input photon, and on the incidence angle. This means that scattering extinction becomes anisotropic.
   - Thermal emission and absorption extinction will also no longer be isotropic. Moreover, they are no longer scalar: they are described by a non-trivial Müller matrix.

   The complexity of this case is rather large, and a proper treatment requires substantial departures of the standard structure of the scalar radiative transfer method in RADMC-3D. In particular this requires drastic changes in the Monte Carlo module. We may, at some point, include this in its full glory in RADMC-3D, but for now we will allow aligned grains only for the computation of thermal emission in images/spectra in the camera module. *And for the moment, in version 0.26, we are still working on the implementation of this.*

THIS SECTION MUST BE STILL MUCH EXPANDED

Chapter 7

Line radiative transfer

RADMC-3D is capable of modeling radiative transfer in molecular and/or atomic lines. Due to the complexity of line radiative transfer, and the huge computational and memory requirements of full-scale non-LTE line transfer, RADMC-3D has various different modes of line transfer. Some modes are very memory efficient, but slower, while others are faster, but less memory efficient, yet others are more accurate but much slower and memory demanding. The default mode (and certainly recommended initially) is LTE ray-tracing in the slow but memory efficient way: the *simple LTE mode* (see Section 7.2). Since this is the default mode, you do not need to specify anything to have this selected.

7.1 Quick start for adding line transfer to images and spectra

Do properly model line transfer requires dedication and experimentation. This is *not* a simple task. See Section 7.5 for an analysis of several pitfalls one may encounter. However, nothing is better than experimenting and thus gaining hands-on experience. So the easiest and quickest way to start is to start with one of the simple line transfer test models in the `examples/` directory.

So simply visit `examples/run_test_lines_1/`, `examples/run_test_lines_2/` or `examples/run_test_lines_3/` and follow the directions in the `README` file. The main features of adding line ray tracing to a model is to add the following files into any previously constructed model with dust radiative transfer:

- `lines.inp`: A control file for line transfer.
- `molecule_co.inp`: or any other molecular data file containing properties of the molecule or atom.
- `gas_temperature.inp`: The gas temperature at each grid cell. You do not need to specify this file if you add the keyword `tgas_eq_tdust = 1` into the `radmc3d.inp` file.

and then start the `viewimage` viewer (see Section 14.3) with keyword "`/lines`". Or you can use the `makeimage` or `doimage` routines from the `readradmc.pro`.

7.2 Line transfer modes and how to activate the line transfer

Here is a list of the various modes for line transfer:

1. *Simple LTE mode (=default mode):* In this mode the line radiative transfer is done under LTE assumptions. The level populations will be calculated on-the-fly while doing the ray-tracing. It is therefore cheap in memory (the level populations do not have to be stored), but slower than the *fast LTE mode*, as the populations continuously have to be re-calculated on the fly. The *simple LTE mode* is default. *NOTE: The simple LTE mode allows the use of "line lists" instead of full molecular data input files. See Section 7.3.3.*

2. *Fast LTE mode:* This is like *simple LTE mode*, but here the LTE populations are pre-computed. This requires some more memory, but can be done still quite efficiently if a subset of levels is chosen in the `lines.inp` file, in particular if for a particular line just the upper and lower level is chosen.
3. *Simple LVG mode:* **IN PROGRESS!**
4. *Fast LVG mode:* **Not yet ready**
5. *Full non-LTE modes:* **Not yet ready**

Which model of line transfer is used is specified in the radmc3d.inp file. If no option is given, then the *simple LTE mode* is used. For each of the modes (including the default one) there is a switch that can be set to 1 to select that mode:

- `simplelte=1` selects the *simple LTE mode* (default, so you do not need to set this).
- `fastlte=1` selects the *fast LTE mode*.

NOTE 1: Line emission is automatically included in the images and spectra if RADMC-3D finds the file `lines.inp` in the model directory. You can switch off the lines with the command-line option `'noline'`.

NOTE 2: The `viewimage.pro` image viewer also automatically includes line emission. But you would have to seek the precise wavelength of the lines yourself. If, however, you call `viewimage` with option `/lines`, then some extras appear that allow you to directly find the right wavelength of the lines. Try it out, and you will see how it works.

7.3 The various input files for line transfer

7.3.1 INPUT: The line transfer entries in the radmc3d.inp file

Like all other modules of `radmc3d`, also the line module can be steered through keywords in the `radmc3d.inp` file. Here is a list:

- `tgas_eq_tdust` (default: 0)
  Normally you must specify the gas temperature at each grid cell using the `gas_temperature.inp` file (or directly in the `userdef_module.f90`, see Chapter 13). But sometimes you may want to compute first the dust temperature and then set the gas temperature equal to the dust temperature. You can do this obviously by hand: read the output dust temperature and create the equivalent gas temperature input file from it. But that is cumbersome. By setting `tgas_eq_tdust=1` you tell `radmc3d` to simply read the `dust_temperature.inp` file and then equate the gas temperature to the dust temperature. If multiple dust species are present, only the first species will be used.

7.3.2 INPUT: The line.inp file

Like with the dust (which has this `dustopac.inp` master file, also the line module has a master file: `lines.inp`. It specifies which molecules/atoms are to be modeled and in which file the molecular/atomic data (such as the energy levels and the Einstein A coefficients) are to be found.

```
iformat                           <=== Put this to 1
N                                 Nr of molecular or atomic species to be modeled
molname1 inpstyle1 iduma1 idumb1  Which molecule used as species 1, where to read it?
.
.
.
molnameN inpstyleN idumaN idumbN  Which molecule used as species N, where to read it?
```

The `N` is the number of molecular or atomic species you wish to model. Typically this is 1. But if you want to *simultaneously* model for instance the ortho-H_2O and para-H_2O infrared lines, you would need to set this to 2.

The N lines following N (i.e. lines 3 to N+2) specify the molecule or atom, the kind of input file format (explained below), and two integers which, at least for now, can be simply set to 0 (see Section 7.7 for the meaning of these integers - for experts only).

The molecule name can be e.g. `co` for carbon monoxide. The file containing the data should then be called `molecule_co.inp` (even if it is an atom rather than a molecule; I could not find a good name which means both molecule or atom). This file should be either generated by the user, or (which is obviously the preferred option) taken from one of the databases of molecular/atomic radiative properties. Since there are a number of such databases and I want the code to be able to read those files without the need of casting them into some special RADMC-3D format, `radmc3d` allows the user to select which *kind* of file the `molecule_co.inp` (for CO) file is. At present only one format is supported: the Leiden database. But more will follow. To specify to `radmc3d` to use the Leiden style, you put the `inpstyle` to "leiden". So here is a typical example of a `lines.inp` file:

```
1
1
co   leiden   0   0
```

This means: one molecule will be modeled, namely CO (and thus read from the file `molecule_co.inp`), and the data format is the Leiden database format.

7.3.3 INPUT: The molecule_XXX.inp file

As mentioned in Section 7.3.2 the atomic or molecular fundamental data such as the level diagram and the radiative decay rates (Einstein A coefficients) are read from a file (or more than one files) named `molecule_XXX.inp`, where the `XXX` is to be replaced by the name of the molecule or atom in question. In the `lines.inp` you can specify which style this file has. Currently the following input style is supported: the Leiden database. More will follow.

INPUT: The Leiden database format of molecule_XXX.inp

The precise format of the Leiden database data files is of course described in detail on their web page[1]. Here we only give a very brief overview, based on an example of CO in which only the first few levels are specified:

```
#leiden
!MOLECULE
CO
!MOLECULAR WEIGHT
28.0
!NUMBER OF ENERGY LEVELS
10
!LEVEL + ENERGIES(cm^-1) + WEIGHT + J
    1     0.000000000  1.0  0
    2     3.845033413  3.0  1
    3    11.534919938  5.0  2
    4    23.069512649  7.0  3
    5    38.448164669  9.0  4
    6    57.670329083 11.0  5
    7    80.735459105 13.0  6
    8   107.642407981 15.0  7
    9   138.390328288 17.0  8
   10   172.978074417 19.0  9
!NUMBER OF RADIATIVE TRANSITIONS
9
!TRANS + UP + LOW + EINSTEINA(s^-1) + FREQ(GHz) + E_u(K)
    1     2     1   7.203e-08     115.2712018      5.53
    2     3     2   6.910e-07     230.5380000     16.60
    3     4     3   2.497e-06     345.7959899     33.19
    4     5     4   6.126e-06     461.0407682     55.32
    5     6     5   1.221e-05     576.2679305     82.97
    6     7     6   2.137e-05     691.4730763    116.16
    7     8     7   3.422e-05     806.6518060    154.87
    8     9     8   5.134e-05     921.7997000    199.11
    9    10     9   7.330e-05    1036.9123930    248.88
!BELOW CAN FOLLOW MORE DATA, FOR INSTANCE COLLISION RATE DATA
```

[1] http://www.strw.leidenuniv.nl/∼moldata/

```
!WHICH ARE USEFUL FOR NON-LTE STUFF. BUT PLEASE REFER TO THE
!LEIDEN DATABASE INFORMATION ABOUT THAT.
```

The very first line is optional: it shows that we are dealing with a file from the leiden database.

The next first few lines are self-explanatory. The first of the two tables is about the levels. Column one is simply a numbering. Column 2 is the energy of the level E_k, specified in units of $1/\text{cm}$. To get the energy in erg you multiply this number with hc/k where h is the Planck constant, c the light speed and k the Boltzmann constant. Column 3 is the degeneration number, i.e. the the g parameter of the level. Column 4 is redundant information, not used by the code.

The second table is the line list. Column 1 is again a simple counter. Column 2 and 3 specify which two levels the line connects. Column 4 is the radiative decay rate in units of $1/\text{seconds}$, i.e. the Einstein A coefficient. The last two columns are redundant information that can be easily derived from the other information.

7.3.4 INPUT: The number density of each molecular species

For the line radiative transfer we need to know how many molecules of each species are there per cubic centimeter. For molecular/atom species `XXX` this is given in the file `numberdens_XXX.inp` (formatted) or `numberdens_XXX.uinp` (fortran-style unformatted). For each molecular/atomic species listed in the `lines.inp` file there must be a corresponding `numberdens_XXX.inp` or `numberdens_XXX.uinp` file. The structure of the file is very similar (though not identical) to the structure of the dust density input file `dust_density.inp` (Section A.3). For the precise way to address the various cells in the different AMR modes, we refer to Section A.3, where this is described in detail.

For formatted style (`numerdens_XXX.inp`):

```
iformat                                  <=== Typically 1 at present
nrcells
numberdensity[1]
..
numberdensity[nrcells]
```

The number densities are to be specified in units of molecule per cubic centimeter.

For unformatted input we have again a very similar structure as with `dust_density.uinp` (see Chapter B for more details on unformatted I/O)

```
iformat       reclen
nrcells
numberdensity[1]        ... numberdensity[reclen]
numberdensity[reclen+1] ... numberdensity[2*reclen]
..
..... numberdensity[nrcells] ... 0   0   0     <==== fill with 0 until end of record
```

All integers (`iformat`, `reclen` and `nrcells` are 8-byte integers. The data of the number density is stored as series of double-precision (8-byte) reals organized in records of `reclen`/8 numbers long. If the last double precision number ends before the end of its record, then the remainder of the record is filled with 0 until the end (see example above). For details on this unformatted structure, please read the last part of Section A.3.

7.3.5 INPUT: The gas temperature

For line transfer we need to know the gas temperature. You specify this in the file `gas_temperature.inp` (formatted) or `gas_temperature.uinp` (fortran-style unformatted). The structure of this file is identical to that described in Section 7.3.4, but of course with number density replaced by gas temperature in Kelvin. For the precise way to address the various cells in the different AMR modes, we refer to Section A.3, where this is described in detail.

Note: see Chapter B for more details on unformatted I/O.

Note: Instead of literally specifying the gas temperature you can also tell `radmc3d` to copy the dust temperature (if it know it) into the gas temperature. See the keyword `tgas_eq_tdust` described in Section 7.3.1.

7.3.6 INPUT: The velocity field

Since gas motions are usually the main source of Doppler shift or broadening in astrophysical settings, it is obligatory to specify the gas velocity. This can be done with the file `gas_velocity.inp` (formatted) or `gas_velocity.uinp` (unformatted) or `gas_velocity.usinp` (unformatted, single precision). The structure is again similar to that described in Section 7.3.4, but now with three numbers at each grid point instead of just one. The three numbers are the velocity in x, y and z direction for Cartesian coordinates, or in r, θ and ϕ direction for spherical coordinates. Note that both in cartesian coordinates and in spherical coordinates *all* velocity components have the same dimension of cm/s. For spherical coordinates the conventions are: positive v_r points outwards, positive v_θ points downward (toward larger θ) for $0 < \theta < \pi$ (where "downward" is toward smaller z), and positive v_ϕ means velocity in counter-clockwise direction in the x, y-plane.

For the precise way to address the various cells in the different AMR modes, we refer to Section A.3, where this is described in detail.

The unformatted style is similar in structure as that of `dust_density.uinp`, but now with three numbers at each grid point. See Chapter B for more details on unformatted I/O.

7.3.7 INPUT: The local microturbulent broadening (optional)

The `radmc3d` code automatically includes thermal broadening of the line. But sometimes it is also useful to specify a local (spatially unresolved) turbulent width. This is not obligatory (if it is not specified, only the thermal broadening is used) but if you want to specify it, you can do so in the file `microturbulence.inp` (formatted) or `microturbulence.uinp` (unformatted). Same structure as described in Section 7.3.4. For the precise way to address the various cells in the different AMR modes, we refer to Section A.3, where this is described in detail.

The unformatted style is similar in structure as that of `dust_density.uinp`. See Chapter B for more details on unformatted I/O.

7.3.8 INPUT for LTE line transfer: The partition function (optional)

If you use the LTE mode (either *simple LTE* or *fast LTE*), then the partition function is required to calculate, for a given temperature the populations of the various levels. Since this involves a summation over *all* levels of all kinds that can possibly be populated, and since the molecular/atomic data file may not include all these possible levels, it may be useful to look the partition function up in some literature and give this to `radmc3d`. This can be done with the file `partitionfunction_XXX.inp`, where again `XXX` is here a placeholder for the actual name of the molecule at hand. If you do not have this file in the present model directory, then `radmc3d` will compute the partition function itself, but based on the (limited) set of levels given in the molecular data file. The structure of the `partitionfunction_XXX.inp` file is:

```
iformat                   ; The usual format number, currently 1
ntemp                     ; The number of temperatures at which it is specified
temp(1)       pfunc(1)
temp(2)       pfunc(2)
  .             .
  .             .
  .             .
temp(ntemp)   pfunc(ntemp)
```

7.4 Making images and spectra with line transfer

Making images and spectra with/of lines works in the same way as for the continuum. RADMC-3D will check if the file `lines.inp` is present in your directory, and if so, it will automatically switch on the line transfer. If you insist on *not* having the lines switched on, in spite of the presence of the `lines.inp` file, you can add the option `noline` to `radmc3d` on the command line. If you don't, then lines are normally automatically switched on, except in situations where it is obviously not required.

You can just make an image at some wavelength and you'll get the image with any line emission included if it is there. For instance, if you have the molecular data of CO included, then

```
radmc3d image lambda 2600.757
```

will give an image right at the CO 1-0 line center. The code will automatically check if (and if yes, which) line(s) are contributing to the wavelength of interest. Also it will include all the continuum emission (and absorption) that you would usually obtain.

There is, however, an exception to this automatic line inclusion: If you make a spectral energy distribution (with the command `sed`, see Section 9.3), then lines are not included. The same is true if you use the `loadcolor` command. But for normal spectra or images the line emission will automatically be included. So if you make a spectrum at wavelength around some line, you will get a spectrum including the line profile from the object, as well as the dust continuum.

It is not always convenient to have to know by heart the exact wavelengths of the lines you are interested in. So RADMC-3D allows you to specify the wavelength by specifying which line of which molecule, and at which velocity you want to render:

```
radmc3d image iline 2 vkms 2.4
```

If you have CO as your molecule, then iline 2 means CO 2-1 (the second line in the rotational ladder).

By default the first molecule is used (if you have more than one molecule), but you can also specify another one:

```
radmc3d image imolspec 2 iline 2 vkms 2.4
```

which would select the second molecule instead of the first one.

If you wish to make an entire spectrum of the line, you can do for instance:

```
radmc3d spectrum iline 1 widthkms 10
```

which produces a spectrum of the line with a passband going from -10 km/s to +10 km/s. By default 40 wavelength points are used, and they are evenly spaced. You can set this number of wavelengths:

```
radmc3d spectrum iline 1 widthkms 10 linenlam 100
```

which would make a spectrum with 100 wavelength points, evenly spaced around the line center. You can also shift the passband center:

```
radmc3d spectrum iline 1 widthkms 10 linenlam 100 vkms -10
```

which would make the wavelength grid 10 kms shifted in short direction.

For more details on how to specify the spectral sampling, please read Section 9.4. Note that keywords such as `incl`, `phi`, and any other keywords specifying the camera position, zooming factor etc, can all be used in addition to the above keywords.

7.4.1 Speed versus realism of rendering of line images/spectra

As usual with numerical modeling: including realism to the modeling goes at the cost of rendering speed. A "fully realistic" rendering of a model spectrum or image of a gas line involves (assuming the level populations are already known):

1. Doppler-shifted emission and absorption.
2. Inclusion of dust thermal emission and dust extinction while rendering the lines.
3. Continuum emission scattered by dust into the line-of-sight

4. Line emission from (possibly obscured) other regions is allowed to scatter into the line-of-sight by dust grains (see Section 7.4.2).

RADMC-3D always includes the Doppler shifts. By default, RADMC-3D also includes dust thermal emission and extinction, as well as the scattered continuum radiation. The scattering of line emission into the line of sight can only be done if the level populations are pre-computed (the *fast* mode, see Section **XXXX**), so it is by default not included (since by default RADMC-3D renders lines in the memory-saving *slow* mode).

If we use the memory-saving *slow* mode of line rendering, then the default inclusion of scattererd continuum light *can slow down the rendering enormously, if spectra or multi-wavelength images are made!* The reason is that, again for saving memory, the scattering Monte Carlo simulation (see Section 6.5.4) will be done once for each wavelength, and the different wavelength images are rendered one at a time (each preceded by a scattering Monte Carlo simulation). At each image rendering, the local level populations along the ray have to be re-computed (because in the memory-saving *slow* mode they are not stored). This can be very time-consuming.

For many lines, however, dust continuum scattering is a negligible portion of the flux, so you can speed things up by not including dust scattering! This can be easily done by adding the `noscat` option on the command-line when you issue the command for a line spectrum or multi-frequency image. This way, the scattering source function is not computed (is assumed to be zero), and no scattering Monte Carlo runs are necessary. This means that the ray-tracer can now render all wavelength simultaneously (each ray doing all wavelength at the same time), and the local level populations along each ray can now be computed once, and be used for all wavelengths. *This may speed up things drastically, and for most purposes virtually perfectly correct.* Just beware that when you render short-wavelength lines (optical) or you use large grains, i.e. when the scattering albedo at the wavelength of the line is not negligible, this may result in a mis-estimation of the continuum around the line.

7.4.2 Line emission scattered off dust grains

NOTE: The contents of this subsection may not be 100% implemented yet.

Also any line emission from obscured regions that get scattered into the line of sight by the dust (if dust scattering is included) will be included. Note, however, that any possible Doppler shift *induced* by this scattering is *not* included. This means that if line emission is scattered by a dust cloud moving at a very large speed, then this line emission will be scattered by the dust, but no Doppler shift at the projected velocity of the dust will be added. Only the Doppler shift of the line-emitting region is accounted for. This is rarely a problem, because typically the dust that may scatter line emission is located far away from the source of line emission and moves at substantially lower speed.

7.5 What can go wrong with line transfer?

Even the simple task of performing a ray-tracing line transfer calculation with given level populations (i.e. the so-called *formal transfer equation*) is a non-trivial task in complex 3-D AMR models with possibly highly supersonic motions. I recommend the user to do extensive and critical experimentation with the code and make many simple tests to check if the results are as they are expected to be. In the end a result must be understandable in terms of simple argumentation. If weird effects show up, please do some detective work until you understand why they show up, i.e. that they are either a *real* effect or a numerical issue. There are many numerical artifacts that can show up that are *not* a bug in the code. The code simply does a numerical integration of the equations on some spatial- and wavelength-grid. If the user chooses these grids unwisely, the results may be completely wrong even if the code is formally OK. These possible pitfalls is what this section is about.

So here is a list of things to check:

1. Make sure that the line(s) you want to model are indeed in the molecular data file you use. Also make sure that it/they are included in the line selection (if you are using this option; by default all lines and levels from the molecular/atomic data files are included; see Section 7.7).

2. If you do LTE line transfer, and you do not let `radmc3d` read in a special file for the partition function, then the partition function will be computed internally by `radmc3d`. The code will do so based on the levels specified

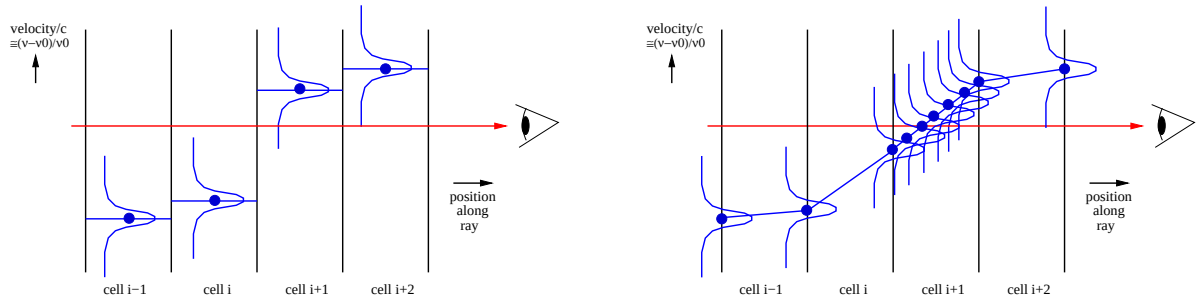


Figure 7.1. Left: Pictographic representation of the doppler jumping problem with ray-tracing through a model with strong cell-to-cell velocity differences. Right: Pictographic representation of the doppler catching method to prevent this problem: First of all, second order integration is done instead of first order. Secondly, the method automatically detects a possibly dangerous doppler jump and makes sub-steps to neatly integrate over the line that shifts in- and out of the wavelength channel of interest.

in the `molecule_XXX.inp` file for molecule `XXX`. This requires of course that all levels that may be excited at the temperatures found in the model are in fact present in the `molecule_XXX.inp` file. If, for instance, you model 1.3 mm and 2.6 mm rotational lines of CO gas of up to 300 K, and your file `molecule_co.inp` only contains the first three levels because you think you only need those for your 1.3 and 2.6 mm lines, and you *don't* specify the partition function explicitly, then `radmc3d` will compute the partition function for all temperatures including 300 K based on only the first three levels. This is evidently wrong. The nasty thing is: the resulting lines won't be totally absurd. They will just be too bright. But this can easily go undetected by you as the user. So please keep this always in mind. Note that if you make a *selection* of the first three levels (see Section 7.7) but the file `molecule_XXX.inp` contains many more levels, then this problem will not appear, because the partition function will be calculated on the original data from the `molecule_XXX.inp` file, not from the selected levels. Of course it is safer to specify the true partition function directly through the file `partitionfunction_XXX.inp` (see Section 7.3.8).

3. If you have a model with non-zero gas velocities, and if these gas velocities have cell-to-cell differences that are larger than or equal to the intrinsic (thermal+microturbulent) line width, then the ray-tracing will not be able to pick up signals from intermediate velocities. In other words, because of the discrete gridding of the model, only discrete velocities are present, which can cause numerical problems. See Fig. 7.1-Left for a pictographic representation of this problem. There are two possible solutions. One is the wavelength band method described in Section 9.5. But a more systematic method is the "doppler catching" method described in Section 7.6 (which can be combined with the wavelength band method of Section 9.5 to make it even more perfect).

7.6 Preventing doppler jumps: The "doppler catching method"

If the local co-moving line width of a line (due to thermal/fundamental broadning and/or local subgrid "microturbulence") is much smaller than the typical velocity fields in the model, then a dangerous situation can occur. This can happen if the co-moving line width is narrower than the doppler shift between two adjacent cells. When a ray is traced, in one cell the line can then have a doppler shift substantially to the blue of the wavelength-of-sight, while in the next cell the line suddenly shifted to the red side. If the intrinsic (= thermal + microturbulent) line width is smaller than these shifts, neither cell gives a contribution to the emission in the ray. See Fig. 7.1-Left for a pictographic representation of this problem. In reality the doppler shift between these two cells would be smooth, and thus the line would smoothly pass over the wavelength-of-sight, and thus make a contribution. Therefore the numerical integration may thus go wrong.

The problem is described in more detail in Section 9.5, and one possible solution is proposed there. But that solution does not always solve the problem.

RADMC-3D has a special method to catch situations like the above, and when it detects one, to make sub-steps in the integration of the formal transfer equation so that the smooth passing of the line through the wavelength-of-sight can be properly accounted for. Here this is called "doppler catching", for lack of a better name. The technique was discussed in great detail in Pontoppidan et al. (2009, ApJ 704, 1482). The idea is that the method automatically tests

if a line might "doppler jump" over the current wavelength channel. If so, it will insert substeps in the integration at the location where this danger is present. See Fig. 7.1-Left for a pictographic representation of this method. Note that this method can only be used with the second order ray-tracing (see Section 9.8); in fact, as soon as you switch the doppler catching on, RADMC-3D will automatically also switch on the second order ray-tracing.

To switch on doppler catching, you simply add the command-line option `doppcatch` to the image or spectrum command. For instance:

```
radmc3d spectrum iline 1 widthkms 10 doppcatch
```

(again: you do not need to add `secondorder`, because it is automatic when `doppcatch` is used).

The Doppler catching method will assure that the line is integrated over with small enough steps that it cannot accidently get jumped over. How fine these steps will be can be adjusted with the `catch_doppler_resolution` keyword in the `radmc3d.inp` file. The default value is 0.2, meaning that it will make the integration steps small enough that the doppler shift over each step is not more than 0.2 times the local intrinsic (thermal+microturbulent) line width. That is usually enough, but for some problems it might be important to ensure that smaller steps are taken. By adding a line

```
catch_doppler_resolution = 0.05
```

to the `radmc3d.inp` file you will ensure that steps are small enough that the doppler shift is at most 0.05 times the local line width.

So why is doppler catching an *option*, i.e. why would this not be standard? The reason is that doppler catching requires second order integration, which requires RADMC-3D to first map all the cell-based quantities to the cell-corners. This requires extra memory, which for very large models can be problematic. It also requires more CPU time to calculate images/spectra with second order integration. So if you do not need it, i.e. if your velocity gradients are not very steep compared to the intrinsic line width, then it saves time and memory to not use doppler catching.

It is, however, important to realize that doppler catching is not the golden bullet. Even with doppler catching it might happen that some line flux is lost, but this time as a result of too low *image resolution*. This is less likely to happen in problems like ISM turbulence, but it is pretty likely to happen in models of rotating disks. Suppose we have a very thin local line width (i.e. low gas temperature and no microturbulence) in a rotating thin disk around a star. In a given velocity channel (i.e. at a given observer-frame frequency) a molecular line in the disk emits only in a very thin "ear-shaped" ring or band in the image. The thinner the intrinsic line width, the thinner the band on the image. See Pontoppidan et al. (2009, ApJ 704, 1482) and Pavlyuchenkov et al. (2007, ApJ 669, 1262) for example. If the pixel-resolution of the image is smaller than that of this band, the image is simply underresolved. This has nothing to do with the doppler jumping problem, but can be equally devastating for the results if the user is unaware of this. There appears to be only one proper solution: assure that the pixel-resolution of the image is sufficiently fine for the problem at hand. This is easy to find out: The image would simply look terribly noisy if the resolution is insufficient. However, if you are not interested in the images, but only in the spectra, then some amount of noisiness in the image (i.e. marginally sufficient resolution) is OK, since the total flux is an integral over the entire image, smearing out much of the noise. It requires some experimentation, though.

Here are some additional issues to keep in mind:

- The doppler catching method uses second order integration (see Section 9.8), and therefore all the relevant quantities first have to be interpolated from the cell centers to the cell corners. Well inside the computational domain this amounts to linear interpolation. But at the edges of the domain it would require *extra*polation[2] RADMC-3D does does not do extrapolation but simply takes the average values of the nearest cells. Also the gas velocity is treated like this. This means that over the edge cells the gradient in the gas velocity tends to be (near) 0. Since for the doppler catching it is the gradient of the velocity that matters, this might yield some artifacts in the spectrum if the density in the border cells is high enough to produce substantial line emission. Avoiding this numerical artifact is relatively easy: One should then simply put the number density of the molecule in question to zero in the boundary cells.
- If you are using RADMC-3D on a 3-D (M)HD model which has strong shocks in its domain, then one must

[2] In 1-D this is more easily illustrated, because there the cell corners are in fact cell interfaces. Cells i and $i+1$ share cell interface $i+1/2$. If we have N cells, i.e. cells $i = 1, \cdots, N$, then we have $N+1$ interfaces, i.e. interfaces $i = \frac{1}{2}, \cdots, N+\frac{1}{2}$. To get physical quantities from the cell centers to cell interfaces $i = \frac{3}{2}, \cdots, N-\frac{1}{2}$ requires just interpolation. But to find the physical quantities at cell interfaces $i = \frac{1}{2}$ and $i = N+\frac{1}{2}$ one has to extrapolate or simply take the values at the cell centers $i = 1$ and $i = N$.

be careful that (magneto-)hydrodynamic codes tend to smear out the shock a bit. This means that there will be some cells that have intermediate density and velocity in the smeared out region of the shock. This is unphysical, but an intrinsic numerical artifact of numerical hydrodynamics codes. This might, under some conditions, lead to unphysical signal in the spectrum, because there would be cells at densities, temperatures and velocities that would be in between the values at both sides of the shock and would, in reality, not be there. It is very difficult to avoid this problem, and even to find out if this problem is occurring and by how much. One must simply be very careful of models containing strong shocks and do lots of testing. One way to test is to use the doppler catching method and vary the doppler catching resolution (using the `catch_doppler_resolution` keyword in `radmc3d.inp`).

7.7 For experts: Selecting a subset of lines

If you use standard molecular/atomic data files from e.g. the Leiden database[3], then you may have many more lines than you are actually interested in. This is no problem if you use the *simple LTE mode* (see Section 7.2) because then `radmc3d` will anyway only work with those lines that happen to be close to the wavelength of interest. But if you use other modes of line transfer, the level populations may need to be stored into memory. If you have a large spatial grid, and for each grid point you have to store a 100 level populations, you can easily run out of memory. For non-LTE modes (such as the LVG or escape probability modes that will be implemented later) it may be unavoidable to store these, but if you already know in advance that temperatures will never be high enough to populate the levels above, say, 20, then you may want to be able to tell `radmc3d` not to use all 100 levels, but only the first 20. One can do this of course by editing the molecular data file by hand and simply strip all the levels and lines you don't need. But a more elegant way is to specify in the `lines.inp` file that `radmc3d.inp` should use only a subset of the levels.

The way to do this is to replace, in the example `line.inp` file of Section 7.3.2, the line

```
co   leiden   0  0
```

with

```
co   leiden   0  10
```

where the 10 now says that only levels 1 (=ground) to 10 are to be used. That's it!

7.8 For developers: some details on the internal workings

[This section is only interesting for developers]

7.8.1 Automatic selection of sub-sets of levels and lines for optimal performance

The line module is optimized such that at any time the code only uses those lines and levels that are in fact important for carrying out the task at hand. For example: if an image at one special wavelength is made (a "channel map") then the code will check which lines and levels are 'active' in the sense that they potentially contribute to the emission at that particular wavelength. This means that before the ray-tracing is done the code will make a selection of levels and lines that it will need to consider. If the lines / level lists are large, this may save a lot of computer time. Note that this is in addition to the possibility that the user makes its own sub-set selection of levels. So what happens is that at the start of the code the molecular fundamental data are read, then a subset of these levels can be chosen by the user (if not, the entire level list is used), and after that, when a specific wavelength or wavelength range is chosen for the camera routines, another subset from this subset (a "sub-sub-set") is automatically selected by RADMC-3D. The latter automatic sub-sub-set selection is redone when (during the same run, see calling RADMC-3D in child mode, Chapter 12) a new wavelength domain is chosen by the user. The user-specified sub-set of levels is, however, only done once, when the molecular data is read.

This subset and subsubset selection makes the line module a bit complex to read and understand. But it is essential

[3] http://www.strw.leidenuniv.nl/∼moldata/

for high efficiency, because if the line module has to walk through the entire line lists and level lists for each action is wants to do, and if the line/level lists are large, the code will slow down enormously.

The matter gets a bit more complicated due to the methods of ray-tracing for images and spectra (See Chapter 9). For the treatment of multi-wavelength images (method 1 in which in which all wavelength are traced within a single call to the ray-tracing routine and method 2 in which an image is made for each wavelength separetely) this is done differently: For method 1 the automatic subsubset selection is done for the complete set of wavelength simultaneously; For method 2 the automatic subsubset selection is made for each wavelength separately. Also for the way the rays are traced (sequential [method A] or as 1-D prepared rays [method B]) there are differences: For method A the automatic subsubset selection is done at the start of the image making routine, while for method B it is done at the start of each ray-trace (i.e. for each pixel separately). The latter (for each pixel separately) is useful when there are small regions in the model which have very high temperatures or very large velocities while by far the most of the model has low temperatures and velocities: then the possibility for line overlapping is only realistic in parts of the image that probe these high-T/velocity regions, while for the rest no overlapping has to be expected, and hence a smaller subsubset can be selected. That will speed up the calculation. But this works only for method B (the 1-D prepared ray method).

Chapter 8

Gas continuum opacities and emissivities

In addition to dust and line radiative transfer, RADMC-3D can also handle gas continuum opacities. Here is a list of features that are now already working, marked with [+], and those which are not yet (!!) built in, marked with [-]. Those that are currently being developed are marked with [.] and those that are ready, but are still in the testing phase are marked with [t].

The following processes are included:

[t] Gas free-free absorption/emission

[.] Gas bound-free absorption/emission

[.] Self-consistent determination of the gas temperature, based on the continuum opacity and possibly also the lines. Perhaps also various non-thermal heating processes (photoelectric heating etc) will be included. But this is future work.

8.1 Gas continuum opacities and emissivities

While under most circumstances gas emission is mostly in the form of lines, there are also continuum sources of emission and opacity. Gas continuum opacities can be included in a way very similar to the dust opacities. Since currently we envisage thermalized gas (i.e. no accelerated particle distributions, for instance), we can then use Kirchhoff's law to compute, with the gas temperature, the emissivities. Normally the user will have to determine all the locally required quantities, including the gas temperature, the ion and/or electron density etc. In the first version(s) of this module the gas temperature will not be computed self-consistently, but instead have to be given by the user. The processes will be discussed one-by-one below.

Each process can be switched on or off independently:

Process:	Command-line switch on	Command-line switch off	`radmc3d.inp` variable (0/1)
All processes	`inclgascont`	`nogascont`	`incl_gascont`
Thermal free-free	`inclfreefree`	`nofreefree`	`incl_freefree`

The main switch is the `incl_gascont`. If this is 0, all gas continuum processes are switched off. If it is 1, then gas continuum processes that have their own switch on will be included. Typically, by switching on one of the processes, the main switch will also be automatically switched on. But the reverse is not true: if you switch off one of the processes, the main switch remains on.

8.1.1 Thermal free-free emission/absorption

The process of thermal free-free emission and absorption is given by the following formula (Eq. 2.96 of Gordon & Sorochenko, 2002, Kluwer Academic Publishers):

$$\alpha_\nu^{\rm ff} = 0.2120 \frac{N_e N_i}{\nu^{2.1} T^{1.35}} \tag{8.1}$$

in units of cm^{-1} (i.e. the mean free photon path is $1/\alpha_\nu^{\rm ff}$).

This mode requires the electron_numdens(:), ion_numdens(:), and gastemp(:) arrays to be set. This is typically done by providing the files electron_numdens.inp, ion_numdens.inp and gas_temperature.inp (see Chapter A), or by allocating and setting the arrays directly in the `userdef_module.f90`.

8.2 Self-consistent gas temperature iteration

In future versions of RADMC-3D the gas temperature can, like the dust temperature, be determined through a Monte Carlo procedure. In contrast to the dust, however, the gas opacities change with temperature. The Monte Carlo simulation must therefore be repeated a few times to converge on the right temperatures.

At the moment this method is, however, not yet implemented. Instead, the gas temperature must be set by the user, either in the form of a file called `gas_temperature.inp` or in the `userdef_module.f90` module.

Chapter 9

Making images and spectra

Much has already been said about images and spectra in the chapters on dust radiative transfer and line radiative transfer. But here we will combine all this and go deeper into this material. So presumably you do not need to read this chapter if you are a beginning user. But for more sophisticated users (or as a reference manual) this chapter may be useful and presents many new features and more in-depth insight.

9.1 Basics of image making with RADMC-3D

Images and spectra are typically made after the dust temperature has been determined using the thermal Monte Carlo run (see Chapter 6). An image can now be made with a simple call to `radmc3d`[1]:

```
radmc3d image lambda 10
```

This makes an image of the model at wavelength $\lambda = 10\mu$m and writes this to the file `image.out`[2]. The vantage point is at infinity at a default inclination of 0, i.e. pole-on view. You can change the vantage point:

```
radmc3d image lambda 10 incl 45 phi 30
```

which now makes the image at inclination 45 degrees (between pole-on and edge-on) and with ϕ-angle 30 degrees in the x-y plane. Note that this 'inclination' and 'ϕ angle' are just ways to specify angles in the x-y-z space. These are angles with respect to the x-y plane. You can also rotate the camera with

```
radmc3d image lambda 10 incl 45 phi 30 posang 20
```

which rotates the camera by 20 degrees. Up to now the camera always pointed to one single point in space: the point (0,0,0). You can change this:

```
radmc3d image lambda 10 incl 45 phi 30 posang 20 pointau 3.2 0.1 0.4
```

which now points the camera at the point (3.2,0.1,0.4), where the numbers are in units of AU. The same can be done in units of parsec:

```
radmc3d image lambda 10 incl 45 phi 30 posang 20 pointpc 3.2 0.1 0.4
```

Note that `pointau` and `pointpc` are always 3-D positions specified in cartesian coordinates. This remains also true when the model-grid is in spherical coordinates and/or when the model is 2-D (axisymmetric) or 1-D (spherically symmetric): 3-D positions are always specified in x,y,z.

Let's now drop the pointing again, and also forget about the `posang`, and try to change the number of pixels used:

```
radmc3d image lambda 10 incl 45 phi 30 npix 100
```

[1] Please also read Section 14.1.3 for IDL routines that do all of this for you conveniently. The choice is up to you: you can either do this directly as described here, or use the IDL routines.

[2] We refer to Section A.15 for details of this file and how to interpret the content. See Chapter 14 for an extensive IDL tool set that make it easy to read and handle these files.

This will make an image of 100x100. You can also specify the x- and y- direction number of pixels separately:

```
radmc3d image lambda 10 incl 45 phi 30 npixx 100 npixy 30
```

Now let's forget again about the number of pixels and change the size of the image, i.e. which zooming factor we have:

```
radmc3d image lambda 10 incl 45 phi 30 sizeau 30
```

This makes an image which has 30 AU width and 30 AU height (i.e. 15 AU from the center in both directions). Same can be done in units of parsec

```
radmc3d image lambda 10 incl 45 phi 30 sizepc 30
```

Although strictly speaking redundant is the possibility to zoom-in right into a selected box in this image:

```
radmc3d image lambda 10 incl 45 phi 30 zoomau -10 -4. 0 6
```

which means that we zoom in to the box given by $-10 \leq x \leq -4$ AU and $0 \leq y \leq 6$ AU on the original image (note that `zoomau -15 15 -15 15` gives the identical result as `sizeau 30`). This possibility is strictly speaking redundant, because you could also change the `pointau` and `sizeau` to achieve the same effect (unless you want to make a non-square image, in which case this is the only way). But it is just more convenient to do any zooming-in this way. Please note that when you make non-square images with `zoomau` or `zoompc`, the code will automatically try to keep the pixels square in shape by adapting the number of pixels in x- or y- direction in the image and adjusting one of the sizes a tiny bit to assure that both x- and y- size are an integer times the pixel size. These are very small adjustments (and only take place for non-square zoom-ins). If you want to force the code to take *exactly* the zoom area, and you don't care that the pixels then become slightly non-square, you can force it with `truezoom`:

```
radmc3d image lambda 10 incl 45 phi 30 sizeau 30 zoomau -10 -4. 0 3.1415 truezoom
```

If you do not want the code to adjust the number of pixels in x- and y- direction in its attempt to keep the pixels square:

```
radmc3d image lambda 10 incl 45 phi 30 sizeau 30 zoomau -10 -4. 0 3.1415 npixx 100 npixy 4 truepix
```

Now here are some special things. Sometimes you would like to see an image of just the dust, not including stars (for stars in the image: see Section 9.7). So blend out the stars in the image, you use the `nostar` option:

```
radmc3d image lambda 10 incl 45 phi 30 nostar
```

Another special option is to get a 'quick image', in which the code does not attempt assure flux conservation in the image (see Section 9.6 for the issue of flux conservation). Doing the image with flux conservation is slower than if you make it without flux conservation. Making an image without flux conservation can be useful if you want to have a 'quick look', but is strongly discouraged for actual scientific use. But for a quick look you can do:

```
radmc3d image lambda 10 incl 45 phi 30 nofluxcons
```

Note: In the IDL widget interface `viewimage.pro` the default is to use this 'quick look' option, because you typically want to make images quickly if you use the `viewimage.pro` interface. But there is a button (called "preview") that if you unclick it, it will do flux-conserving imaging.

Finally, if you want to produce images with a smoother look (and which also are more accurate), you can ask RADMC-3D to use second order integration for the images:

```
radmc3d image lambda 10 incl 45 phi 30 secondorder
```

NOTE: The resulting intensities may be slightly different from the case when first order integration (default) is used, in particular if the grid is somewhat course and the objects of interest are optically thick. Please consult Section 9.8 for more information.

Note: All the above commands call `radmc3d` separately. If it needs to load a large model (i.e. a model with many cells), then the loading may take a long time. If you want to make many images in a row, this may take too much time. Then it is better to call `radmc3d` as a child process and pass the above commands through the biway pipe (see Chapter 12).

9.2 Making multi-wavelength images

Sometimes you want to have an image of an object at multiple wavelength simultaneously. Rather than calling RADMC-3D separately to make an image for each wavelength, you can make all images in one command. The only thing you have to do is to tell RADMC-3D which wavelengths it should take. There are various different ways you can tell RADMC-3D what wavelengths to take. This is described in detail in Section 9.4. Here we will focus as an example on just one of these methods. Type, for instance,

```
radmc3d image incl 45 phi 30 lambdarange 5. 20. nlam 10
```

This will create 10 images at once, all with the same viewing perspective, but at 10 wavelengths regularly distributed between 5 μm and 20 μm. All images are written into a single file, `image.out` (See Section A.15 for its format).

In IDL you simply type:

```
.r readradmc
a=readimage()
```

and you will get all images at once. To plot one of them:

```
plotimage,a,ilam=3
```

which will plot image number 3 (out of images number 0 to 9). To find out which wavelength this image is at:

```
print,a.lambda[3]
```

which will return 7.9370053 in this example.

Note that all of the commands in Section 9.1 are of course also applicable to multi-wavelength images, except for the `lambda` keyword, as this conflicts with the other method(s) of specifying the wavlengths of the images. Now please turn to Section 9.4 for more information on how to specify the wavelengths for the multiple wavelength images.

9.3 Making spectra

The standard way of making a spectrum with `radmc3d` is in fact identical to making 1x1 pixel images with flux conservation (i.e. recursive sub-pixeling, see Section 9.6) at multiple frequencies. You can ask `radmc3d` to make a *spectral energy distribution (SED)* with the command

```
radmc3d sed incl 45 phi 30
```

This will put the observer at inclination 45 degrees and angle phi 30 degrees, and make a spectrum with wavelength points equal to those listed in the `wavelength_micron.inp` file.

The output will be a file called `spectrum.out` (see Section A.14). In Section 14.4 it is discussed how to read this file into IDL.

You can also make a spectrum on a set of wavelength points of your own choice. There are multiple ways by which you can specify the set of frequencies/wavelength points for which to make the spectrum: they are described in Section 9.4. If you have made your selection in such a way, you can make the spectrum at this wavelength grid by

```
radmc3d spectrum incl 45 phi 30 <COMMANDS FOR WAVELENGTH SELECTION>
```

where the last stuff is telling `radmc3d` how to select the wavelengths (Section 9.4). An example:

```
radmc3d spectrum incl 45 phi 30 lambdarange 5. 20. nlam 100
```

will make a spectrum with a regular wavelength grid between 5 and 20 μm and 100 wavelength points. But see Section 9.4 for more details and options.

The output file `spectrum.out` will have the same format as for the `sed` command.

9.3.1 What is "in the beam" when the spectrum is made?

As mentioned above, a spectrum is simply made by making a rectangular image at all the wavelengths points, and integrating over these images. The resulting fluxes at each wavelength point is then the spectral flux at that wavelength point. This means that the integration area of flux for the spectrum is (a) rectangular and (b) of the same size at all wavelengths.

So, what *is* the size of the image that is integrated over? The answer is: it is the same size as the default size of an image. In fact, if you make a spectrum with

```
radmc3d spectrum incl 45 phi 30 lambdarange 5. 20. nlam 10
```

then this is the same as if you would type

```
radmc3d image incl 45 phi 30 lambdarange 5. 20. nlam 10
```

and read in the file `image.out` in into IDL (see Section 9.2) or your favorite other data language, and integrate the images to obtain fluxes. In other words: the command `spectrum` is effectively the same as the command `image` but then instead of writing out an `image.out` file, it will integrate over all images and write a `spectrum.out` file.

If you want to have a quick look at the area over which the spectrum is to be computed, but you don't want to compute all the images, just type e.g.:

```
radmc3d image lambda 10 incl 45 phi 30
```

then you see an image of your source at $\lambda = 10\mu$m, and the integration area is precisely this area – at all wavelengths. Like with the images, you can specify your viewing area, and thus your integration area. For instance, by typing

```
radmc3d image lambda 10 incl 45 phi 30 zoomau -2 -1 -0.5 0.5
```

makes an image of your source at $\lambda = 10\mu$m at inclination 45 degrees, and orientation 30 degrees, and zooms in at an are from -2 AU to -1 AU in x-direction (in the image) and from -0.5 AU to 0.5 AU in y-direction (in the image). To make an SED within the same integration area:

```
radmc3d sed incl 45 phi 30 zoomau -2 -1 -0.5 0.5
```

In this case we have an SED with a "beam size" of 1 AU diameter, but keep in mind that the "beam" is square, not circular.

9.3.2 Can one specify more realistic "beams"?

Clearly, a wavelength-independent beam size is unrealistic, and also the square beam is unrealistic. So is there a way to do this better? In reality one should really know exactly how the object is observed and how the flux is measured. If you use an interferometer, for instance, maybe your flux is meant to be the flux in a single synthesized beam. For a spectrum obtained with a slit, the precise flux is dependent on the slit width: the wider the slit, the more signal you pick up, but it is a signal from a larger area.

So if you really want to be sure that you know exactly what you are doing, then the best method is to do this youself by hand. You make multi-wavelength images:

```
radmc3d image incl 45 phi 30 lambdarange 5. 20. nlam 10
```

and integrate over the images in the way you think best mimics the actual observing procedure. You can do so, for instance, in IDL. See Section 9.2 for more information about multi-wavelength images.

But to get some reasonable estimate of the effect of the wavelength-dependent size and circular geometry of a "beam", RADMC-3D allows you to make spectra with a simplistic circular mask, the radius of which can be specified as a function of wavelength in the file `aperture_info.inp` (see Section A.16.2). This file should contain a table of mask radii at various wavelengths, and when making a spectrum with the command-line keyword `useapert` the mask radii will be found from this table by interpolation. In other words: the wavelength points of the `aperture_info.inp` file do not have to be the same as those used for the spectrum. But their range *must* be larger or equal than the range of the wavelengths used for the spectrum, because otherwise interpolation does not work. In the most extreme simplistic case the `aperture_info.inp` file contains merely two values: one for a very short wavelength (shorter than used in the spectrum) and one for a very long wavelength (longer than used in

the spectrum). The interpolation is then done double-logarithmically, so that a powerlaw is used between sampling points. So if you use a telescope with a given diameter for the entire range of the spectrum, two sampling points would indeed suffice.

You can now make the spectrum with the aperture in the following way:

```
radmc3d sed useapert dpc 100
```

The keyword `dpc 100` is the distance of the observer in units of parsec[3], here assumed to be 100. This distance is necessary because the aperture information is given in arcseconds, and the distance is used to convert this is image size.

Important note: Although you specify the distance of the observer here, the `spectrum.out` file that is produced is still normalized to a distance of 1 parsec.

Note also that in the above example you can add any other keywords as shown in the examples before, as long as you add the `useapert` keyword and specify `dpc`.

A final note: the default behavior of RADMC-3D is to use the square field approach described before. You can explicitly turn off the use of apertures (which may be useful in the child mode of RADMC-3D) with the keyword `noapert`, but normally this is not necessary as it is the default.

9.4 Specifying custom-made sets of wavelength points for the camera

If you want to make a spectrum at a special grid of wavelengths/frequencies, with the `spectrum` command (see Section 9.3), you must tell `radmc3d` which wavelengths you want to use. Here is described how to do this in various ways.

9.4.1 Using `lambdarange` and (optionally) `nlam`

The simplest way to choose a set of wavelength for a spectrum is with the `lambdarange` and (optionally) `nlam` command line options. Here is how to do this:

```
radmc3d spectrum incl 45 phi 30 lambdarange 5. 20.
```

This will make a spectrum between 5 and 20 μm. It will use by default 100 wavelength points logarithmically spaced between 5 and 20 μm. You can change the number of wavelength points as well:

```
radmc3d spectrum incl 45 phi 30 lambdarange 5. 20. nlam 1000
```

This will do the same, but creates a spectrum of 1000 wavelength points.

9.4.2 Using `loadcolor`

By giving the command `loadcolor` on the command line, `radmc3d` will search for the file `color_inus.inp`. This file contains integers selecting the wavelengths from the file `wavelength_micron.inp`. The file is described in Section A.16.1.

9.4.3 Using `loadlambda`

By giving the command `loadlambda` on the command line, `radmc3d` will search for the file `camera_wavelength_micron.inp`. This file contains a list of wavelengths in micron which constitute the grid in wavelength. This file is described in Section A.12.

[3] This is still (and only) valid in the observer-at-infinity default mode. But the distance is necessary for internal computations as described in the text.

9.4.4 Using `iline`, `imolspec` etc (for when lines are included)

By adding for instance `iline 3` to the command line you specify a window around line number 3 (by default of molecule 1). By also specifying for instance `imolspec 2` you select line 3 of molecule 2. By adding `widthkms 3` you specify how wide the window around the line should be (3 km/s in this example). With `vkms 2` you set the window offset from line center by 2 km/s in this example. By adding `linenlam 30` you set the number of wavelength points for this spectrum to be 30 in this example. So a complete (though different) example is:

```
radmc3d spectrum incl 45 phi 30 iline 2 imolspec 1 widthkms 6.0 vkms 0.0 linenlam 40
```

9.5 Heads-up: In reality wavelength are actually wavelength bands

In a radiative transfer program like `RADMC-3D` the images or spectral fluxes are calculated at *exact* wavelengths. This would correspond to making observations with infinitely narrow filters, i.e. filters with $\Delta\lambda = 0$. This is not how real observations work. In reality each wavelength channel has a finite width $\Delta\lambda$ and the measured flux (or image intensity) is an average over this range. To be even more precise, each wavelength channel i has some profile $\Phi_i(\lambda)$ defined such that

$$\int_0^\infty \Phi_i(\lambda) d\lambda = 1 \tag{9.1}$$

For wide filters such as the standard photometric systems (e.g. UVBRI in the optical and JHK in the near infrared) these profiles span ranges with a width of the order of λ itself. Many instruments have their own set of filters. Usually one can download these profiles as digital tables. It can, under some circumstances, be important to include a treatment of these profiles in the model predictions. As an example take the N band. This is a band that includes the 10 μm silicate feature, which is a strong function of wavelength *within* the N band. If you have a wide filter in the N band, then one cannot simply calculate the model spectrum in one single wavelength. Instead one has to calculate it for a properly finely sampled set of wavelengths λ_i for $1 \leq i \leq n$, where n is the number of wavelength samples, and then compute the filter-averaged flux with:

$$F_{band} = \int_0^\infty \Phi_i(\lambda) F(\lambda) d\lambda = \sum_{i=1}^{n} \Phi_i F_i \delta\lambda \tag{9.2}$$

where $\delta\lambda$ is the wavelength sampling spacing used. The same is true for image intensities. `RADMC-3D` will not do this automatically. You have to tell it the λ_i sampling points, let it make the images or fluxes, and you will then have to perform this sum yourself. *Note that this will not always be necessary!* In many (most?) cases the dust continuum is not expected to change so dramatically over the width of the filter that such degree of accuracy is required. So you are advised to think carefully: "do I need to take care of this or can I make do with a single wavelength sample for each filter?". If the former, then do the hard work. If the latter: then you can save time.

9.5.1 Using channel-integrated intensities to improve line channel map quality

When you make line channel maps you may face a problem that is somehow related to the above issue of single-λ-sampling versus filter-integrated fluxes/intensities. If the model contains gas motion, then doppler shift will shift the line profile around. In your channel map you may see regions devoid of emission because the lines have doppler shifted out of the channel you are looking at. However, as described in Section 7.5, if the intrinsic line width of the gas is smaller than the cell-to-cell velocity differences, then the channel images may look very distorted (they will look "blocky", as if there is a bug in the code). Please refer to Section 7.5 for more details and updates on this important, but difficult issue. It is not a bug, but a general problem with ray-tracing of gas lines in models with large velocity gradients.

As one of the β-testers of `RADMC-3D`, Rahul Shetty, has found out, this problem can often be alleviated a lot if you treat the finite width of a channel. By taking multiple λ_i points in each wavelength channel (i.e. multiple v_i points in each velocity channel) and simply averaging the intensities (i.e. assuming a perfectly square Φ function) and taking the width of the channels to be not smaller (preferably substantially wider) than the cell-to-cell velocity differences, this "blocky noise" sometimes smoothes out well. However, it is always safer to use the "doppler catching" mode (see Section 7.6) to automatically prevent such problems (though this mode requires more computer memory).

9.6 The issue of flux conservation: recursive sub-pixeling

9.6.1 The problem of flux conservation in images

If an image of nx×ny pixels is made simply by ray-tracing one single ray for each pixel, then there is the grave danger that certain regions with high refinement (for instance with AMR in cartesian coordinates, or near the center of the coordinate system for spherical coordinates) are not properly 'picked up'. An example: suppose we star with a circumstellar disk ranging from 0.1 AU out to 1000 AU. Most of the near infrared flux comes from the very inner regions near 0.1 AU. If an image of the disk is made with 100x100 pixels and image half size of 1000 AU, then none of the pixels in fact pass through these very bright inner regions, for lack of spatial resolution. The problem is then that the image, when integrated over the entire image, does not have the correct flux. What *should* be is that the centermost pixels contain the flux from this innermost region, even if these pixels are much larger than the entire bright region. In other words, the intensity of these pixels must represent the average intensity, averaged over the entire pixel. Strictly speaking one should trace an infinite continuous 2-D series of rays covering the entire pixel and then average over all these rays; but this is of course not possible. In practice we should find a way to estimate the average intensity with only a finite number of rays.

9.6.2 The solution: recursive sub-pixeling

In RADMC-3D what we do is to use some kind of 'adaptive grid refinement' of the pixels of the image. For each pixel in the image the intensity is computed through a call to a subroutine called `camera_compute_one_pixel()`. In this subroutine a ray-tracing is performed for a ray that ends right in the middle of our pixel. During the ray-tracing, however, we check if we pass regions in the model grid that have grid cells with sizes S that are smaller than the pixel size divided by some factor $f_{\rm ref}$ (where pixel size is, like the model grid size S itself, measured in centimeters[4]). If this is found *not* to be true, then the pixel size was apparently ok, and the intensity resulting from the ray-tracing is now returned as the final intensity of this pixel. If, however, this condition *is* found to be true, then the result of this ray is rejected, and instead 2x2 sub-pixels are computed by calling the `camera_compute_one_pixel()` subroutine recursively. We thus receive the intensity of each of these four sub-pixels, and we return the average of these 4 intensities.

Note, by the way, that each of these 2x2 subpixels may be split even further into 2x2 sub-pixels etc until the desired resolution is reached, i.e. until the condition that S is larger or equal to the pixel size divided by $f_{\rm ref}$ is met. By this recursive calling, we always end up at the top level with the average intesity of the entire top-level pixel. This method is very similar to quad-tree mesh refinement, but instead of retaining and returning the entire complex mesh structure to the user, this method only returns the final average intensity of each (by definition top level) pixel in the image. So the recursive sub-pixeling technique described here is all done internally in the RADMC-3D code, and the user will not really notice anything except that this sub-pixeling can of course be computationally more expensive than if such a method is not used.

Note that the smaller we choose $f_{\rm ref}$ the more accurate our image becomes. In the `radmc3d.inp` file the value of $f_{\rm ref}$ can be set by setting the variable `camera_refine_criterion` to the value you want $f_{\rm ref}$ to be. Not setting this variable means RADMC-3D will use the default value which is reasonable as a choice (default is 1.0). The smaller you set `camera_refine_criterion`, the more accurate and reliable the results become (but the heavier the calculation becomes, too).

NOTE: The issue of recursive sub-pixeling becomes tricky when stars are treated as spheres, i.e. non-point-like (see Section 9.7 and Chapter 11).

9.6.3 A danger with recursive sub-pixeling

It is useful to keep in mind that for each pixel the recursive sub-pixeling is triggered if the ray belonging to that pixel encounters a cell that is smaller than the pixel size. This *normally* works well if $f_{\rm ref}$ is chosen small enough. But if there exist regions in the model where one big non-refined cell lies adjacent to a cell that is refined, say, 4 times (meaning the big cell has neighbors that are 16 times smaller!), then if the ray of the pixel just happens to miss

[4] This is not possible for images for local observers, but see Section 9.10 for details.

the small cells and only passes the big cell, it won't "notice" that it may need to refine to correctly capture the tiny neighboring cells accurarely.

Such a problem only happens if refinement levels jump by more than 1 between adjacent cells. If so, then it may be important to make $f_{\rm ref}$ correspondingly smaller. A bit of experimentation may be needed here.

9.6.4 Recursive sub-pixeling in spherical coordinates

In spherical coordinates the recursive sub-pixeling has a few issues that you may want to be aware of. First of all, in 1-D spherical coordinates each cell is in fact a shell of a certain thickness. In 2-D spherical coordinates cells are rings. In both cases the cells are not just local boxes, but have 2 or 1 (respectively) extended dimensions. RADMC-3D takes care to still calculate properly how to define the recursive sub-pixeling scale. But for rays that go through the central cavity of the coordinate system there is no uniquely defined pixel resolution to take. The global variable `camera_spher_cavity_relres` (with default value 0.05) defines such a relative scale. You can change this value in the `radmc3d.inp` file.

A second issue is when the user introduces extreme "separable refinement" (see Section 10.2) in the R coordinate. This may, for instance, be necessary near the inner edge of a dusty disk model in order to keep the first cell optically thin. This may lead, however, to extremely deep sub-pixeling for rays that skim the inner edge of the grid. This leads to a huge slow-down of the ray-tracing process although it is likely not to give much a different result. To avoid this RADMC-3D has a global variable `camera_min_aspectratio` (default value is 0.05) that limits this. You can change it in the `radmc3d.inp` file. The smaller you make this number, the more accurate and reliable the results. *It may be prudent to experiment with smaller values for models with extremely optically thick inner edges, e.g. a protoplanetary disk with an abrupt inner edge and a high dust surface density.*

9.6.5 How can I find out which pixels RADMC-3D is recursively refining?

Sometimes you notice that the rendering of an image or spectrum takes much more time than you expected. When recursive sub-pixeling is used for imaging, RADMC-3D will give diagnostic information about how many more pixels it has rendered than the original image resolution. This factor can give some insight if extreme amount of sub-pixeling refinement has been used. But it does not say where in the image this occurs. If you want to see exactly which pixels and subpixels RADMC-3D has rendered for some image, you can use the following command-line option:

```
radmc3d image lambda 10 diag_subpix
```

This `diag_subpix` option will tell RADMC-3D to write a file called `subpixeling_diagnostics.out` which contains four columns: One for the x-coordinate of the (sub-)pixel, one for the y-coordinate of the (sub-)pixel, one for the x-width of the (sub-)pixel and a final one for the y-width of the (sub-)pixel. In IDL, if you use for instance the astrolib library, you can use the `readcol` procedure to read these columns. So *inside IDL* you then type

```
.r readcol
readcol,'subpixeling_diagnostics.out',px,py
plot,px,py,psym=3
```

and you get a plot of all the pixel-centers and sub-pixel-centers used.

9.6.6 Alternative to recursive sub-pixeling

As an alternative to using this recursive sub-pixeling technique to ensure flux conservation, one can simply enhance the spatial resolution of the image. This has the clear advantage that the user gets the complete information of the details in the image (while in the recursive sub-pixeling technique only the averages are retained). The clear disadvantages are that one may need rediculously high-resolution images (i.e. large data sets) to resolve all the details and one may waste a lot of time rendering parts of the image which do not need that resolution. The latter is typically an issue when images are rendered from models that use AMR techniques.

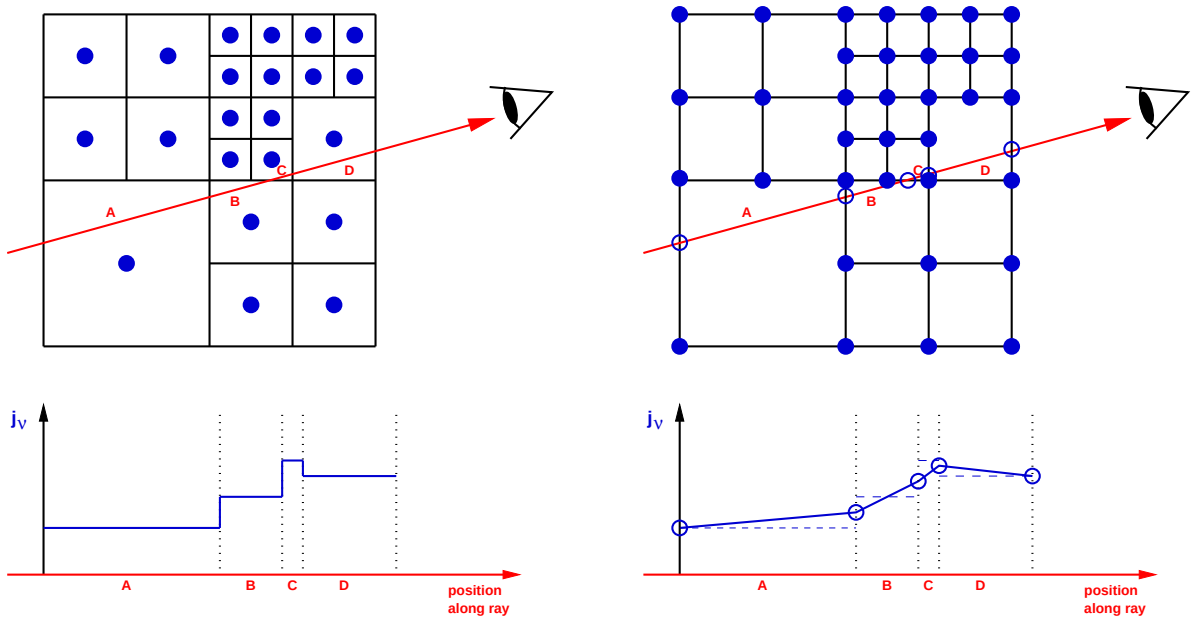


Figure 9.1. Pictographic representation of the integration of the transfer equation along a ray (red line with arrow head) through an AMR grid (black lines). The grid cuts the ray into ray segments A, B, C and D. At the bottom it is shown how the integrands are assumed to be along these four segments. Left: When using first order integration. The emissivity function j_ν and extinction function α_ν are constant within each cell and thus constant along each ray segment. Right: When using second order integration. The emissivity function j_ν and extinction function α_ν are given at the cell corners (solid blue circles), and linearly interpolated from the cell corners to the locations where the ray crosses the cell walls (open blue circles). Then, along each ray segment the emissivity and extinction functions are assumed to be linear functions, so that the integration result is quadratic. The thin blue horizontal dashed lines are the same as those in the Left panel, and are just there for comparison. Note that these figures are 2-D, whereas this actually happens in 3-D. See Section 9.8 for more information.

9.7 Stars in the images and spectra

Per default, stars are still treated as point sources. That means that none of the rays of an image can be intercepted by a star. Starlight is included in each image as a post-processing step. First the image is rendered without the stars (though with of course all the emission of dust, lines etc *induced* by the stars) and then for each star a ray tracing is done from the star to the observer (where only extinction is taken into account, because the emission is already taken care of) and the flux is then added to the image at the correct position. You can switch off the inclusion of the stars in the images or spectra with the `nostar` command line option.

However, as of version 0.17, stars can also be treated as the finite-size spheres they are. This is done with setting `stars_sphere = 1` in `radmc3d.inp`. However, this mode can slow down the code a bit or even substantially. And it may still be partly under development, so the code may stop if it is required to handle a situation it cannot handle yet. See Chapter 11 for details.

9.8 Second order ray-tracing (Important information!)

Ideally we would like to assure that the model grid is sufficiently finely spaced everywhere. But in many cases of interest one does not have this luxury. One must live with the fact that, for memory and/or computing time reasons, the grid is perhaps a bit coarser than would be ideal. In such a case it becomes important to consider the "order" of integration of the transfer equation. By default, for images and spectra, RADMC-3D uses first order integration: The source term and the opacity in each cell are assumed to be constant over the cell. This is illustrated in Fig. 9.1-Left. The integration over each cell proceeds according to the following formula:

$$I_{\rm result} = I_{\rm start} e^{-\tau} + (1 - e^{-\tau}) S \tag{9.3}$$

where $S = j/\alpha$ is the source function, assumed constant throughout the cell, $\tau = \alpha\,\Delta s$ is the optical depth along the path that the ray makes through the cell, and $I_{\rm start}$ is the intensity upon entering the cell. This is the default used by RADMC-3D because the Monte Carlo methods also treat cells as having constant properties over each cell. This type of simple integration is therefore the closest to how the Monte Carlo methods (thermal MC, scattering MC and mono MC) "see" the grid. However, with first order integration the images look somewhat "blocky": you can literally see the block structure of the grid cells in the image, especially if you make images at angles aligned with the grid. For objects with high optical depths you may even see grid patterns in the images.

RADMC-3D can also use second order integration for its images and spectra. This is illustrated in Fig. 9.1-Right. This is done with a simple "`secondorder`" option added on the command line, for instance:

```
radmc3d image lambda 10 secondorder
```

The integration now follows the formula (Olson et al. 1986):

$$I_{\rm result} = I_{\rm start} e^{-\tau} + (1 - e^{-\tau} - \beta) S_{\rm start} + \beta S_{\rm end} \tag{9.4}$$

with

$$\beta = \frac{\tau - 1 + e^{-\tau}}{\tau} \tag{9.5}$$

and

$$\tau = \frac{\alpha_{\rm start} + \alpha_{\rm end}}{2} \Delta s \tag{9.6}$$

For $\tau \rightarrow 0$ we have the limit $\beta \rightarrow \tau/2$, while for $\tau \rightarrow \infty$ we have the limit $\beta \rightarrow 1$.

The values of α, S etc., at the "start" position are obtained at the cell interface where the ray enters the cell. The values at the "end" position are obtained at the cell interface where the ray leaves the cell. The above formulas represent the exact solution of the transfer equation along this ray-section if we assume that all variables are linear functions between the "start" and "end" positions.

The next question is: How do we determine the physical variables at the cell interfaces ("start" and "end")? After all, initially all variables are stored for each cell, not for each cell interface or cell corner. The way that RADMC-3D does this is:

- First create a "grid of cell corners", which we call the *vertex grid* (see the solid blue dots in Fig. 9.1-Right). The cell grid already implicitly defines the locations of all the cell corners, but these corners are, by default, not explicitly listed in computer memory. When the `secondorder` option is given, however, RADMC-3D will explicitly find all cell corners and assign an identity (a unique integer number) to each one of them. NOTE: Setting up this vertex grid costs computer memory!
- At each vertex (cell corner) the physical variables of the (up to) 8 cells touching the vertex are averaged with equal weight for each cell. This now maps the physical variables from the cells to the vertices.
- Whenever a ray passes through a cell wall, the physical variables of the 4 vertices of the cell wall are interpolated bilinearly onto the point where the ray passes through the cell wall (see the open blue circles in Fig. 9.1-Right). This gives the values at the "start" or "end" points.
- Since the current "end" point will be the "start" point for the next ray segment, the physical variables need only be obtained once per cell wall, as they can be recycled for the next ray segment. Each set of physical variables will thus be used twice: once for the "end" and once for the "start" of a ray segment (except of course at the very beginning and very end of the ray).

If you compare the images or spectra obtained with first order integration (default) or second order integration (Fig. 9.2) you see that with the first order method you still see the cell structure of the grid very much. Also numerical noise in the temperature due to the Monte Carlo statistics is much more prominent in the first order method. The second order method makes much smoother results.

For line transfer the second order mode can be even improved with the "doppler catching method", see Section 7.6.

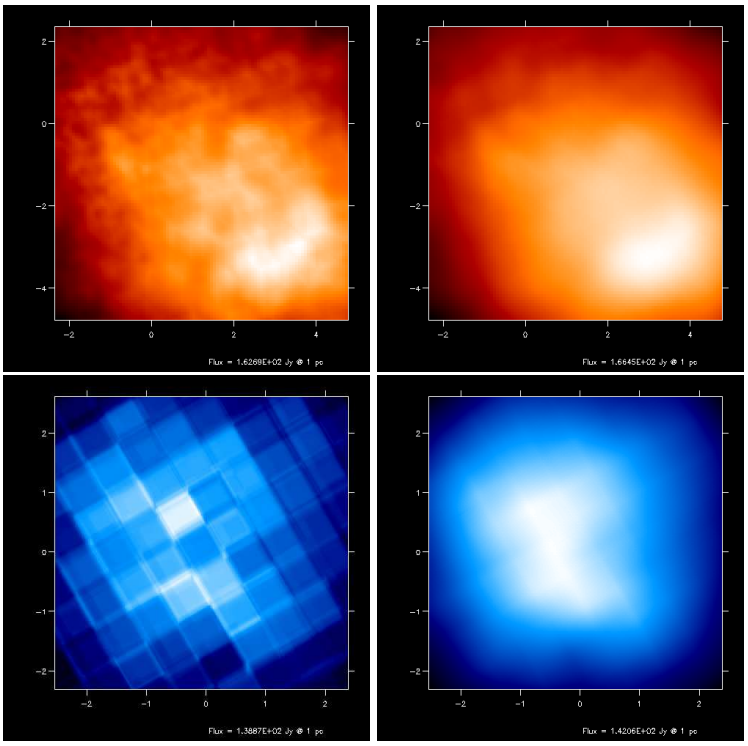


Figure 9.2. First-order integration of transfer equation in ray-tracing (left two panels), versus second order integration (right two panels). Upper: 60 degrees inclination, Lower: 4 degrees inclination. Shown here is the `run_simple_1_layers` model (in the `examples` directory) at $\lambda = 10$ μm at some zoom factor. See Section 9.8 for more information.

9.9 Using circularly arranged pixels for spectra (special topic)

In the predecessor code (RADMC) the issue with flux conservation was dealt with using a trick different from subpixeling: Rather than arranging the pixels of the images in rows and columns, the pixels were arranged in concentric circles. The radii of these circles are tuned to the radii of the spherical coordinate system. In this way the huge dynamic range of scales of the model could be dealt with automatically.

Here, in RADMC-3D, we do not really need this trick, because of the new technique of recursive subpixeling (see Section 9.6.2).

But it might sometimes nevertheless be useful to use this circular pixel arrangement, because it is faster (though less reliable) than the recursive subpixeling. Also, the results would be easier to compare to the results of RADMC. But this mode works only when you use spherical coordinates! Also, there is no recursive subpixeling done when you use this mode, so if you use spherical coordinate *and* AMR grid refinement, then the refined regions may be not well resolved and flux may not be well conserved. And it works only for spectra. Images will remain rectangular pixel arrangements.

You can active it by the command-line option `circ`, *iff* you specified `spectrum` or `sed` as well.

9.10 For public outreach work: local observers inside the model

While it may not be very useful for scientific purposes (though there may be exceptions), it is very nice for public outreach to be able to view a model from the inside, as if you, as the observer, were standing right in the middle of the model cloud or object. One can then use physical or semi-physical or even completely ad-hoc opacities to create the right 'visual effects'. RADMC-3D has a viewing mode for this purpose. You can use different projections:

- *Projection onto flat screen:*
  The simplest one is a projection onto a screen in front (or behind) the point-location of the observer. This gives

an image that is good for viewing in a normal screen. This is the default (`camera_localobs_projection=1`).

- Another projection is a projection onto a sphere, which allow fields of view that are equal or larger than 2π of the sky. It may be useful for projection onto an OMNIMAX dome. This is projection mode `camera_localobs_projection=2`.

You can set the variable `camera_localobs_projection` to 1 or 2 by adding on the command line `projection 2` (or 1), or by setting it in the `radmc3d.inp` as a line `camera_localobs_projection = 2` (or 1).

To use the local projection mode you must specify the following variables on the command line:

- `sizeradian`:
  This sets the size of the (square) image in radian. Setting this will make the image square (like setting `sizeau` in the observer-at-infinity mode, see Section 9.1).
- `zoomradian`:
  Instead of `sizeradian` you can also specify `zoomradian`, which is the local-observer version of `zoomau` or`zoompc` (see Section 9.1).
- `posang`:
  The position angle of the camera. Has the same meaning as in the observer-at-infinity mode.
- `locobsau` or `locobspc`:
  Specify the 3-D location of the local observer inside the model in units of AU or parsec. This requires 3 numbers which are the x, y and z positions (also when using spherical coordinates for the model setup: these are still the cartesian coordinates).
- `pointau` or `pointpc`:
  These have the same meaning as in the observer-at-infinity model. They specify the 3-D location of the point of focus for the camera (to which point in space is the camera pointing) in units of AU or parsec. This requires 3 numbers which are the x, y and z positions (also when using spherical coordinates for the model setup: these are still the cartesian coordinates).

Setting `sizeradian`, `zoomradian`, `locobsau` or `locobspc` on the command line automatically switches to the local observer mode (i.e. there is no need for an extra keyword setting the local observer mode on). To switch back to observer-at-infinity mode, you specify e.g. `incl` or `phi` (the direction toward which the observer is located in the observer-at-infinity mode). Note that if you accidently specify both e.g. `sizeradian` and `incl`, you might end up with the wrong mode, because the mode is set by the last relevant entry on the command line.

The images that are produced using the local observer mode will have the x- and y- pixel size specifications in radian instead of cm. The first line of an image (the format number of the file) contains then the value 2 (indicating local observer image with pixel sizes in radian) instead of 1 (which indicates observer-at-infinity image with pixel sizes in cm).

NOTE: For technical reasons dust scattering is (at least for now) not included in the local observer mode! It is discouraged to use the local observer mode for scientific purposes.

9.11 Multiple vantage points: the "Movie" mode

It can be useful, both scientifically and for public outreach, to make movies of your model, for instance by showing your model from different vantage points or by "travelling" through the model using the local observer mode (Section 9.10). For a movie one must make many frames, each frame being an image created by RADMC-3D's image capabilities. If you call `radmc3d` separately for each image, then often the reading of all the large input files takes up most of the time. One way to solve this is to call `radmc3d` in "child mode" (see Chapter 12). But this is somewhat complicated and cumbersome. A better way is to use RADMC-3D's "movie mode". This allows you to ask RADMC-3D to make a sequence of images in a single call. The way to do this is to call `radmc3d` with the `movie` keyword:

```
radmc3d movie
```

This will make `radmc3d` to look for a file called `movie.inp` which contains the information about each image it should make. The structure of the `movie.inp` file is:

```
iformat
nframes
<<information for frame 1>>
<<information for frame 2>>
<<information for frame 3>>
...
<<information for frame nframes>>
```

The `iformat` is an integer that is described below. The `nframes` is the number of frames. The <<`information for frame xx`>> are lines containing the information of how the camera should be positioned for each frame of the movie (i.e. for each imag). It is also described below.

There are multiple ways to tell RADMC-3D how to make this sequence of images. Which if these ways RADMC-3D should use is specified by the `iformat` number. Currently there are 2, but later we may add further possibilities. Here are the current possibilities

- **iformat=1:**
  The observer is at infinity (as usual) and the <<`information for frame xx`>> consists of the following numbers (separated by spaces):

  ```
  pntx pnty pntz hsx hsy pa incl phi
  ```

  These 8 numbers have the following meaning:

  - `pntx,pnty,pntz`
    These are the x, y and z coordinates (in units of cm) of the point toward which the camera is pointing.
  - `hsx,hsy`
    These are the image half-size in horizontal and vertical direction on the image (in units of cm).
  - `pa`
    This is the position angle of the camera in degrees. This has the same meaning as for a single image.
  - `incl,phi`
    These are the inclination and phi angle toward the observer in degrees. These have the same meaning as for a single image.

- **iformat=-1:**
  The observer is local (see Section 9.10) and the <<`information for frame xx`>> consists of the following numbers (separated by spaces):

  ```
  pntx pnty pntz hsx hsy pa obsx obsy obsz
  ```

  These 9 numbers have the following meaning:

  - `pntx,pnty,pntz,hsx,hsy,pa`
    Same meaning as for iformat=1.
  - `obsx,obsy,obsz`
    These are the x, y and z position of the local observer (in units of cm).

Apart from the quantities that are thus set for each image separately, all other command-line options still remain valid.

Example, let us make a movie of 360 frames of a model seen at infinity while rotating the object 360 degrees, and as seen at a wavelength of $\lambda = 10\mu$m with 200x200 pixels. We construct the `movie.inp` file:

```
1
360
0. 0. 0. 1d15 1d15 0. 60.  1.
0. 0. 0. 1d15 1d15 0. 60.  2.
0. 0. 0. 1d15 1d15 0. 60.  3.
.
.
.
```

```
0. 0. 0. 1d15 1d15 0. 60.  358.
0. 0. 0. 1d15 1d15 0. 60.  359.
0. 0. 0. 1d15 1d15 0. 60.  360.
```

We now call RADMC-3D in the following way:

```
radmc3d movie lambda 10. npix 200
```

This will create image files `image_0001.out`, `image_0002.out`, all the way to `image_0360.out`. The images will have a full width and height of 2×10^{15}cm (about 130 AU), will always point to the center of the image, will be taken at an inclination of 60 degrees and with varying ϕ-angle.

Another example: let us move through the object (local observer mode), approaching the center very closely, but not precisely:

```
-1
101
0. 0. 0. 0.8 0.8 0. 6.d13 -1.0000d15 0.
0. 0. 0. 0.8 0.8 0. 6.d13 -0.9800d15 0.
0. 0. 0. 0.8 0.8 0. 6.d13 -0.9600d15 0.
.
.
0. 0. 0. 0.8 0.8 0. 6.d13 -0.0200d15 0.
0. 0. 0. 0.8 0.8 0. 6.d13  0.0000d15 0.
0. 0. 0. 0.8 0.8 0. 6.d13  0.0200d15 0.
.
.
0. 0. 0. 0.8 0.8 0. 6.d13  0.9600d15 0.
0. 0. 0. 0.8 0.8 0. 6.d13  0.9800d15 0.
0. 0. 0. 0.8 0.8 0. 6.d13  1.0000d15 0.
```

Here the camera automatically rotates such that the focus remains on the center, as the camera flies by the center of the object at a closest-approach to the center of 6×10^{13}cm. The half-width of the image is 0.8 radian.

9.12 For developers: some details on the internal workings

[This section is only interesting for developers]

9.12.1 Multi-wavelength images and spectra: two methods (1 and 2)

[TO BE COMPLETED]

9.12.2 Ray-tracing: two methods (A and B)

The camera module of RADMC-3D features two different ways of tracing a ray for making images and spectra:

- **Method A:** Tracing in a sequential step-by-step fashion, whereby at each step the opacities are computed, the intensities are updated and the next position of the ray in the 3-D model is determined. This is the method by default, and it the simplest method. But it may not always be the most optimized in terms of speed.
- **Method B:** First find out how the ray goes through the 3-D model, and prepare a 1-D array of dust tempetures and densities, line transfer quantities (such as level populations etc) and Δs values (length of ray elements). Then compute the opacities at each point. Then finally do the 1-D formal transfer. This method has the advantage that it lends itself well for parallellization on a GPU (which only gives speed-up if method 2 is used for multi-wavelength images and spectra, see Section 9.12.1). Also, by scouting the entire ray before doing the full transfer – the automatic line/level subsubset selection for the line transfer (see Section 7.8.1) – can be done and may give some speed up in some cases for the line ray-tracing.

Chapter 10

More information about the gridding

We already discussed the various types of grids in Section 5.3, and the grid input file structure is described in Section A.2. In this chapter let us take a closer look at the gridding possibilities and things to take special care of.

10.1 Regular grids

A regular grid is called "grid style 0" in RADMC-3D. It can be used in Cartesian coordinates as well as in spherical coordinates (Section 5.2).

A regular grid, in our definition, is a multi-dimensional grid which is separable in x, y and z (or in spherical coordinates in r, θ and ϕ). You specify a 1-D monotonically increasing array of values $x_1, x_2, \cdots, x_{\rm nx+1}$ which represent the cell walls in $x - direction$. You do the same for the other directions: $y_1, y_2, \cdots, y_{\rm ny+1}$ and $z_1, z_2, \cdots, z_{\rm nz+1}$. The value of, say, x_2 is the same for every position in y and z: this is what we mean with "separable".

In Cartesian coordinates RADMC-3D enforces perfectly cubic grid cells (i.e. linear grids). But that is only to make the image sub-pixeling easier (see Section 9.6.2). For spherical grids this is not enforced, and in fact it is strongly encouraged to use non-linear grids in spherical coordinates. Please read Section 10.2 if you use spherical coordinates!

In a regular grid you specify the grids in each direction separately. For instance, the x-grid is given by specifying the cell walls in x-direction. If we have, say, 10 cells in x-direction, we must specify 11 cell wall positions. For instance: $x_i = \{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$. For the y-direction and z-direction likewise. Fig. 10.1 shows an example of a 2-D regular grid of 4x3 cells. In Cartesian coordinates we *must* define our model in full 3-D (proper 2-D and 1-D modes are not available, but see Section 10.6 how to simulate a 1-D plane-parallel mode). In Cartesian coordinates the cell sizes *must* be perfectly cubical, i.e. the spacing in each direction must be the same. If you need a finer grid in some location, you can use the AMR capabilities discussed below.

In spherical coordinates you can choose between 1-D spherically symmetric models, 2-D axisymmetric models or fully 3-D models. In spherical coordinates you do *not* have restrictions to the cell geometry or grid spacing. You can choose any set of numbers $r_1, \cdots, r_{\rm nr}$ as radial grid, as long as this set of numbers is larger than 0 and monotonically increasing. The same is true for the θ-grid and the ϕ-grid.

The precise way how to set up a regular grid using the `amr_grid.inp` file is described in Section A.2.1. The input of any spatial variables (such as e.g. the dust density) uses the sequence of grid cells in the same order as the cells are specified in that `amr_grid.inp` file.

10.2 Separable grid refinement in spherical coordinates (important!)

Spherical coordinates are a very powerful way of dealing with centrally-concentrated problems. For instance, collapsing protostellar cores, protoplanetary disks, disk galaxies, dust tori around active galactic nuclei, accretion disks around compact objects, etc. In other words: problems in which a single central body dominates the problem, and

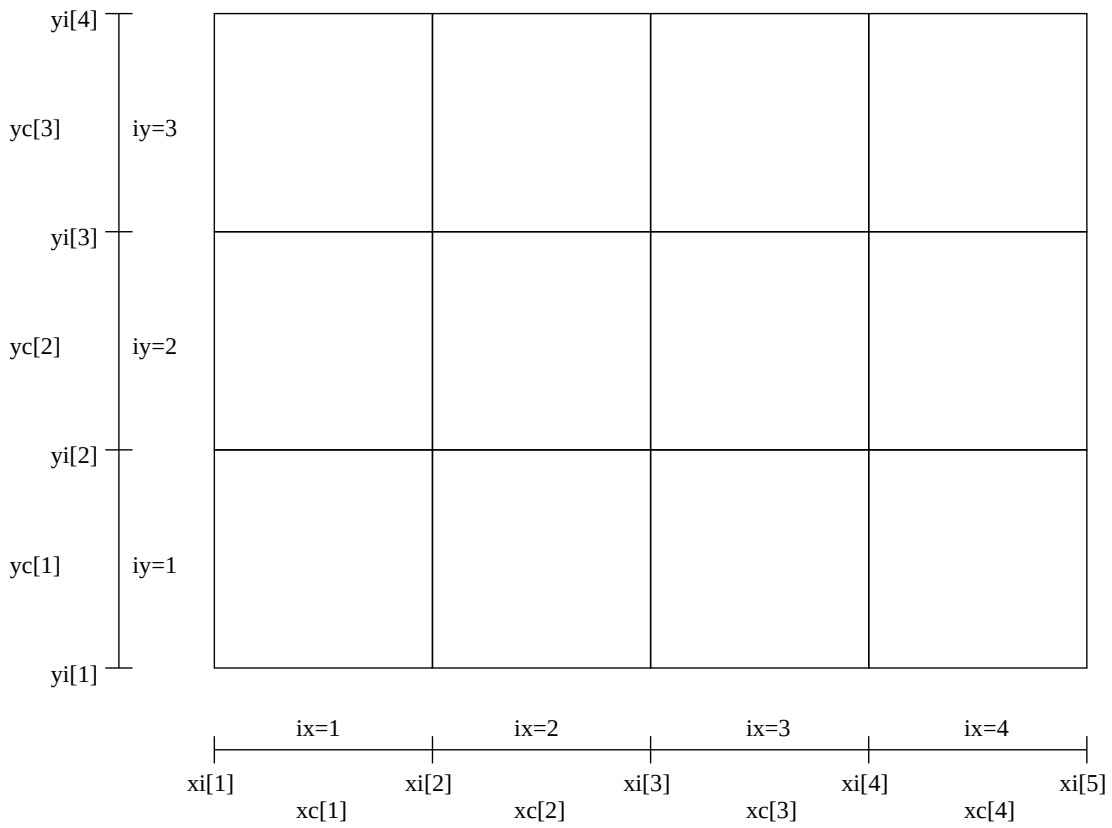


Figure 10.1. Example of a regular 2-D grid with nx=4 and ny=3.

material at all distances from the central body matters. For example a disk around a young star goes all the way from 0.01 AU out to 1000 AU, covering 5 orders of magnitude in radius. Spherical coordinates are the easiest way of dealing with such a huge radial dynamic range: you simply make a radial grid, where the grid spacing $r_{i+1} - r_i$ scales roughly with r_i.

This is called a *logarithmic radial grid*. This is a grid whith a spacing in which $(r_{i+1} - r_i)/r_i$ is constant with r. In this way you assure that you have always the right spatial resolution in r at each radius. In spherical coordinates it is highly recomended to use such a log spacing. But you can also refine the r grid even more (in addition to the log-spacing). This is also strongly recommended near the inner edge of a circumstellar shell, for instance. Or at the inner dust rim of a disk. There you must refine the r grid (by simply making the spacing smaller as you approach the inner edge from the outside) to assure that the first few cells are optically thin and that there is a gradual transition from optically thin to optically thick as you go outward. This is particularly important for, for instance, the inner rim of a dusty disk.

In spherical coordinates you can vary the spacing in r, θ and ϕ completely freely. That means: you could have for instance r to be spaced as $1.00, 1.01, 1.03, 1.05, 1.1, 1.2, 1.35, \cdots$. There is no restriction, as long as the coordinate points are monotonically increasing.

For models of accretion disks it can, for instance, be useful to make sure that there are more grid points of θ near the equatorial plane $\theta = \pi/2$. So the grid spacing between $\theta = 0.0$ and $\theta = 1.0$ may be very coarse while between $\theta = 1.0$ and $\theta = \pi/2$ you may put a finer grid. All of this "grid refinement" can be done without the "AMR" refinement technique: this is the "separable" grid refinement, because you can do this separately for r, for θ and for ϕ.

Sometimes, however, separable refinement may not help you to refine the grid where necessary. For instance: if you model a disk with a planet in the disk, then you may need to refine the grid around the planet. You could refine the grid in principle in a separable way, but you would then have a large redundancy in cells that are refined by far away from the planet. Or if you have a disk with an inner rim that is not exactly at $r = r_{\rm rim}$, but is a rounded-off rim. In these cases you need refinement exactly located at the region of interest. For that you need the "AMR" refinement (Sections 10.3 and 10.4).

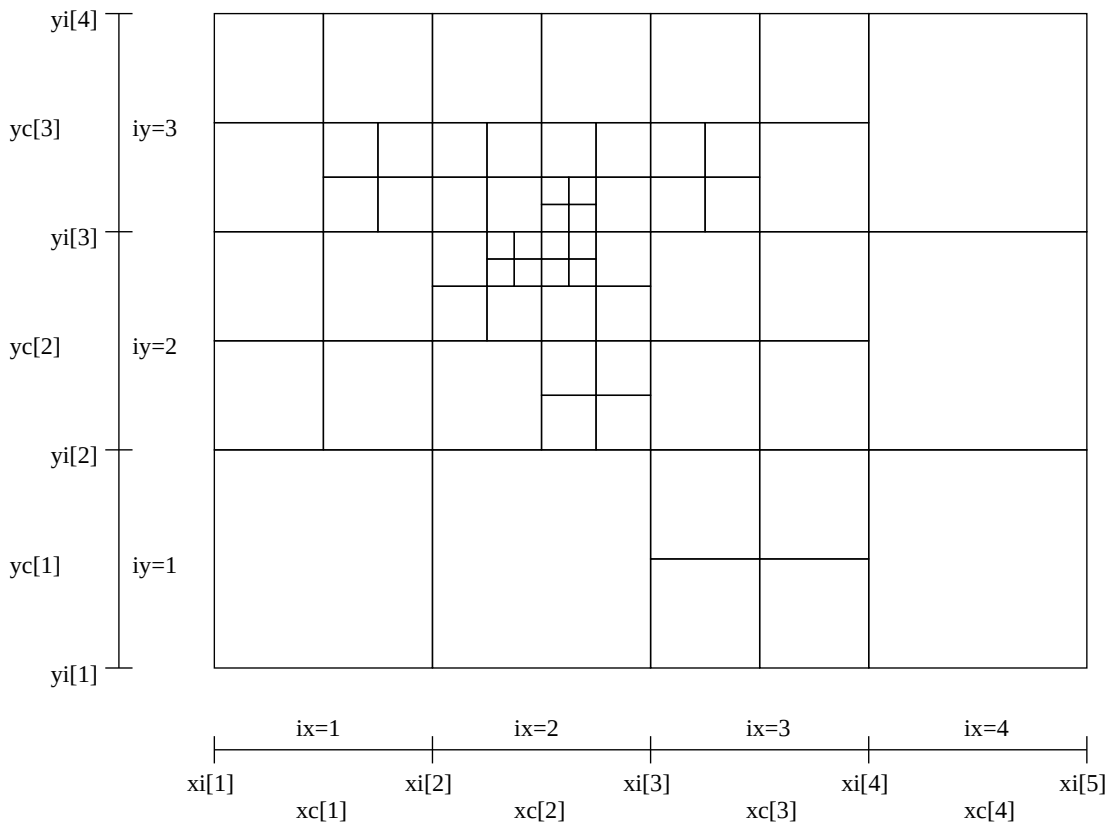


Figure 10.2. Example of a 2-D grid with oct-tree refinement. The base grid has nx=4 and ny=3. Three levels of refinement are added to this base grid.

10.3 Oct-tree Adaptive Mesh Refinement

An oct-tree refinened grid is called “grid style 1” in RADMC-3D. It can be used in Cartesian coordinates as well as in spherical coordinates (Section 5.2).

You start from a normal regular base grid (see Section 10.1), possibly even with “separable refinement” (see Section 10.2). You can then split some of the cells into 2x2x2 subcells (or more precisely: in 1-D 2 subcells, in 2-D 2x2 subcells and in 3-D 2x2x2 subcells). If necessary, each of these 2x2x2 subcells can also be split into further subcells. This can be repeated as many times as you wish until the desired grid refinement level is reached. Each refinement step refines the grid by a factor of 2 in linear dimension, which means in 3-D a factor of 8 in volume. In this way you get, for each refined cell of the base grid, a tree of refinement. The base grid can have any size, as long as the number of cells in each direction is an even number. For instance, you can have a 6x4 base grid in 2-D, and refine cell (1,2) by one level, so that this cell splits into 2x2 subcells.

Note that it is important to set which dimensions are “active” and which are “non-active”. For instance, if you have a 1-D model with 100 cells and you tell RADMC-3D (see Section A.2.2) to make a base grid of 100x1x1 cells, but you still keep all three dimensions “active” (see Section A.2.2), then a refinement of cell 1 (which is actually cell (1,1,1)) will split that cell into 2x2x2 subcells, i.e. it will also refine in y and z direction. Only if you explicitly switch the y and z dimensions off the AMR will split it into just 2 subcells.

Oct-tree mesh refinement is very powerful, because it allows you to refine the grid exactly there where you need it. And because we start from a regular base grid like the grid specified in Section 10.1, we can start designing our model on a regular base grid, and then refine where needed. See Fig. 10.2

The AMR stand for “Adaptive Mesh Refinement”, which may suggest that RADMC-3D will refine internally. At the moment this is not yet the case. The “adaptive” aspect is left to the user: he/she will have to “adapt” the grid such that it is sufficiently refinened where it is needed. In the future we may allow on-the-fly adaption of the grid, but that is not yet possible now.

One problem with oct-tree AMR is that it is difficult to handle such grids in external plotting programs, or even in

programs that set up the grid. While it is highly flexible, it is not very user-friendly. Typically you may use this oct-tree refinement either because you import data from a hydrodynamics code that works with oct-tree refinement (e.g. FLASH, RAMSES), or when you internally refine the grid using the `userdef_module.f90` (see Chapter 13). In the former case you are anyway forced to manage the complexities of AMR, while in the latter case you can make use of the AMR modules of RADMC-3D internally to handle them. But if you do not need to full flexibility of oct-tree refinement and want to use a simpler kind of refinement, then you can use RADMC-3D's alternative refinement mode: the layer-style AMR described in Section 10.4 below.

The precise way how to set up such an oct-tree grid using the `amr_grid.inp` file is described in Section A.2.2. The input of any spatial variables (such as e.g. the dust density) uses the sequence of grid cells in the same order as the cells are specified in that `amr_grid.inp` file.

10.4 Layered Adaptive Mesh Refinement

A layer-style refinened grid is called "grid style 10" in RADMC-3D. It can be used in Cartesian coordinates as well as in spherical coordinates (Section 5.2).

This is an alternative to the full-fledged oct-tree refinement of Section 10.3. The main advantage of the layer-style refinement is that it is far easier to handle by the human brain, and thus easier for model setup and the analysis of the results.

The idea here is that you start again with a regular grid (like that of Section 10.1), but you can now specify a rectangular region which you want to refine by a factor of 2. The way you do this is by choosing the starting indices of the rectangle and specifying the size of the rectangle by setting the number of cells in each direction from that starting point onward. For instance, setting the starting point at (2,3,1) and the size at (1,1,1) will simply refine just cell (2,3,1) of the base grid into a set of 2x2x2 sub-cells. But setting the starting point at (2,3,1) and the size at (2,2,2) will split cells (2,3,1), (3,3,1), (2,4,1), (3,4,1), (2,3,2), (3,3,2), (2,4,2) and (3,4,2) each into 2x2x2 subcells. This in fact is handled as a 4x4x4 regular sub-grid patch. And setting the starting point at (2,3,1) and the size at (4,6,8) will make an entire regular sub-grid patch of 8x12x16 cells. Such a sub-grid patch is called a *layer*.

The nice thing of these layers is that each layer (i.e. subgrid patch) is handled as a regular sub-grid. The base grid is layer number 0, and the first layer is layer number 1, etc. Each layer (including the base grid) can contain multiple sub-layers. The only restriction is that each layer fits entirely inside its parent layer, and layers with the same parent layer should not overlap. Each layer can thus have one or more sub-layers, each of which can again be divided into sub-layers. This builds a tree structure, with the base layer as the trunk of the tree (this is contrary to the oct-tree structure, where each base grid *cell* forms the trunk of its own tree). In Fig. 10.3 an example is shown of two layers with the same parent (= layer 0 = base grid), while in Fig. 10.4 an example is shown of two nested layers.

If you now want to specify data on this grid, then you simply specify it on each layer separately, as if each layer is a separate entity. Each layer is treated as a regular grid, irrespective of whether it contains sub-layers or not. So if we have a base grid of 4x4x4 grid cells containing two layers: one starting at (1,1,1) and having (2,2,2) size and another starting at (3,3,3) and having (1,1,2) size, then we first specify the data on the 4^3=64 base grid, then on the $(2*2)^3$=64 grid cells of the first layer and then on the 2x2x4=16 cells of the second layer. Each of these three layers are regular grids, and the data is inputted/outputted in the same way as if these are normal regular grids (see Section 10.1). But instead of just one such regular grid, now the data file (e.g. `dust_density.inp`) will contain three successive lists of numbers, the first for the base grid, the second for the first layer and the last for the second layer. You may realize at this point that this will introduce a redundancy. See Subsection 10.4.1 for a discussion of this redundancy.

The precise way how to set up such an oct-tree grid using the `amr_grid.inp` file is described in Section A.2.3. The input of any spatial variables (such as e.g. the dust density) uses the sequence of grid cells in the same order as the cells are specified in that `amr_grid.inp` file.

10.4.1 On the "successively regular" kind of data storage, and its slight redundancy

With the layered grid refinement style there will be *redundant* data in the data files (such as e.g. the `dust_density.inp` file. Each layer is a regular (sub-)grid and the data will be specified in each of these grid cells of that regular (sub-)grid. If then some of these cells are overwritten by a higher-level layer, these data are then redundant. We could

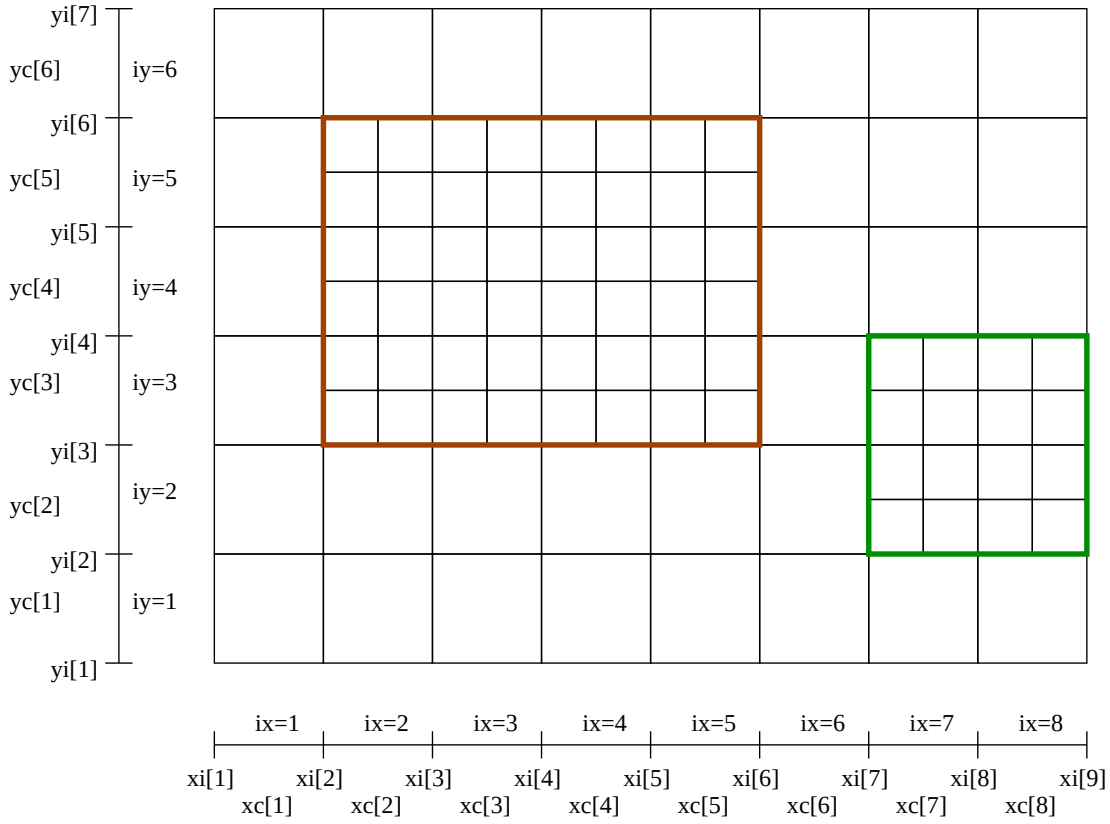


Figure 10.3. Example of a 2-D base grid with nx=4 and ny=3, with two AMR-layers added to it. This example has just one level of refinement, as the two layers (brown and gree) are on the same level (they have the same parent layer = layer 0).

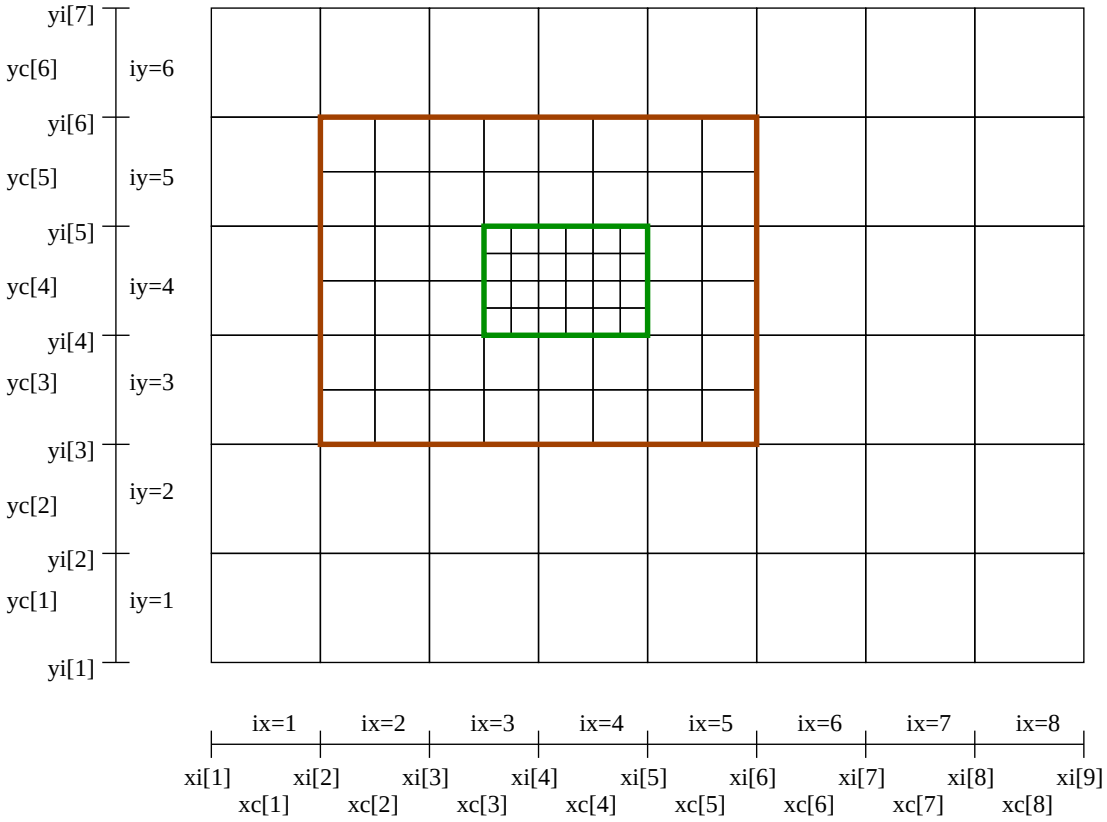


Figure 10.4. Example of a 2-D base grid with nx=4 and ny=3, with two nested AMR-layers added to it. This example has two levels of refinement, as layer 1 (brown) is the parent of layer 2 (green).

of course have insistent that only the data in those cells that are not refined by a layer should be written to (or read from) the data files. But this would require quite some clever programming on the part of the user to a-priori find out where the layers are and therefore which cells should be skipped. We have decided that it is far easier to just insist that each layer (including the base grid, which is layer number 0) is simply written to the data file as a regular block of data. The fact that some of this data will be not used (because they reside in cells that are refined) means that we write more data to file than really exists in the model. This makes the files larger than strictly necessary, but it makes the data structure by far easier. Example: suppose you have a base grid of 8x8x8 cells and you replace the inner 4x4x4 cells with a layer of 8x8x8 cells (each cell being half the size of the original cells). Then you will have for instance a `dust_density.inp` file containing 1024 values of the density: 8^3=512 values for the base grid and again 8^3=512 values for the refinement layer. Of the first 8^3=512 values 4^3=64 values are ignored (they could have any value as they will not be used). The file is thus 64 values larger than strictly necessary, which is a redundancy of 64/1024=0.0625. If you would have used the oct-tree refinement style for making exactly the same grid, you would have only 1024-64=960 values in your file, making the file 6.25% smaller. But since 6.25% is just a very small difference, we decided that this is not a major problem and the simplicity of our "successively regular" kind of data format is more of an advantage than the 6.25% redundance is a disadvantage.

10.5 Unstructured grids

In a future version of RADMC-3D we will include unstructured gridding as a possibility. But at this moment such a gridding is not yet implemented.

10.6 1-D Plane-parellel grid

Sometimes it can be useful to make simple 1-D plane parallel models, for instance if you want to make a simple 1-D model of a stellar atmosphere. RADMC-3D does *not* have a 1-D plane-parallel mode. *But* you can simulate a plane-parallel mode by making a 1-D spherically symmetric model in which you make, for instance, a radial grid in which $r_{\rm nr}/r_1 - 1 \ll 1$. An example: $r = \{10000.0, 10000.1, 10000.2, \cdots, 10001.0\}$. This is not perfectly plane-parallel, but sufficiently much so that the difference is presumably indiscernable. The spectrum is then automatically that of the entire large sphere, but by dividing it by the surface area, you can recalculate the local flux. In fact, since a plane-parallel model usually is meant to approximate a tiny part of a large sphere, this mode is presumably even more realistic than a truly 1-D plane-parallel model.

Chapter 11

More information about the treatment of stars

How stars are treated in RADMC-3D is perhaps something that needs some more background information. This is the structure:

1. *Stars as individual objects:*
   The most standard way of injecting stellar light into the model is by putting one or more individual stars in the model. A star can be placed anywhere, both inside the grid and outside. The main input file specifying their location and properties is: `stars.inp`. The stars can be treated in two different ways, depending on the setting of the variable `stars_sphere` that can be set to 0 or 1 in the file `radmc3d.inp` file.
   - The default is to treat stars as zero-size point sources. This is the way it is done if (as is the default) `stars_sphere=0`. The stars are then treated as point sources in spite of the fact that their radius is specified as non-zero in the `stars.inp` file. This default mode is the easiest and quickest. For most purposes it is perfectly fine. Only if you have material very close to a stellar surface it may be important to treat the finite size(s) of the star(s).
   - If `stars_sphere=1` in the `radmc3d.inp` file, then all stars are treated as spheres, their radii being the radii specified in the `stars.inp` file. This mode can be tricky, so please read Section 11.2.
2. *Smooth distributions of zillions of stars:*
   For modeling galaxies or objects of that size scale, it is of course impossible and unnecessary to treat each star individually. So *in addition to the individual stars* you can specify spatial distributions of stars, assuming that the number of stars is so large that there will always be a very large number of them in each cell. Please note that using this possibility does *not* exclude the use of individual stars as well. For instance, for a galaxy you may want to have distributions of unresolved stars, but one single "star" for the active nucleus and perhaps a few individual "stars" for bright star formation regions or O-star clusters or so. The distribution of stars is described in Section 11.3.
3. *An external "interstellar radiation field":*
   Often an object is affected not only by the stellar radiation from the stars inside the object itself, but also by the diffuse radiation from the many near and far stars surrounding the object. This "Interstellar Radiation Field" can be treated by RADMC-3D as well. This is called the "external source" in RADMC-3D. It is described in Section 11.4.

11.1 Stars treated as point sources

By default the stars are treated as point-sources. Even if the radius is specified as non-zero in the `stars.inp` file, they are still treated as points. The reason for this is that it is much easier and faster for the code to treat them as point-sources. Point sources cannot occult anything in the background, and nothing can partly occult them (they are

only fully or not occulted, of course modulo optical depth of the occulting object). This approximation is, however, not valid if the spatial scales you are interested in are not much larger (or even the same or smaller) than the size of the star. For instance, if we are interested in modeling the radiative transfer in a disk around a Brown Dwarf, where dust can survive perhaps even all the way down to the stellar surface, we must take the non-point-like geometry of the star into account. This is because due to its size, the star can shine *down* onto the disk, which would not be possible if the star is treated as a point source. However, for a dust disk arounda Herbig Ae star, where the dust evaporation radius is at about 0.5 AU, the star can be treated as a point-source without problems.

So if you just use RADMC-3D as-is, or if you explicitly set `stars_sphere = 0` in the file `radmc3d.inp`, then the stars are all treated as point sources.

11.2 Stars treated as spheres

For problems in which the finite geometrical size of the star (or stars) is/are important, RADMC-3D has a mode by which the stars are treated as spheres. This can be necessary for instance if you model a disk around a Brown Dwarf, where the dusty disk goes all the way down to the stellar surface. The finite size of the star can thus shine *down* onto the disk, but only if its finite size is treated as such. In the default point-source approximation the surface layers of such a disk would be too cold, because this "shining down onto the disk" phenomenon is not treated.

You can switch this mode on by setting `istar_sphere = 1` in the file `radmc3d.inp`. Note that no limb darkening or brightening is included in this mode, and currently RADMC-3D does not have such a mode available.

This mode is, however, somewhat complex. A sphere can partly overlap the grid, while being partly outside the grid. A sphere can also overlap multiple cells at the same time, engulfing some cells entirely, while only partly overlapping others. The correct and fast treatment of this makes the code a bit slower, and required some complex programming. So the user is at the moment advised to use this mode only if necessary and remain aware of possible errors for now (as of version 0.17).

For the Monte Carlo simulations the finite star size means that photon packages are emitted from the surface of the sphere of the star. It also means that any photon that re-enters the star during the Monte Carlo simulation is assumed to be lost.

11.3 Distributions of zillions of stars

For models of galaxies it is important to be able to have distributed stellar sources instead of individual stars. The way to implement this in a model for RADMC-3D is to

1. Prepare one or more *template stellar spectra*, for instance, one for each stellar type you wish to include. These must be specified in the file `stellarsrc_templates.inp` (see Section A.8). Of course the more templates you have, the more memory consuming it becomes, which is of particular concern for models on large grids. You can of course also take a sum of various stellar types as a template. For instance, if we wish to include a 'typical' bulge stellar component, then you do not need to treat each stellar type of bulge stars separately. You can take the 'average spectrum per gram of average star' as the template and thus save memory.
2. For each template you must specify the *spatial distribution*, i.e. how many stars of each template star are there per unit volume in each cell. The stellar density is, in fact, given as gram-of-star/cm^3 (i.e. not as number density of stars). The stellar spatial densities are specified in the file `stellarsrc_density.inp` (see Section A.9).

Note that if you have a file `stellarsrc_templates.inp` in your model directly, then the stellar sources are automatically switched on. If you do not want to use them, then you must delete this file.

The smooth stellar source distributions are nothing else than source functions for the radiative transfer with the spectral shape of the template stellar spectra from the `stellarsrc_templates.inp`. You will see that if you make a spectrum of your object, then even if the dust temperature etc is zero everywhere, you still see a spectrum: that of the stellar template(s). In the Monte Carlo simulations these stellar templates act as net sources of photons, that subsequently move through the grid in a Monte Carlo way.

Note that the smooth stellar source distributions assume that the zillions of stars that they represent are so small that they do not absorb any appreciable amount of radiation. They are therefore pure sources, not sinks.

11.4 The interstellar radiation field: external source of energy

You can include an *isotropic* interstellar radiation field in RADMC-3D. This will take effect both in the making of spectra and images, as well as in the Monte Carlo module.

The way to activate this is to make a file `external_source.inp` and fill it with the information needed (see Section A.10).

11.4.1 Role of the external radiation field in Monte Carlo simulations

For the Monte Carlo simulations this means that photons may be launched from outside inward. The way that this is done is that RADMC-3D will make a sphere around the entire grid, just large enough to fit in the entire grid but not larger. Photon packages can freely leave this sphere. But if necessary, photon packages can be launched from this sphere inward. RADMC-3D will then calculate the total luminosity of this sphere, which is $L = 4\pi^2 I r^2_{\rm sphere}$ where I is the intensity. For monochromatic Monte Carlo it is simply $I = I_\nu$, while for the thermal Monte Carlo it is $I = \int_0^\infty I_\nu d\nu$, where I_ν is the intensity as specified in the file `external_source.inp`. Note that if the sphere would have been taken larger, then the luminosity of the external radiation field would increase. This may seem anti-intuitive. The trick, however, is that if the sphere is larger, then also more of these interstellar photons never enter the grid and are lost immediately. That is why it is so important that RADMC-3D makes the sphere as small as possible, so that it limits the number of lost photon packages. It also means that you, the user, would make the grid much larger than the object you are interested in, then RADMC-3D is forced to make a large sphere, and thus potentially many photons will get lost: they may enter the outer parts of the grid, but there they will not get absorbed, nor will they do much.

In fact, this is a potential difficulty of the use of the external sources: since the photon packages are lauchned from outside-inward, it may happen that only few of them will enter in the regions of the model that you, the user, are interested in. For instance, you are modeling a 3-D molecular cloud complex with a few dense cold starless cores. Suppose that no stellar sources exist in this model, only the interstellar radiation field. The temperature in the centers of these starless cores will be determined by the interstellar radiation field. But since the cores are very small compared to the total model (e.g. you have used AMR to refine the grid around/in these cores), the chance of each external photon package to 'hit' the starless core is small. It means that the larger the grid or the smaller the starless core, the more photon packages (`nphot`, see Section 6.1) one must use to make sure that at least some of them enter the starless cores. If you choose `nphot` too small in this case, then the temperature in these cores would remain undetermined (i.e. they will be zero in the results).

11.4.2 Role of the external radiation field in images and spectra

The interstellar radiation field also affects the images and spectra that you make. Every ray will start at minus-infinity with the intensity given by the external radiation field, instead of 0 as it would be if no external radiation field is specified. If you make an image, the background of your object will then therefore not be black. You can even make silhouette images like those of the famous silhouette disks in Orion.

But there is a danger: if you make spectra, then also the background radiation is inside the beam, and will thus contribute to the spectrum. In fact, the larger you make the beam the more you will pick up of the background. This could thus lead to the spectrum of your source to be swamped by the background if you do not specify a beam in the spectrum.

Chapter 12

Using RADMC-3D in child mode (optional)

For large datasets it may take a considerable amount of time for RADMC-3D to read the entire dataset into memory. Having to wait such a long time for every action one wants to take (e.g. re-making an image at a different vantage point or a different wavelength) quickly gets on one's nerves and strongly inhibits the interactivity between RADMC-3D and the user. The "child mode" is designed to circumvent this problem. It allows RADMC-3D to be started as a child process of another process, stay in memory (with all the data loaded once-and-for-all) for as long as the parent process lives, and have communication with its parent process via a bi-way pipe. A bi-way pipe is like a file to which you can write or from which you can read. Your parent process, which calls RADMC-3D as a child, can give RADMC-3D the command to do something by writing to the pipe file unit, and then receiving the results from RADMC-3D by reading from that same file unit.

The IDL program `viewimage.pro` (see chapter 14), which is part of the RADMC-3D package, calls RADMC-3D as a child process and communicates with it in precisely this way.

NOTE: Currently there appears to be a problem when trying to use RADMC-3D in child mode on some systems. For instance, viewimage may freeze. This appears to be a problem with buffering of the standard I/O unit. I have been trying to figure out what causes this, and particularly, why it happens on some machines and not on other machines (in fact it happened on one Macbook but not on another, while the systems were seemingly identical). I will continue to work on this. When calling viewimage and the thing freezes, try calling viewimage,/nochild. That is slower, but should work.

In IDL this is done with the keyword `unit=unit` in the `spawn` command. For instance, in viewimage.pro there is a line

```
spawn,'nice radmc3d child',unit=iounit
```

The `nice` is simply to let RADMC-3D run at a low priority under Linux or Mac OS X, the `child` command line option is a RADMC-3D specific command line option that tells RADMC-3D that it should not exit after its first action, but wait further orders. The `unit=iounit` gets the file unit through which we can communicate with RADMC-3D. Of course, by virtue of the fact that RADMC-3D is called by IDL in the first place, it is naturally IDL's child process. But by asking RADMC-3D not to exit after the first action, and by getting the file unit from the keyword `unit=`, RADMC-3D will wait for our commands and only exit when we tell it so by giving it the command `quit`.

The way we can communicate with RADMC-3D is by writing to the file `iounit` commands like the ones on the command line. But contrary to the normal command line, they are now given one word per line. For instance, to let RADMC-3D make an image at wavelength number `ilambda` (from IDL):

```
printf,iounit,'image'
printf,iounit,'npix'
printf,iounit,strcompress(string(npix),/remove_all)
printf,iounit,'ilambda'
printf,iounit,strcompress(string(ilambda),/remove_all)
printf,iounit,'incl'
printf,iounit,strcompress(string(incl),/remove_all)
printf,iounit,'phi'
printf,iounit,strcompress(string(phi),/remove_all)
printf,iounit,'enter'
```

The `enter` is meant to tell RADMC-3D: now go and do your work. In fact, you rarely have to interact like this with RADMC-3D youself, because you can use the IDL subroutine `makeimage()` in the `readradmc.pro` file that does all these things for you; just set the keyword `iounit` to the value you obtained from starting `radmc3d` in child mode.

In child mode the results from `radmc3d` are not returned immediately to the parent. To ask `radmc3d` for the results of its latest calculation (for instance an image), you do:

```
printf,iounit,'writeimage'
```

followed by

```
iformat=0
nx=0
ny=0
nf=0
readf,iounit,iformat
readf,iounit,nx,ny
readf,iounit,nf
readf,iounit,sizepix_x,sizepix_y
lambda=dblarr(nf)
readf,iounit,lambda
image=fltarr(nx,ny,nf)
readf,iounit,image
```

But again you don't have to do this complex stuff yourself. Instead this is done for you by the `readimage()` IDL routine in the `readradmc.pro` file, again when `iounit` is specified.

So IDL users can simply do (in IDL):

```
.r readradmc
;
; Start RADMC-3D
;
spawn,'nice radmc3d child',unit=iounit
;
; Make an image
;
makeimage,incl=45.,phi=10.,npix=200,ifreq=10,iounit=iounit
;
; Read the image from RADMC-3D
;
a=readimage(iounit=iounit)
;
; Plot the image on the screen
;
plotimage,a
;
; ...and here many more images or spectra...
; ....
; ....
;
; And when we are done, quit RADMC-3D and free the file number
;
printf,iounit,'quit'
close,iounit
free_lun,iounit
```

Tip: If you use RADMC-3D in child mode, then all the usual output that normally would go to screen will now be redirected to a separate file called `radmc3d.out`. *This is useful for debugging the code when using it in child mode. So if RADMC-3D fails somehow when in child mode, then have a look at* `radmc3d.out` *to see what went wrong.*

Chapter 13

Using the userdef_module.f90 file for internal model setup (optional)

It has been mentioned several times before that as an alternative to the standard 'compile once-and-for-all' philosophy, one can also use RADMC-3D by modifying the code directly so that `radmc3d` will have new functionality that might be of use for you. We refer to Section 4.5 for an in-depth description of how to modify the code in a way that is *non-invasive* to the main code. We urge the reader to read Section 4.5 first before continuing to read this chapter. In all of the following we assume that the editings to the fortran files are done in the local way described in Section 4.5 so that the original source files in the `src/` directory stay unaffected, and only local copies are edited.

The most common reason for editing the code itself is for setting up the model *internally* rather than reading in all data via input files. This is what this chapter is about. For a list of advantages and disadvantages of setting models up internally as opposed to the standard way, see Section 13.2 below. This is done by editing the file `userdef_module.f90`. This file contains a set of standard subroutines that are called by the main program at special points in the code. Each subroutine has a special purpose which will be described below. By keeping a subroutine empty, nothing is done. By filling it with your own code lines, you can set up the density, temperature or whatever needs to be set up for the model. In addition to this you can do the following as well:

- Add new variables or arrays in the module header (above the `contains` command), which you can use in the subroutines of the `userdef_module.f90` module. You are completely free to add any new variables you like. A small tip: it may be useful (though not required) to start all their names with e.g. `userdef_` to make sure that no name conflicts with other variables in the code happen.
- Add new subroutines at will (below the `contains` command) which you can call from within the standard subroutines.
- Introduce your own `radmc3d` command-line options (see Section 13.1).
- Introduce your own `radmc3d.inp` namelist variables (see Section 13.1).

13.1 The pre-defined subroutines of the userdef_module.f90

The idea of the `userdef_module.f90` is that it contains a number of standard pre-defined subroutines that are called from the `main.f90` code (and *only* from there). Just browse through the `main.f90` file and search for the sequence "`call userdef_`" and you will find all the points where these standard routines are called. It means that at these points you as the user have influence on the process of model setup. Here is the list of standard routines and how they are used. They are ordered roughly in chronological order in which they are called.

- `userdef_defaults()`
  This subroutine allows you to set the default value of any new parameters you may have introduced. If neither on the command line nor in the `radmc3d.inp` file the values of these parameters are set, then they will simply retain this default value.

- `userdef_commandline(buffer,numarg,iarg,fromstdi,gotit)`
  This subroutine allows you to add your own command-line options for `radmc3d`. The routine has a series of standard arguments which you are not allowed to change. The `buffer` is a string containing the current command line option that is parsed. You will check here if it is an option of your module, and if yes, activate it. An example is listed in the code. You an also require a second argument, for which also an example is listed in the original code.

- `userdef_commandline_postprocessing()`
  After the command line options have been read, it can be useful to check if the user has not asked for conflicting things. Here you can do such checks.

- `userdef_parse_main_namelist()`
  Here you can add your own namelist parameters that read from the `radmc3d.inp` file. An example is provided in the original code.

- `userdef_main_namelist_postprocessing()`
  Also here, after the entire `radmc3d.inp` file has been read and interpreted, you can do some consistency checks and postprocessing here.

- `userdef_prep_model()`
  This routine can be used if you wish to set up the grid not from input files but internally. You will have to know how to deal with the `amr_module.f90` module. You can also set your own global frequency grid here. And finally, you can set your own stellar sources here. In all cases, if you set these things here (which requires you to make the proper memory allocations, or in case of the gridding, let the `amr_module.f90` do the memory allocations for you) the further course of `radmc3d` will skip any of its own settings (it will simply detect if these arrays are allocated already, and if yes, it will simply not read or allocate them anymore).

- `userdef_setup_model()`
  This is the place where you can actually make your own model setup. By the time this subroutine is called, all your parameters have been read in, as well as all of the other parameters from the original `radmc3d` code. So you can now set up the dust density, or the gas velocity or you name it. For all of these things you will have to allocate the arrays youself (!!!). Once you did this, the rest of the `radmc3d` code won't read those data anymore, because it detects that the corresponding arrays have already been allocated (by you). This allows you to completely circumvent the reading of any of the following files by making these data yourself here at this location:

  - `amr_grid.inp` or `amr_grid.uinp` or in the future the input files for any of the other griding types.
  - `dust_density.inp` or `dust_density.uinp`
  - `dust_temperature.dat` or `dust_temperature.udat`
  - `gas_density.inp` or `gas_density.uinp`
  - `gas_temperature.inp` or `gas_temperature.uinp`
  - `gas_velocity.inp` or `gas_velocity.uinp`
  - `microturbulence.inp` or `microturbulence.uinp`
  - `levelpop_XXX.inp` or `levelpop_XXX.uinp`
  - `numberdens_XXX.inp` or `numberdens_XXX.uinp`

  To learn how to set up a model in this way, we refer you for now to the `ioput_module.f90` or `lines_module.f90` and search for the above file names to see how the arrays are allocated and how the data are inserted. I apologise for not explaining this in more detail at this point. But examples are or will be given in the `examples/` directory.

- `userdef_dostuff()`
  This routine will be called by the main routine to allow you to do any kind of calculation after the main calculation (for instance after the monte carlo simulation). This is done within the execution-loop, so that if you use RADMC-3D in child mode, this routine will be called after each calculation.

- `userdef_myaction()`
  If RADMC-3D is called as `radmc3d myaction`, then the user-defined routine `userdef_myaction()` is called, just like the spectrum making routine is called if you type `radmc3d sed`. This allows the user to make RADMC-3D do special things on demand. Note that this can be used in combination with many of the above subroutines to interpret command-line options and `radmc3d.inp` entries. *Not yet working in version 0.15.*

- `userdef_compute_levelpop()`
  This is a subroutine that can be called by the camera module for on-the-fly calculation of level populations according to your own recipe. This may be a bit tricky to use, but I hope to be able to provide some example(s) in the near future.

- `userdef_writemodel()`
  This allows the user to dump any stuff to file that the user computed in this module. You can also use this routine to write out files that would have been used normally as input file (like `amr_grid.inp` or `dust_density.inp`) so that the IDL routines can read them if they need. In particular the grid information may be needed by these external analysis tools. Here is a list of standard subroutines you can call for writing such files:

  - `write_grid_file()`
  - `write_dust_density()`
  - ...more to come...

- `userdef_reset_flags()`
  If the user wants some flags to be reset after each command (in the child mode, see Chapter 12), then here it can be done.

For now this is it, more routines will be included in the future.

Note that the `userdef_compute_levelpop()` subroutine, in contrast to all the others, is called not from the `main.f90` program but from the `camera_module.f90` module. This is why the camera module is the only module that is higher in compilation ranking than the userdef module (i.e. the userdef module will be compiled before the camera module). For this reason the userdef module has no access to the variables of the camera module. For the rest, the userdef module has access to the variables in all other modules.

Note also that not all input data is meant to be generated in this way. The following types of data are still supposed to be read from file:

- Dust opacity data
- Molecular fundamental data

Please have a look in the `examples/` directory for models which are set up in this internal way.

13.2 Some caveats and advantages of internal model setup

Setting up the models internally has several advantages as well as disadvantages compared to the standard way of feeding the models into `radmc3d` via files. The advantages are, among others:

- You can modify the model parameters in `radmc3d.inp` and/or in the command line options (depending on how you allow the user to set these parameters, i.e. in the `userdef_parse_main_namelist()` routine and/or in the `userdef_commandline()` routine. You then do not need to run IDL anymore (except for setting up the basic files; see examples). Some advantages of this:

  1. It allows you, for instance, to create a version of the `radmc3d` code that acts as if it is a special-purpose model. You can specify model parameters on the command line (rather than going through the cumbersome IDL stuff).

2. It is faster: even a large model is built up quickly and does not require a long read from large input files.

- You can make use of the AMR module routines such as the `amr_branch_refine()` routine, so you can adaptively refine the grid while you are setting up the model.

Some of the disadvantages are:

- The model needs to be explicitly written out to file and read into IDL or any other data plotting package before you can analyze the density structure to test if you've done it right. You can explicitly ask `./radmc3d` to call the `userdef_writemodel()` subroutine (which is supposed to be writing out all essential data; but that is the user's responsibility) by typing `./radmc3d writemodel`.
- Same is true for the grid, and this is potentially even more dangerous if not done. You can explicitly ask `./radmc3d` to write out the grid file by typing `./radmc3d writegridfile`. Note that if you call the `write_grid_file()` subroutine from within `userdef_writemodel()`, then you do not have to explicitly type `./radmc3d writegridfile` as well. Note also that `radmc3d` will automatically call the `write_grid_file()` subroutine when it writes the results of the thermal Monte Carlo computation, iff it has its grid from inside (i.e. it has not read the grid from the file `amr_grid.inp`.
- It requires a bit more knowledge of the internal workings of the `radmc3d` code, as you will need to directly insert code lines in the `userdef_module.f90` file.

13.3 Using the userdef module to compute integrals of J_ν

With the monochromatic Monte Carlo computation (see Section 6.4) we can calculate the mean intensity J_ν at every location in the model at a user-defined set of wavelengths. However, as mentioned before, for large models and large numbers of wavelengths this could easily lead to a data volume that is larger than what the computer can handle. Since typically the main motivation for computing J_ν is to compute some integral of the the form:

$$Q = \int_0^\infty J_\nu K_\nu d\nu \tag{13.1}$$

where K_ν is some cross section function or so, it may not be necessary to store the entire function J as a function of nu. Instead we would then only by interested in the result of this integral at each spatial location.

So it would be useful to allow the user to do this computation internally. We should start by initializing $Q(x, y, z) = 0$ (or $Q(r, \theta, \phi) = 0$ if you use spherical coordinates). Then we call the monochromatic Monte Carlo routine for the first wavelength we want to include, and multiply the resulting mean intensities with an appropriate $\Delta\nu$ and add this to $Q(x, y, z)$. Then we do the monochromatic Monte Carlo for the next wavelength and again add to Q everywhere. We repeat this until our integral (at every spatial location on the grid) is finished, and we are done. This saves a huge amount of memory.

Since this is somewhat hard to explain in this PDF document, we refer to the example model `run_example_jnu_integral/`.

STILL IN PROGRESS.

13.4 Some tips and tricks for programming user-defined subroutines

Apart from the standard subroutines that *must* be present in the `userdef_module.f90` file (see Section 13.1), you are free to add any subroutines or functions that you want, which you can call from within the predefined subroutines of Section 13.1. You are completely free to expand this module as you wish. You can add your own variables, your own arrays, allocate arrays, etc.

Sometimes you may need to know "where you are" in the grid. For instance, the subroutine `userdef_compute_levelpop()` is called with an argument `index`. This is the index of the current cell from within which the subroutine has been called. You can now address, for instance, the dust temperature at this location:

```
temp = dusttemp(1,index)
```

(for the case of a single dust species). You may also want to know the coordinates of the center of the cell. For this, you must first get a pointer to the AMR-tree structure of this cell. The pointer `b` is declared as

```
type(amr_branch), pointer :: b
```

Then you can point the pointer to that cell structure

```
b => amr_index_to_leaf(index)%link
```

And now you can get the x,y,z-coordinates of the center of the cell:

```
xc = amr_finegrid_xc(b%ixyzf(1),1,b%level)
yc = amr_finegrid_xc(b%ixyzf(2),2,b%level)
zc = amr_finegrid_xc(b%ixyzf(3),3,b%level)
```

Or the left and right cell walls:

```
xi_l = amr_finegrid_xi(b%ixyzf(1),1,b%level)
yi_l = amr_finegrid_xi(b%ixyzf(2),2,b%level)
zi_l = amr_finegrid_xi(b%ixyzf(3),3,b%level)
xi_r = amr_finegrid_xi(b%ixyzf(1)+1,1,b%level)
yi_r = amr_finegrid_xi(b%ixyzf(2)+1,2,b%level)
zi_r = amr_finegrid_xi(b%ixyzf(3)+1,3,b%level)
```

Chapter 14

Model analysis (I): The IDL model analysis tool set

While the code RADMC-3D is written in fortran-90, there is an extensive set of tools written in IDL that make it easier for the user to set up models and interpret results. See Section 4.4 for where they are and how they can be properly installed so that they are easy to use. Here we describe these tools.

14.1 The readradmc.pro tools

The `readradmc.pro` program file contains a series of subroutines for reading RADMC-3D output into IDL so that the user can do post-processing and analysis on these data. The file also contains subroutines for operating RADMC-3D directly from within IDL.

14.1.1 Function readimage

The `readimage()` function reads the latest produced image into IDL. This image is (i.e. should be) located in the file `image.out`, which is produced by RADMC-3D. The `readimage()` function returns an IDL structure containing the image (be it a single-frequency image or a multi-frequency image) in units of intensity (erg/s/cm^2/Hz/ster), as well as information about the pixel grid and at which frequency(ies) the image was taken. With the "help" command you can see the full contents of the returned structure:

```
.r readradmc.pro
a=readimage()
help,a,/str
```

This will show you the contents of the structure. Here is a quick summary of these contents:

`nx, ny`: The number of pixels in x- and y- direction in the image

`nrfr`: The number of frequencies (wavelengths), i.e. the number of images at different wavelengths

`sizepix_x, _y`: The size of the pixels in x- and y- directions in units of centimeters. This is of course only possible for images at semi-infinity (the default). For images made as a local observer, see Section 9.10 for details.

`image`: The nx$\times$ny array of intensities of the image. If multiple colors (wavelengths, frequencies) are present, then the `image` array will be three-dimensional: nx$\times$ny$\times$nrfr. The intensities are in units of erg/cm^2/s/Hz/ster.

`flux`: The integral of the intensity over the entire image, i.e. the flux in the image. The units are erg/cm^2/s/Hz for an observer at 1 parsec distance. This "1 parsec" is just a normalization distance. If you make images of objects much larger than 1 parsec in size, this does *not* mean that the image is made by a local observer (unless explicitly specified, see Section 9.10). It is just so that you can compute the actually observed flux

by multiplying the flux by a factor $(\mathrm{pc}/d)^2$, where d is the true distance of the observer (which must then be much larger than the size of the object to keep the far-field limit valid).

`x,y`: The actual image coordinates in cm (unless local observer, see Section 9.10). These are in principle redundant because you can calculate them yourself from the `sizepix_x, _y` and `nx, ny` values. They are here just for convenience.

`lambda`: The wavelength (in micron) at which this image was made. For multiple colors/freqs/wavelengths this is an array.

The `readimage()` function can also be used to read from a pipe between IDL and RADMC (see "child mode" in Chapter 12). One then gives it as an argument the file number of the pipe (see example shown in Chapter 12). This is in fact what is done by `viewimage.pro` below.

14.1.2 Subroutine plotimage

The subroutine `plotimage` plots the image read by `readimage()` to screen (or postscript file) in a proper way. Check out the following example (to be executed only after the dust temperatures have been written to file `dust_temperature.dat`):

```
.r readradmc.pro
a=readimage()
plotimage,a,/au
```

This subroutine is in fact used by the `viewimage.pro` below to display images on the drawing pane of the GUI widget. The `plotimage` subroutine has a large number of optional arguments:

`/au` or `/pc`: Display spatial scales in units of AU or parsec.

`/log`: Display the image using logarithmic spacing of the brightness levels. This allows you to gain far greater depth in the image.

`/contour`: Overplot contours over the image

`nlevels`: Number of levels for the contours

`/noimage`: If set, only plot axes

`position`: An array of 4 numbers specifying position of plot on the canvas. Like position keyword in typical IDL plotting routines.

`maxlog`: Set the maximum number of factors of 10 the log brightness color coding will span

`saturate`: Allows you to enhance the contrast of very weak emission regions by saturating bright regions

`/jpg`: Write the image to a JPEG file

`lgrange`: A two-valued array specifying the range in brightness (in 10-log) that the image will show

`filenr`: (For case of /jpg): if set to e.g. 6 it will write the JPEG image to file `image_6.jpg`.

`ilam`: If the image is a multi-color image, ilam specfies which of the images you wish to plot

`coltune`: If set to 1, then rescale the brightness of all channels the same value, to get the best color depth. If set to a 3-element array, you can directly specify the weight of each color. In this way you can really fine-tune the colors.

`zoom`: If set to a 4-valued array, it makes `plotimage` put the proper x- and y- axis scaling for the particular zoom-in. Normally the center of the image is taken to be $(0, 0)$, but with this zoom keyword you can set exactly what the x- and y-axes should display. Warning: it overrides the pixel size specifications in the image.

14.1.3 Subroutine makeimage

The subroutine `makeimage` allows the user to interact with RADMC-3D for making images in an easy way, although a direct calling of the `radmc3d` command with the corresponding keywords is almost just as easy. So one can also consider this as an example routine for calling RADMC-3D for making images. From within IDL you can call makeimage as follows:

```
makeimage,incl=34.,phi=40.,lambda=11.3
```

Here are the keywords:

`incl`: The inclination of the object on the sky of the observer. `incl`=0 means a view from the northpole downward, `incl`=180 means a view from the southpole upward and `incl`=90 means an edge-on view.

`phi`: The rotation of the object along its z-axis. A positive `phi` means that the object rotates counter-clockwise, i.e. that the observer rotates clockwise around the object.

`npix`: Number of pixels (assumed to be the same for x and y)

`sizecm / sizeau / sizepc`: The size of the image in units of centimeter / AU / parsec. The size means the full width and full height of the square image.

`posang`: The rotation of the image in the image plane, i.e. the position angle of the image on the sky. Default is 0.

`nofluxcons`: If set to 1 (`/nofluxcons`) we use the fast ray-tracing method, while if not set (default) we use the accurate method with sub-pixeling for flux conservation (see Section 9.6).

`pointcm / pointau / pointpc`: A three-valued array giving the 3-D coordinates of the point toward which we aim our camera. Default is (0.,0.,0.). Units are cm / AU / parsec.

`ifreq`: If specifying `ifreq` (putting it to an integer value of 1 or higher) then the wavelength at which the image is going to be taken is taken from the global frequency array from the `wavelength_micron.inp` file. The integer `ifreq` is then the index of the wavelength you want to use. Note that this integer starts with 1 (fortran convention).

`lambda`: If `lambda` is specified, this will be the wavelength at which the image is to be taken. This wavelength does not have to be part of the global frequency array. It can be any value, even a value in between wavelength grid points of the dust opacity files or so. In that case, a linear interpolation of these opacities will then allow RADMC-3D to nevertheless make the image. So any positive value of `lambda` is allowed. NOTE: You cannot specify both `lambda` and `ifreq` simultaneously.

`nostar`: If set, then the star(s) in the model are not included in the images.

`zoomau / zoompc`: Specify the precise window on the image plane which you like to zoom in to. Note that $(0.,0.)$ is the location in the image plane that points to the pointing location specified by `pointcm / pointau / pointpc`. NOTE: You cannot specify both `sizecm / sizeau / sizepc` and `zoomau / zoompc` simultaneously.

`plottau`: If set to 1, the images will not show the emission at that wavelength but instead the total integrated optical depth of the ray at that wavelength. This is only useful for debugging purposes.

`iounit`: For child mode (See chapter 12).

NOTE: If you want to make multiple images of the same object, then it may be much too slow if each time a new image is to be taken, the RADMC-3D code must be restarted and the entire model must be re-read into RADMC-3D. You can use RADMC-3D in "child mode" to have it start up just once (and reading all input data just once) and keeping alive until explicitly told to end. By communicating with it via a pipe you can then quickly get your images one-by-one while having the slow I/O only once. You do this by starting RADMC-3D in the way described in Chapter 12, and then calling `makeimage` with keyword `iounit` equal to the unit of the pipe. See chapter 12 for an example.

14.1.4 Subroutine read_data()

It is always useful to analyze the dust temperature structure that RADMC-3D produces. The function `read_data()` (or `readdata()` in abbreviated form) is meant to do that. In IDL, the way it works is:

```
.r readradmc
a=read_data(/dtemp)
```

The `/dtemp` stands for "read the dust temperature". You can also read the dust density (which is an input, not an output of RADMC-3D):

```
a=read_data(/ddens)
```

You can find what is contained in the structure `a` by

```
help,a,/struct
```

The structure `a` contains a sub-structure `grid`:

```
help,a.grid,/struct
```

which contains all the information of the grid. We come back to that later.

If you just read the dust temperature (with `/dtemp`) then you can see that the structure contains a large array called `a.temp`. Its dimensionality depends on which kind of grid you are using:

- *Regular grid:*
  The array `a.temp` (or any other data array) will have dimensions `a.temp(nx,ny,nz,nspec)`, where `nx,ny,nz` are the number of cells in x, y and z direction and `nspec` is the number of dust species. If `nspec=1`, then the array automatically becomes `a.temp(nx,ny,nz)` (this is IDL convention).
- *Oct-tree AMR grid:*
  The array `a.temp` (or any other data array) will have dimensions `a.temp(ncells,nspec)`, where `ncells` is the number of real cells (excluding the branches that are divided into subcells: only the leafs count) in the AMR oct-tree. How the (complex!) oct-tree is structured is specified in `a.grid`. The `nspec` is again the number of dust species.
- *Layer-style AMR grid:*
  The array `a.temp` (or any other data array) will have dimensions `a.temp(nxmax,nymax,nzmax,nspec,nlayers+1)`, where `nxmax,nymax,nzmax` are the maximum of number of cells in x, y and z direction of all the layers (including the base grid). The `nspec` is again the number of dust species. The `nlayers` is the number of refinement layers (patches), where 0 means that there is only the base grid, 1 means there is one single patch of refinement, etc. The last index of the `a.temp` array goes from 0 to `nlayers`. Here 0 means the base grid, 1 the first layer of refinement, etc. Note that if you just read the dust temperature in this way, the regions in the parent layers (including the base grid) that are replaced by a refined layer will have the value 0. This makes a "hole" in the dust temperature distribution. If you want IDL to fill these holes with the rebinned values of the refined layers, then you can call `read_data` with the `/fill` keyword, i.e. `read_data(/dtemp,/fill)`.

Now coming back to the `a.grid` sub-structure. This contains all the information about the grid. Again the content of this structure depends on which gridding you use:

- *Regular grid:*
  The structure contains `x,y,z` which are 1-dimensional arrays with the cell-centered x, y and z coordinates (for spherical coordinates they are r, θ and ϕ, but then the grid also contains, for your convenience, the entries `r`, `theta` and `phi`). It also contains `xi,yi,zi` which are again 1-D arrays with the x, y and z coordinates of the cell walls (for spherical coordinates `ri`, `thetai` and `phii`). `nx, ny, nz` are the number of cells in x, y and z direction.

- *Oct-tree AMR grid:*
  The `x,y,z` and `xi,yi,zi` (and in addition to that in spherical coordinates `r`, `theta`, `phi`, `ri`, `thetai` and `phii`) still have the same meaning as in the regular grid case. In this case, however, this "regular grid" is just the "base grid" of the oct-tree refinement. The oct-tree structure is now specified by the 1-D byte-array `octtree`, which contains only the values 0 and 1. See Section A.2 for details about their meaning. The order of the values of the data are the same as the order of the cells given by `octtree`.

- *Layer-style AMR grid:*
  Again, as in the oct-tree case, the `x,y,z` and `xi,yi,zi` and their spherical-coordinates counterparts now have their meaning for the base grid. The `nlayers` tells how many layers (in addition to the base grid) there are. The `iparent(nlayers+1)` give the parent layer for each layer, where 0 means base grid. The `ixyz(3,nlayers+1)` give the starting point of the layer in the parent grid, `nxyz(3,nlayers+1)` the size of the layer in the parent grid and `nnxyz(3,nlayers+1)` the size of the layer in its own grid. The `layer_x(nxmax,nlayers+1)`, `layer_y(nxmax,nlayers+1)`, `layer_z(nxmax,nlayers+1)`, `layer_xi(nxmax,nlayers+` `layer_yi(nxmax,nlayers+1)` and `layer_zi(nxmax,nlayers+1)` are like the `x,y,z` and `xi,yi,zi`, but now for each layer separately. Note that e.g. `layer_x[*,0] = x[*]`, etc, because layer 0 is identical to the base grid. Note also that `ixyz[*,0]` and `nxyz[*,0]` have no meaning, but `nnxyz[*,0]` are the same as `nx, ny, nz`.

For all the above griddings, the following additional elements are present in `a.grid`. For instance, `gridstyle (=0,1,10)` specifies which of the above griddings is used, `coordsys` is the coordinate system, `mirror (=0,1)` a flag whether equatorial mirror symmetry is present (only for spherical coordinates), `ncell` gives you the total number of actual cells, `ncellinp` gives you the total number of read-in cell values (in case of layer-type refinement this is generally larger than `ncell`, `incx, incy, incz (=0,1)` are flags whether the x, y or z dimensions are active or not.

Note that even if you have e.g. a regular grid, the `ixyz` etc elements are still in the structure, but they are simply 0.

14.2 Support for FITS

Many people in astronomy use the FITS format (Flexible Image Transport System) for analyzing images or other observational data. Many software packages are geared toward reading and processing FITS data. For instance, the `ds9` image viewer[1] is very powerful, but requires its images in FITS format.

We provide a conversion routine from the standard `image.out` image format produced by RADMC-3D to FITS format. The routine is called `radmcimage_to_fits` and is located in the file `idl/radmc3dfits.pro`. To use this, you must have the ASTROLIB [2] library of IDL installed.

To convert to FITS format, you must specify the distance at which the observer stands from the object. The reason is that the `image.out` file produced by RADMC-3d is *distance independent!* The pixel size in `image.out` is specified in centimeters, not as an angular size. The `radmcimage_to_fits` routine automatically converts this to pixel scale in degrees, but it must know the distance.

Here is how you convert `image.out` into `image.fits`. First go into IDL and then:

```
.r radmc3dfits
radmcimage_to_fits,'image.out','image.fits',140.
```

where the last number (140.) is the distance to the object in units of parsec. In the FITS file the unit of the intensity is Jansky/Pixel. The pixel sizes are specified in degrees. Once you have made this conversion, you can, for instance, use `ds9` (if it is installed on your system!) to view your image. From the unix shell you type:

```
ds9 image.fits
```

Just for your information, in case you want to know more about the FITS conversion: The FITS header looks for example like this:

```
SIMPLE  =                    T /image conforms to FITS standard
```

[1] `http://hea-www.harvard.edu/RD/ds9/`
[2] `http://idlastro.gsfc.nasa.gov/`

```
BITPIX  =                  -64 /bits per data value
NAXIS   =                    2 /number of axes
NAXIS1  =                  100 /
NAXIS2  =                  100 /
EXTEND  =                    T /file may contain extensions
BTYPE   = 'Intensity'
BUNIT   = 'JY/PIXEL'
CDELT1  =  6.9418601352850E-07
CUNIT1  = 'deg     '
CRPIX1  =  5.0000000000000E+01
CDELT2  =  6.9418601352850E-07
CUNIT2  = 'deg     '
CRPIX2  =  5.0000000000000E+01
RESTFREQ=   2.997924580000E+13
END
```

which you can see if you type `less image.fits` in the Unix shell and rescale the width of your shell window to 80 characters. If you want to know more about the details of the FITS format, please consult the various papers by E.W. Greisen, for instance Greisen & Calabretta (2002) A&A 395, 1061-1075.

14.3 The image viewing GUI: viewimage.pro

Making images of a model can be done "by hand" using the tools in the `readradmc.pro` file (see Section 14.1). But it is much more convenient to use the fully widget-based graphical user interface `viewimage.pro` (see Fig. 14.1)[3]. This interface can be used once the dust temperatures (in case of dust continuum radiative transfer) have been computed using e.g. the thermal monte carlo method (i.e. after having called `radmc3d mctherm`). Or more precisely: the file `dust_temperature.dat` should be present and consistent with the other files. If this is satisfied, then one can go into IDL (does not work on the IDL-clone "GDL") and type:

```
.r viewimage
viewimage
```

and one should get the GUI shown in Fig. (14.1). *NOTE: It may take a while for RADMC-3D to load all the data into memory the first time, so before the first image appears on the screen it may take some time. From that point on, further ray-trace actions should go much quicker.* Here is a list of controls and their functions:

- `''Quit Viewer'' button:` Ends this viewer and quits RADMC-3D.
- `''Write Image'' button:` Writes a idl.ps postscript version of the plot on the screen.
- `''mouse rotate'' switch:` If unset (default), the mouse clicks on the plotting pane act to select a zoom-in box. If set, the mouse clicks on the plotting pane act to rotate the object. Note that the rotation can also be done by hand by setting the values of `''Inclination''` and `''Phi''` and redoing an image rendering with the `''Render Image''` button.
- `''lin'' switch:` Switch between linear color table of intensity and logarithmic color table of intensity.
- `''preview'' switch:` If set (default) then the ray-traced image is done without sub-pixeling in regions where the model has higher spatial resolution than the image resolution can resolve. This is a fast mode (i.e. hence the name "preview"). If unset, then the ray-tracer always ensures that if a pixel of the image does not resolve details of the model, it will internally refine the pixel in 2x2 (and recursively repeat this until the resolution matches that of the model), and finally integrate the flux of all the sub-pixels to find the flux of the parent pixel. The intensity it then puts into this pixel is then the true average intensity over all the pixel. The sub-pixels will never be seen by the user. They are only made internally in RADMC-3D to ensure the correct flux in the pixel, and then dropped again. For science-quality images the `''preview''` button should be unset. It may take longer, however, to render. Please read Section 9.6 for details about this procedure.

[3] *NOTE: Currently there appears to be a problem that viewimage will freeze, which happens on some machines and not on other machines (in fact it happened on one Macbook but not on another, while the systems were seemingly identical). This is somehow related to the way radmc3d and IDL communicate with each other in child mode (see chapter 12): it must be a buffering problem. So far I could not figure out what is going wrong, but I will continue to work on this. If you experience this problem, try calling viewimage,/nochild. That is slower, but should work.*

- ''contour'' switch: If set: overplot contours.
- ''star'' switch: If set (default): Include the flux of all the point-source stars in the image. Not setting it has the potential advantage that you can concentrate on the circumstellar material and not be 'blinded' by the strong starlight.
- slider in this box: The slider in the same box as the above switches selects the IDL color table for monochromatic images.
- ''MaxLog'' textbox (editable): The maxlog is the maximum number of factors of 10 that we will include in the color table. A high number gives more extreme "depth" to the image, but may also wash out details.
- ''Saturate'' textbox (editable): Saturate the image with this factor. Default is 1.0, i.e. no saturation.
- ''Nr Cont'' textbox (editable): Set how many contours you want if the contour switch is 'on'.
- ''Render Spectrum'' button: Render a complete SED. This may take a long time!
- ''Render Image'' button: Render a single image. See entries below for the settings.
- ''Unzoom'' button: If you are zoomed in, and you want to zoom out again, push this button. Note: Zoomin in is done by selecting a region with the moise in the image pane (make sure the "mouse rotate" switch is off) and pressing ''Render Image''.
- ''Npix'' textbox (editable): The number of x and y pixels of the image.
- ''Size'' textbox (editable): The size of the image. **[BUG HERE: This does not seem to work.]**
- ''Inclination'' textbox (editable): The inclination where the observer is placed (at large distance).
- ''Phi'' textbox (editable): The azimuthal angle where the observer is placed (at large distance).
- slider below: The wavelength slider. These are the wavelengths from the wavelength_micron.inp file and using the slider you can select one of these values. BUT: you can also select the wavelength by directly editing the textbox next to it, see below.
- textbox next to slider: The current wavelength in micron. This is automatically set when the slider is moved. BUT you can also put in any value of the wavelength you want and type return to get the image at the precise wavelength of interest. That may be a wavelength that is not one of the wavelength_micron.inp values, but somewhere in between or even outside that grid. You can try any value.

The image is of course independent of observer distance (except for the local observer mode). The total flux, however, is a distance-dependent quantity. Written in the image is the total flux, normalized to a distance of 1 parsec. Clearly, if the model is far bigger than 1 parsec, then this number has no physical meaning. But by scaling the image flux to a reasonale distance you will get reasonable answers.

The viewimage routines will print to the command line the radmc3d command sequence used to make the image you see now on your screen. This is just for the user's convenience, that it is clearly seen which commands radmc3d receives. You can simply copy-paste such a line to the shell command line and you will see that RADMC-3D will do precisely that command (note that if already another RADMC-3D is running a huge model, you may get memory problems when doing this in parallel to that model).

Note also that each time a new image is made and is shown in the viewer, the same image is also stored in the file image.out in the current directory. This means that you can read the latest image using the readimage() routine in the readradmc.pro file. As an example of such a complete sequence:

```
.r viewimage
viewimage
<<< NOW MAKE WITH THIS WIDGET SOME NICE IMAGE YOU WANT TO STUDY MORE >>>
.r readradmc
a=readimage()
window,0
surface,a.image
```

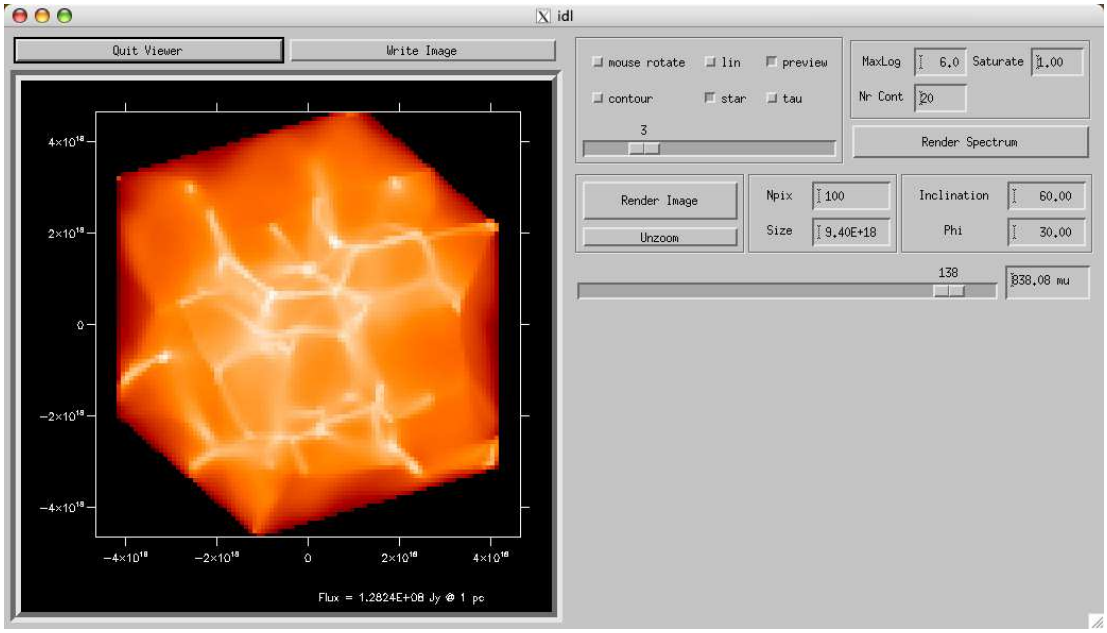


Figure 14.1. The graphical user interface (GUI) to the ray-trace-based imager module of RADMC-3D.

In this example you make a surface plot in window 0 of the image you see in the viewer.

There are further capabilities of `viewimage`, which can be switched on with keyword options to the `viewimage` subroutine. Here is a list of such keyword options (i.e. type e.g. `viewimage,/color` to enable the first option):

> `/color`: When `/color` is set then you will find three wavelength sliders which can be independently shifted. These three sliders represent the red, green and blue channels of the image. This way you can make false color images.
>
> `/au / /pc`: When setting either `/au` or `/pc` the axis will not be drawn with centimeter units, but instead with AU or parsec units.
>
> `/small`: If set, the widget will be smaller, so that it fits on low-resolution screens.
>
> `/verti`: If set, the widget will put the controls below the image instead of next. Can be useful on high-resolution screens to save screen real estate.
>
> `/nochild`: When setting `/nochild`, the RADMC-3D code will be called separately for each image rendering. This can be very slow, but it has the advantage that in case of problems the debugging might be easier, because all I/O of the `radmc3d` executable will then go to screen.
>
> `/lines`: This includes entry fields such as `imol`, `iline` and `vkms` to make it easier to specify the precise wavelength of the image in case of lines.

Tip: If viewimage unexpectedly quits or freezes, please have a look at the file `radmc3d.out` *which contains the messages that RADMC-3D outputs. This may give hints what went wrong. If you have called viewimage with the option /nochild, then the output will have been written to screen, not to* `radmc3d.out`.

Another thing to keep in mind is that when RADMC-3D makes images, it will run a small Monte Carlo simulation beforehand to compute the scattering source function (see).

14.4 Making and reading spectra with IDL

[THIS STILL HAS TO BE WRITTEN]

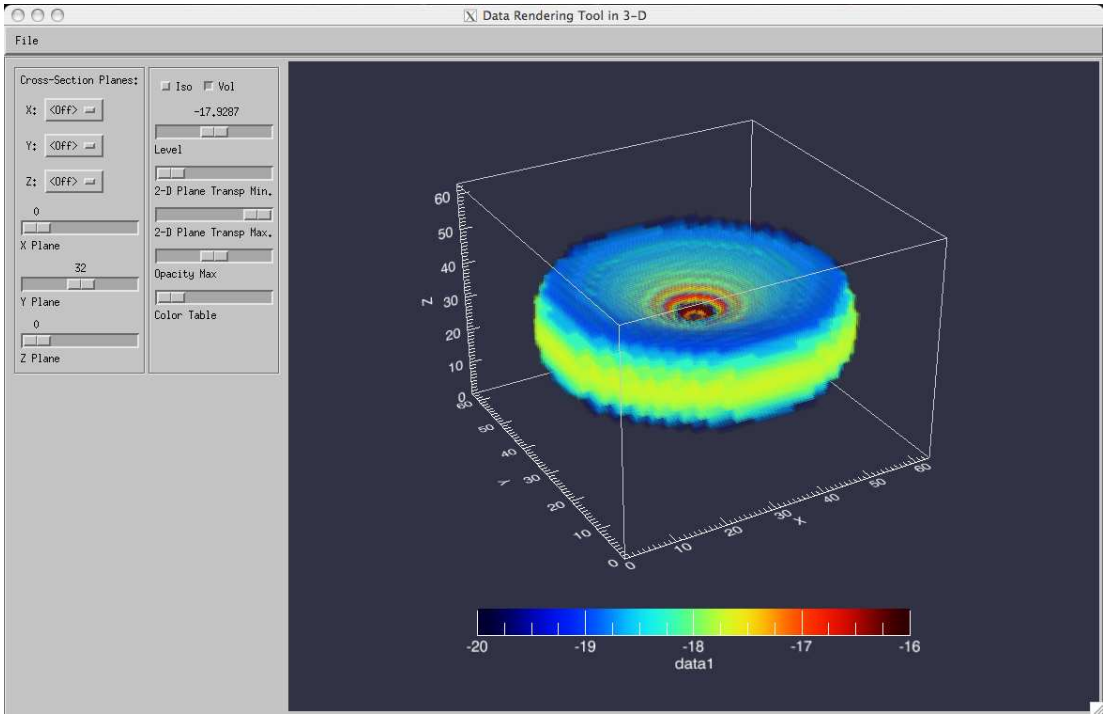


Figure 14.2. The graphical user interface (GUI) `render3d.pro` that allows you to analyze 3-D regularly spaced datacubes. In Chapter 15 it is described how to create regularly spaced datacubes of any variable in the model from a complex AMR-based model. Shown in this figure is in fact the `run_disk_1/` model, which is an AMR-based model, in which the trick of Chapter 15 is applied to create a regularly spaced box.

14.5 A general-purpose 3-D datacube analysis GUI

Although this is not specific for RADMC-3D, we thought it would anyway be useful to provide this: a super-fast interactive IDL GUI for analyzing 3-D data cubes called `render3d.pro`. This code is a general-purpose adaption of one of IDL's example codes, so this is not in any way copyrighted by the RADMC-3D authors, even though we made extensive changes to it for the better. The tool can be used also for any other 3-D datacube data: it is not at all limited to RADMC-3D.

The `render3d.pro` tool can only handle 3-D regularly spaced data. No AMR-based models can be *directly* analyzed in this way. But in Chapter 15 we describe a feature of the `radmc3d` code that allows you to easily create 3-D regularly spaced boxes from anywhere in your model at any spatial resolution you like. You can then subsequently feed that datablock into `render3d.pro` for 3-D viewing.

Using `render3d.pro` is simple:

```
q = <some 3-D array of floats or double precisions>
.r render3d
render3d,q
```

Have fun! For all the functionality, just try things out. It should be reasonably self-explanatory. See Fig. 14.2 for a screen-shot. Note: Apart from all the sliders and buttons, try also out how you can interactively rotate the 3-D datacube with the mouse: just click on the plotting window and drag while keeping clicked.

Chapter 15

Model analysis (II): Tools inside of radmc3d, steered by IDL

There are also some special purpose features in the Fortran-90 `radmc3d` code that can be useful for analyzing complex AMR-gridded models. You do not have to directly interact with `radmc3d` to use these features, as there are IDL routines that do this for you. If you do not have IDL, you will have to directly interact with `radmc3d` to use these features.

15.1 Making a regularly-spaced datacube ('subbox') of AMR-based models

Because handling AMR-based models in IDL or other data analysis packages can be rather cumbersome, we decided that it would be useful to create the possibility in `radmc3d` to generate 1-D, 2-D or 3-D regularly spaced 'cut-outs' or 'sub-boxes' (whatever you want to call them) of any variable of the model. An example of how this all works, and how these 1-D, 2-D or 3-D sub-boxes can be used with the `render3d.pro` tool set described in Section 14.5, is given in the model `examples/run_disk_1/`.

15.1.1 Creating and reading a subbox from within IDL

If all the IDL tools are set up properly, you can make use of this datacube creation feature of `radmc3d` entirely through IDL. An example, type in IDL:

```
.r readradmc.pro
q=subbox('dust_density')
```

This creates a box with the size of the original model box, but this time regularly spaced. The data is in `q.data`. You can see this by typing:

```
help,q,/str
```

You will see that `q.data` is a 3-D array of 64x64x64 (default size).

The box contains the dust density. You can specify the size and the number of grid points of the regularly-spaced box:

```
.r readradmc.pro
@natconst.pro
q=subbox('dust_density',nxyz=64,size=2*100*AU)
.r render3d.pro
render3d,alog10(q.data>1d-20)
```

NOTE: The `size` keyword is the *full-width* size of the box! So if you want to capture a disk with radius 100 AU, then you must have a box size of 200 AU.

You can also make and read in separate steps:

```
.r readradmc.pro
@natconst.pro
makesubbox,'dust_density',nxyz=64,size=200*AU
q=readsubbox('dust_density_subbox.out')
.r render3d.pro
render3d,alog10(q.data>1d-20)
```

You can specify the box location with the keyword `pos`:

```
q=subbox('dust_density',nxyz=64,size=200*AU,pos=[30,30,30]*AU)
```

or by specifying the corners of the box directly:

```
q=subbox('dust_density',nxyz=64,box=[-1,1,-1,1,-1,1]*80*AU)
```

Note that if you have `radmc3d` already running in the background using the `child` mode (see Chapter 12) then you can do the above commands more quickly by directly communicating through the pipe by passing the keyword `iounit=<myiounitnumber>` (where the thing in between $<$ and $>$ should be the unit number of the biway pipe to `radmc3d`) to the above routines `makesubbox` or `subbox`.

15.1.2 Creating and reading a subbox by directly communicating with radmc3d

You can call `radmc3d` directly from the shell asking it to make the subbox. Here is an example:

```
./radmc3d subbox_dust_density subbox_nxyz 64 64 64 \
subbox_xyz01 -2.d15 2.d15 -2.d15 2.d15 -2.d15 2.d15
```

Chapter 16

Creating Protoplanetary Disk Models using a GUI

Author: A. Juhasz

Attila Juhasz has created an IDL-driven graphical user interface (GUI) for modeling protoplanetary disks. The GUI for RADMC-3D was developed for YSO disks/envelopes but can also be used (to some extent) as a general purpose user interface. Or it may serve as a template for other users to create GUIs for their own models. The GUI is written in IDL and consits of five IDL routines; radmc3d_gui.pro, read_params_radmc3d.pro, problem_params.pro, problem_setup_yso.pro, my_gap_function.pro.

- **radmc3d_gui.pro** This file contains all the widget definitions and basically everything related to the visualization.
- **read_params_radmc3d.pro** This file contains two functions to read and write the parameters into a file.
- **problem_params.pro** Those, who have actually worked with the 2D version of RADMC, this file should look familiar. This file contains all parameters required to set up the problem and create the input files for RADMC 3D.
- **problem_setup.pro** This file has also the same function as in the 2D version of RADMC. On the basis of the parameters stored in problem_params.pro, it creates the input files to RADMC 3D.
- **my_gap_function.pro** This is a small routine to cut a gap in the disk to demonstrate, how one can use this GUI with one's own dust density setup.
- **my_gap2_function.pro** This is a small routine to cut a second gap in the disk (in the exact same way as my_gap_function.pro) to demonstrate, how one can use this GUI with one's own dust density setup.

16.1 How to run the GUI

The GUI can be run in a very simple way. First an IDL should be started and the radmc3d_gui.pro routine should be compiled.

```
IDL> .r radmc3d_gui
```

With this step not only radmc3d_gui.pro, but also all other necessary files/routines will be compiled (Note: it assumes that you are in the examples/run_whatever directory and the readradmc.pro file is located in the ../../idl/ directory). After the compilation the GUI is ready to be run with the command radmc3d_gui:

```
IDL> radmc3d_gui
```

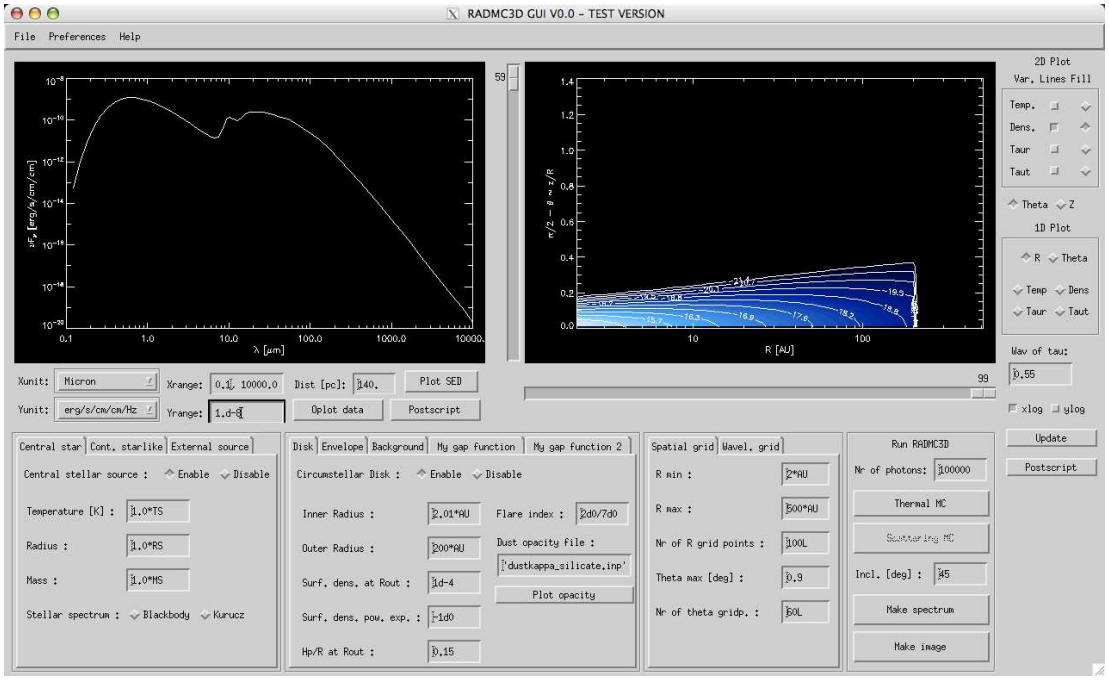


Figure 16.1. The graphical user interface for a protoplanetary disk model. Author: A. Juhasz.

After this command has been executed the user should see a window similar to that in Fig 16.1.

To make things easy (especially for those ones who use a radiative transfer code for the first time) the GUI comes with a simple 'default' setup for the dust density distribution. This setup is based on a common parametrization of a protoplanetary disk and a surrounding envelope. However, the user may want to change the whole density distribution or may want to add extension(s) to this setup (e.g. a gap in the disk). One can add his/her own extension(s) to the model setup so, that the GUI will recognize them and new tabs will automatically open to access the parameters of these extensions.

16.2 Create your own setup and/or open your own tab

Adding new extension to the code which appears as a new tab in the dust density setup frame is relatively easy, one only needs to keep a few rules. To make it clear how this can be done it is useful to understand what the GUI does behind the buttons and windows. In a few steps the GUI does the following;

1. Write the parameter values to problem_params.pro
2. Run problem_setup_yso.pro to create all necessary input files for RADMC3D
3. Run RADMC3D mctherm to calculate the temperature structure
4. Run RADMC3D to calculate images/spectra

After Step 2. any arbitrary IDL procedure can be run to modify (or even completely re-create) the dust density distribution. Such a procedure can be integrated to the GUI in the following way;

- The procedure should not accept any keywords or input variable, instead, the user should add the @problem_params.pro line after the procedure name, allowing the procedure to access the variables in problem_params.pro;

```
pro my_arbitrary_procedure
@problem_params.pro
```

- Create a new section of parameters in problem_params.pro which will be used in the user's procedure. For examle;

```
;
; My extra function - example
;   Cut a gap in the disk reducing the density by an arbitrary factor
;
extra_gap_tab_name = 'My gap function' ; Name of the tab in the GUI
extra_gap_enable = 1                   ; 0-Enable, 1-Disable this function
extra_gap_func = 'my_gap_function'     ; Name of the user defined function
extra_gap_rin = 5.0*AU                 ; Inner radius of the gap
extra_gap_rout = 10.*AU                ; Outer radius of the gap
extra_gap_drfact = 1d-4                ; Density reduction factor in the gap
```

- The parameter names should look like; extra_secname_pname. The 'extra' means that this is an additional parameter for which a new tab should be opened. The 'secname' will be the identifier of this section of parameters within the widgets and should be the same for all parameters within this section. The 'pname' is the actual parameter name, that can be any arbitrary name.
- Each new section should contain a parameter called 'extra_secname_tab_name', which will be the title of the new tab which will open in the dust density setup frame.
- Each new section should contain a parameter called 'extra_secname_func', which should be the name of the user's procedure.
- Each new section should contain a parameter called 'extra_secname_enable', which should have a value of 0 or 1 and disables or enables the execution of the user's procedure after problem_setup_yso.pro.

If the user followed the instructions above and start the GUI a new tab in the density setup frame should open for the user's procedure. Input fields for the parameters will have the parameter names as titles. If one made comments in problem_params.pro at the end of a line to explain the meaning of the parameter at the beginning of the line, this explanation/comment will be available in the Help menu under "Help on parameters". Two examples were given in the current problem_params.pro to open gaps in the dust density distribution.

Chapter 17

How to convert old-style RADMC models to RADMC-3D

Some users of RADMC-3D may have used the predecessor program RADMC before. The input files of RADMC are different from those used by RADMC-3D. You may have model setups created for RADMC which you would like to feed into RADMC-3D. The best way would be of course to edit your `problem_***.pro` files and make sure that they create the RADMC-3D input files instead of the RADMC input files. But that costs quite some work, and you may make errors.

We offer the IDL routine `radmc2radmc3d.pro` in the `idl/` directory that automatically converts all the RADMC input files into RADMC-3D input files. In particular, it will do the following conversion:

RADMC	RADMC-3D
`radius.inp` & `theta.inp`	`amr_grid.inp`
`starinfo.inp` & `starspectrum.inp`	`stars.inp`
`dustdens.inp`	`dust_density.inp`
-	`radmc3d.inp`

The file `radmc3d.inp` is ignored for now, and instead a fresh `radmc3d.inp` is created. While RADMC-3D would read `radmc3d.inp` if no `radmc3d.inp` is present, the presence of `radmc3d.inp` means that RADMC-3D will read `radmc3d.inp` (and *not* read `radmc3d.inp`).

The opacity files are kept as they are. While the recommended style for the opacities is the `dustkappa_*.inp` file, RADMC-3D can also read the old style `dustopac_*.inp` files. Also the `frequency.inp` file is kept as it is.

NOTE: For now the quantum-heated grains are not ported to RADMC-3D. Also some specialties of the RADMC code may not yet work in RADMC-3D – just keep this in mind.

So if you have a working model for RADMC, and you want to run it with RADMC-3D, then just go into the model directory, go into IDL and type:

```
.r radmc2radmc3d.pro
```

and this should convert the files from RADMC-style to RADMC-3D-style, as described above. Note that you must now redo the thermal Monte Carlo, because RADMC-3D reads its own style of dust temperature file. So go out of IDL and type (on the shell):

```
radmc3d mctherm
```

wait until it is finished, and now you can make your spectra and images.

Chapter 18

Tips, tricks and problem hunting

18.1 Tips and tricks

RADMC-3D is a large software package, and the user will in all likelihood not understand all its internal workings. In this section we will discuss some issues that might be useful to know when you do modeling.

- *Things that can drastically slow down ray-tracing:*
  When you create images or spectra, `radmc3d` will perform a ray-tracing calculation. You may notice that sometimes this can be very fast, but for other problems it can be very slow. This is because, depending on which physics is switched on, different ray-tracing strategies must be followed. For instance, if you use a dust opacity without scattering opacity (or if you switch dust scattering off by setting `scattering_mode_max` to 0 in the `radmc3d.inp` file), and you make dust continuum images, or make SEDs, this may go very rapidly: less than a minute on a modern computer for grids of 256x256x256. However, when you include scattering, it may go slower. Why is that? That is because at each wavelength `radmc3d` will now have to make a quick Monte Carlo scattering model to compute the dust scattering source function. This costs time. And it will cost more time if you have `nphot_scat` set to a high value in the `radmc3d.inp` file, although it will create better images. Furthermore, if you *also* include gas lines using the simple LTE or simple LVG methods, then things become even slower, because each wavelength channel image is done after each other, and each time all the populations of the molecular levels have to be re-computed. If dust scattering would be switched off (which is for some wavelength domains presumably not a bad approximation; in particular for the millimeter domain), then no scattering Monte Carlo runs have to be done for each wavelength. Then the code can ray-trace all wavelength simultaneously: each ray is traced only once, for all wavelength simultaneously. Then the LTE/LVG level populations have to be computed only once at each location along the ray. So if you use dust and lines simultaneously, it can be advantageous for speed if you can afford to switch off the dust scattering, for instance, if you model sub-millimeter lines in regions with dust grains smaller than 10 micron or so. If you must include scattering, but your model is not so big that you may get memory limitation problems, then you may also try the fast LTE or fast LVG modes: in those modes the level populations are pre-computed before the ray-tracing starts, which saves time. But that may require much memory.

18.2 Bug hunting

Although we of course hope that `radmc3d` will not run into troubles or crash, it is nevertheless possible that it will. There are several ways by which one can hunt for bugs, and we list here a few obvious ones:

- In principle the `Makefile` should make sure that all dependencies of all modules are correct, so that the most dependent modules are compiled last. But during the further development of the code perhaps this may be not 100% guaranteed. So try do `make clean` followed by `make` (or `make install`) to assure a clean make.
- In the `Makefile` you can add (or uncomment) the line
  `BCHECK = -fbounds-check`, if you use `gfortran`. Find the array boundary check switch on your own

compiler if it is not `gfortran`.

- Make sure that in the `main.f90` code the variable `debug_check_all` is set to 1. This will do some on-the-fly checks in the code.

18.3 Some tips for avoiding troubles and for making good models

Here is a set of tips that we recommend you to follow, in order to avoid troubles with the code and to make sure that the models you make are OK. This list is far from complete! It will be updated as we continue to develop the code.

1. Make a separate directory for each model. This avoids confusion with the many input and output files from the models.
2. When experimenting: regularly keep models that work, and continue experimenting with a fresh model directory. If things go wrong later, you can always fall back on an older model that *did* work well.
3. Keep model directories within a parent directory of the code, just like it is currently organized. This ensures that each model is always associated to the version of the code for which it was developed. If you update to a new version of the code, it is recommended to simply copy the models you want to continue with to the new code directory (and edit the `SRC` variable in the `Makefile` if you use the techniques described in Section 4.5 and Chapter 13).
4. If you make a new model, try to start with as clean a directory as possible. This avoids that you accidently have a old files hanging around, their presence of which may cause troubles in your new model. So if you make a model update, make a new directory and then copy only the files that are necesary (for instance, `problem_setup.pro`, `dustkappa_silicate.inp`, `Makefile` and other necessary files). One way of doing this easily is to write a little perl script or csh script that does this for you.
5. In the example model directories there is always a `Makefile` present, even if no local `*.f90` files are present. The idea is that by typing make cleanall you can safely clean up the model directory and restore it to pre-model status. This can be useful for safely cleaning model directories so that only the model setup files remain there. It may save enormous amounts of disk space. But of course, it means that if you revisit the model later, you would need to redo the Monte Carlo simulations again, for instance. It is a matter of choice between speed of access to results on the one hand and disk space on the other hand.
6. If you use LVG or escape probability to compute the level populations of molecules, please be aware that you must include all levels that could be populated, not only the levels belonging to the line you are interested in.

18.4 For the careful modeler: things you may want to check

In principle RADMC-3D should be fine-tuned such that it produced reliable results in most circumstances. But radiative transfer modeling, like all kinds of modeling, is not an entirely trivial issue. Extreme circumstances can lead to wrong results, if the user is not careful in doing various sanity checks. This section gives some tips that you, the user, may wish to do to check that the results are ok. This is not an exhaustive list! So please remain creative yourself in coming up with good tests and checks.

1. *When making images or spectra*, one important issue is always the proper choice of resolution of the pixels. This is not only for the pixels you see in your image, but also for the *recursive sub-pixeling* (see Section 9.6) which ensures proper flux conservation. In principle the recursive sub-pixeling is pre-tuned by us (the programmers) in a sensible way. But we cannot guarantee that it works always well under all conditions! So if you want to be absolutely sure that the image flux is properly accounted for, please read Section 9.6) carefully, and play a bit with the various tuning parameters.
2. *When making images or spectra* in which dust scattering is important, the scattered light emissivity is computed by a quick Monte Carlo simulation before the ray-tracing (see Section 6.5.4). This requires the setting of the number of photon packages used for this (the variable `nphot_scat` for images and equivalently

nphot_spec for spectra, both can be set in the radmc3d.inp file). If you see too much "noise" in your scattering image, you can improve this by setting nphot_scat to a larger value (default = 100000). If your spectrum contains too much noise, try setting nphot_spec to a larger value (default = 10000).

3. *For rather optically thick models* you may want to experiment with grid resolution and refinement. Strictly speaking the transition from optically thin to optically thick, as seen both by the radiation entering the object and by the observer, has to occur over more than one cell. That is for very optically thick models, one may need to introduce grid refinement in various regions. As an example: an optically thick protoplanetary disk may have an extremely sharp thin-thick transition near the inner edge. To get the spectra and images right, it is important that these regions are resolved by the grid (note: once well inside the optically thick interior, it is no longer necessary to resolve individual optical mean free paths, thankfully). It should be said that in practice it is often impossible to do this in full strictness. But you may want to at least experiment a bit with refining the grid (using either "separable refinement", see Section 10.2, or AMR refinement, see Section A.2.2).

18.5 Common problems and how to fix them

When using a complex code such as RADMC-3D there are many things that can go wrong. Here is a list of common issues and tips how to fix them.

1. **After updating RADMC-3D to a new version, some setups don't work anymore.**
   This problem can be due to several things:
   (a) When your model makes a local radmc3d executable (see Section 4.5), for instance when you use the userdef_module.f90 to set up the model, then you may need to edit the SRC variable in the Makefile again to point to the new code directory, and type make clean followed by make.
   (b) Are you sure to have recompiled radmc3d again *and* installed it (by going in src/ and typing make install)?
   (c) Try going back to the old version and recheck that the model works well there. If that works, and the above tricks don't fix the problem, then it may be a bug. Please contact the author.

2. **After updating RADMC-3D to a new version: the new features are not present/working.**
   Maybe again the Makefile issue. See point 1 above.

3. **After updating RADMC-3D to a new version: model based on userdef_module fails to compile**
   If you switch to a new version of the code and try to 'make' an earlier model that uses the userdef_module.f90, it might sometimes happen that the compilation fails because some subroutine userdef_*** is not known (here *** is some name). Presumably what happened is that a new user-defined functionality has been added to the code, and the corresponding subroutine userdef_*** has been added to the userdef_module.f90. If, however, in your own userdef_module.f90 this subroutine is not yet built in, then the compiler can't find this subroutine and complains. Solution: just add a dummy subroutine to your userdef_module.f90 with that name (have a look at the userdef_module.f90 in the src/ directory). Then recompile and it should now work.

4. **After switching back from the userdef_module.f90-driven model setup to the original IDL-driven setup style, suddenly lots of IDL routines produce problems or do not give the right results.**
   Check if an old radmc3d executable is still present. If so, remove it, because the way the IDL tools check which radmc3d executable to use (local or global) is by checking if a local radmc3d executable is present.

5. **The viewimage GUI aborts with message "Aborting: RADMC-3D does not respond..."**
   This means that the radmc3d code, which is a child process of IDL, has unexpectedly quit. Since the standard IO channel of radmc3d is used for commincation with IDL, the usual standard output is now redirected to the file radmc3d.out. Have a look at that file to see what caused radmc3d to abort.

6. **RADMC-3D stops with message "ERROR: dust_density.inp does not have same number of cells as the grid. -32768 32768" or similar**
   This message means that the file dust_density.inp specifies the dust density in a different number of cells than which is specified in the amr_grid.inp file. In the above example this is in fact caused by the fact that

in the IDL routines for generating the setup the number of cells got an overflow over the maximum range of two-byte signed integers. To avoid this, use long integers. In IDL this is done like `nx=100L` instead of `nx=100`.

7. **While reading an input file, RADMC-3D says "Fortran runtime error: End of file"**
   This can of course have many reasons. Some common mistakes are:
   - In `amr_grid.inp` you may have specified the coordinates of the nx*ny*nz grid centers instead of (nx+1)*(ny+1)*(nz+1) grid cell interfaces.
   - You may have no line feed at the end of one of the ascii input files. Some fortran compilers can read only lines that are officially ended with a return or line feed. Solution: Just write an empty line at the end of such a file.

8. **My changes to the main code do not take effect**
   Did you type, in the `src/` directory, the full `make install`? If you type just `make`, then the code is compiled but not installed as the default code.

9. **My userdef_module.f90 stuff does not work**
   If you run `radmc3d` with own userdefined stuff, then you must make sure to run the right executable. Just typing `radmc3d` in the shell might cause you to run the standard compilation instead of your special-purpose one. Try typing `./radmc3d` instead, which forces the shell to use the local executable.

10. **When I make images from the command line, they take longer than with viewimage.pro**
    If you make images with `radmc3d image` (plus some keywords) from the command line, the default is that a flux-conserving method of ray-tracing is used, which is called recursive sub-pixeling (see Section 9.6). This takes, under some circumstances, much longer than if you make images without recursive sub-pixeling. In the `viewimage` widget (see Section 14.3) the default is to use no sub-pixeling (though by pressing the "preview" button off, the sub-pixeling is used again). You can make an image without sub-pixeling with the command-line option `nofluxcons`.

11. **My line channel maps (images) look bad**
    If you have a model with non-zero gas velocities, and if these gas velocities have cell-to-cell differences that are larger than or equal to the intrinsic (thermal+microturbulent) line width, then the ray-tracing will not be able to pick up signals from intermediate velocities. In other words, because of the discrete gridding of the model, only discrete velocities are present, which can cause numerical problems. There are two possible solutions to this problem. One is the wavelength band method described in Section 9.5. But a more systematic method is the "doppler catching" method described in Section 7.6 (which can be combined with the wavelength band method of Section 9.5 to make it even more perfect).

12. **My line spectra look somewhat noisy**
    If you include dust continuum scattering (Section 6.5) then the ray-tracer will perform a scattering Monte Carlo simulation at each wavelength. If you look at lines where dust scattering is still a strong source of emission, and if `nphot_scat` (Section 6.5.4) is set to a low value, then the different random walks of the photon packages in each wavelength channel may cause slightly different resulting fluxes, hence the noise.

13. **My dust continuum images look very noisy/streaky: many "lines" in the image**
    There are two possible reasons:
    (a) *Photon noise in the thermal Monte Carlo run:* If you have too few photon packages for the thermal Monte Carlo computation (see Chapter 6), then the dust temperatures are simply not well computed. This may give these effects. You must then increase `nphot` in the `radmc3d.inp` file to increase the photon statistics for the thermal Monte Carlo run.
    (b) *Photon noise in the scattering Monte Carlo run:* If you are making an image at a wavelength at which the disk is not emitting much thermal radiation, then what you will see in the image is scattered light. `RADMC-3D` makes a special Monte Carlo run for scattered light before each image. This Monte Carlo run has its own variable for setting the number of photon packages: `nphot_scat`. If this value is set too low, then you can see individual "photon"-trajectories in the image, making the image look bad. It is important to note that this can only be remedied by increasing `nphot_scat` (in the `radmc3d.inp` file, see Section 6.5.4), not by setting `nphot` (which is the number of photon packages for the thermal Monte Carlo computation).

Appendix A

Main input and output files of RADMC-3D

RADMC-3D is written in fortran-90. It is written in such a way that the user prepares input files (ending in `.inp`) for the program and then calls `radmc3d` with some command-line options. The program then reads the input files, and based on the command-line options will perform a certain calculation, and finally outputs the results to output files (ending in `.out`) or intermediate files (ending in `.dat`) which need further processing. In principle the user therefore needs to compile the program only once, and can then use the executable from that point onward. In this chapter we will describe the various input/output and intermediate files and their formats. Just for clarity: the IDL routines in the `idl/` directory are only meant to make it easier for the user to prepare the `.inp` files, and to make sense of the `.out` and `.dat` files. They are not part of the main code `radmc3d`.

A few comments on RADMC-3D input and output files:

- Most (though not all) files start with a *format number*. This number simply keeps track of the version of the way the information is stored the file. The idea is that if new versions of RADMC-3D come out in the future, it would be good to have the possibility that new information is added to the files. The format number is there to tell RADMC-3D whether a file is the new version or still an older version. NOTE: Do not confuse *format number* with *unformatted/formatted I/O* (see below for the latter). These are unrelated issues.
- RADMC-3D is no longer backward compatible with the older RADMC code input files. It has proven to be too messy to maintain this option.
- RADMC-3D has four types of I/O files:
  1. Files ending with `.inp` or `.uinp` are input files that allow the user to specify to RADMC-3D which problem to solve.
  2. Files ending with `.dat` or `.udat` are intermediate files that are typically created by RADMC-3D itself, but can also be read by RADMC-3D for further processing. For instance, the dust temperature is computed by the Monte Carlo method, but can also be read in later for ray-tracing.
  3. Files ending with `.out` or `.uout` are final products of RADMC-3D, such as an image or spectrum.
  4. File ending with `.info` are small files containing some numbers that are useful to better interpret the output files of RADMC-3D. They are typically not very important for every-day use.
- For many of the I/O files RADMC-3D can read and write both formatted (i.e. text style: ascii) files and fortran-style unformatted files. Whether a file is text-style (user-readable) or fortran-style-unformatted (more compact data storage) is specified by the file extension:
  1. Files ending in `.inp`, `.dat` or `.out` are written in text style, i.e. they are ascii files ("formatted output"). They are human-readable lists of numbers or symbols. Typically this I/O style is useful for testing and getting used to the code, so that the user can see exactly (by reading the I/O files in an editor) what numbers are being processed. But the severe drawback is that these files are very large compared to the information they contain.
  2. Files ending in `.uinp`, `.udat` or `.uout` are written in fortran-style *unformatted* form. Here each double-precision variable takes up 8 bytes, each single-precision variable 4 bytes, each normal integer 4 bytes,

each long integer 8 bytes etc. This form of I/O is much more compact than the unformatted style and thus preferable for big models. Note that not all files have this unformatted option. Also note that fortran-style unformatted files work with *records*, in contrast to C-style unformatted (which is *really* unformatted). This makes the fortran-style unformatted I/O a bit more tricky. Each write statement in fortran produces a single record. A record is stored in the file as a block of data that starts with 2 bytes telling which length the record has (in bytes), then contains the data that are written, and ends with another 2 bytes giving the same length of the record that it now closes. C-style unformatted files only write the actual data. So for each unformatted write statement in fortran 4 more bytes are stored which are essentially useless (and can be confusing). In IDL there is a special keyword `/f77_unformatted` that you can add to the `open` statement to make IDL read unformatted files like fortran, so that these additional bytes are read and ignored or written. Note that some compilers may write 4 bytes at the start and end of a record (gfortran used to do this, but this is now turned back for backward compatibility reasons). But most compilers stick (fortunately) to the old 2-byte at start and end of a record convention. Because of the complexity of the fortran record-based I/O style, and because these records therefore have a maximum length (which is 65536 bytes) the unformatted I/O of RADMC-3D is a bit subtle. Details are described in the sections below for each file, and the IDL routines are equipped to read/write the correct form of these files.

A.1 INPUT: radmc3d.inp

The `radmc3d.inp` file is a namelist file with the main settings for RADMC-3D[1]. The namelist is not a standard Fortran namelist style, but a simple *name = value* list. If a name is not specified, the default values are taken. So if the `radmc3d.inp` file is empty, then all settings are standard. Note that some of these settings can be overwritten by command-line options!! Here is a non-exhaustive list of the variables that can be set.

- `incl_dust` (default: depends on which input files are present)
  Normally RADMC-3D will recognize automatically whether dust continuum emission, absorption and scattering must be included: if e.g. a file called `dustopac.inp` is present, it assumes that the dust must be included. But with this flag you can explicitly tell RADMC-3D whether it must be included (1) or not (0).
- `incl_lines` (default: depends on which input files are present)
  Normally RADMC-3D will recognize automatically whether line emission and absorption must be included: if e.g. a file called `lines.inp` is present, it assumes that molecular/atomic lines must be included. But with this flag you can explicitly tell RADMC-3D whether it must be included (1) or not (0).
- `incl_freefree` (default: 0)
  If 1, then include free-free emission and absorption (Bremsstrahlung). For this, the gas temperature must be known (but see option `tgas_eq_tdust` below).
- `nphot` or `nphot_therm` (default: 100000)
  The number of photon packages used for the thermal Monte Carlo simulation.
- `nphot_scat` (default: 100000)
  The number of photon packages for the scattering Monte Carlo simulations, done before image-rendering.
- `nphot_spec` (default: 10000)
  The number of photon packages for the scattering Monte Carlo simulations, done during spectrum-calculation. This is actually the same functionality as for `nphot_scat`, but it is used (and only used) for the spectrum and SED calculations. The reason to have a separate value for this is that for spectra you may not need as many photon packages as for imaging, because you anyway integrate over the images. Many of the annoying "stripe noise" in images when using insufficiently large `nphot_scat` will cancel each other out in the flux calculation. So `nphot_spec` is usually taken smaller than `nphot_scat`.
- `iseed` (default: -17933201) *[Fine-tuning only]*
  A starting value of the random seed for the Monte Carlo simulation.

[1] Originally this was called the `radmc.inp` file. If no `radmc3d.inp` is present, but `radmc.inp` *is* present, then RADMC-3D will read the latter. But if both are present, then RADMC-3D will read the `radmc3d.inp` file.

- `ifast` (default: 0) *[Fine-tuning only]*
  By setting this to 1 or 2 you will get a faster Monte Carlo simulation, at the cost of being less accurate.

- `enthres` (default: 0.01) *[Fine-tuning only]*
  This is the fraction by which the energy in each cell may increase before the temperature is recalculated in the Monte Carlo simulation. The smaller this value, the more accurate the thermal Monte Carlo simulation, but the more computationally costly. 0.01 has proven to be fine.

- `itempdecoup` (default: 1)
  If set to 0, then the temperatures of all coexisting dust species are always forced to be the same. If 1, then each dust species is thermally independent of the other.

- `istar_sphere` (default: 0)
  If 0 (=default), then all stars are treated as point-sources. If 1, then all stars are treated as finite-size spheres. This mode is more accurate and more realistic, but the applications are a bit more restricted. Such finite-size stars are (for technical reasons) not always allowed anywhere in the model. But for problems of circumstellar disks and envelopes in spherical coordinates, it is recommended to set this to 1. Typically, if a star is outside the grid (in spherical coordinates this can also be at the origin of the coordinate system, as long as the inner radius of the coordinate system is larger than the stellar radius!) the use of the finite-size star mode is always possible. But if the star is on the grid, there are technical limitations.

- `ntemp` (default: 1000) *[Fine-tuning only]*
  The temperatures are determined in the Monte Carlo method using tabulated pre-computed integrals. This saves time. This is the number of temperatures for which this is precalculated. The temperatures are sampled in a logarithmic way, i.e. log(temp) is linearly equally spaced between log(temp0) and log(temp1), see below.

- `temp0` (default: 0.01) *[Fine-tuning only]*
  The lowest pre-calculated temperature.

- `temp1` (default: 1e5) *[Fine-tuning only]*
  The highest pre-calculated temperature.

- `scattering_mode_max`
  When `radmc3d` reads the dust opacity files it checks if one or more of the opacity files has scattering opacity included. If yes, the `scattering_mode` will automatically be set to 1. It will also check if one or more includes *anisotropic* scattering. If yes, the `scattering_mode` will automatically be set to 2. But the user *may* nevertheless want to exclude anisotropic scattering or exclude scattering altogether (for instance for testing purposes, or if the user knows from experience that the scattering or anisotropic nature of scattering is not important for the problem at hand). Rather than editing the opacity files to remove the scattering and/or Henyey-Greenstein g-factors, you can limit the value that `radmc3d` is allowed to make `scattering_mode` by setting the variable `scattering_mode_max`. If you set `scattering_mode_max=0` then no matter what opacity files you have, scattering will not be treated. If you set `scattering_mode_max=1`, then no matter what opacity files you have, scattering will be treated in an isotropic way.

- `unformatted` (default: 0)
  If this is set to 0, then all output `.dat` and `.out` files will be written in ascii format. If this is set to 1, then some of the `.dat` and `.out` files (only the big ones) will be written in fortran-style unformatted. NOTE: For the input (`.inp`) files and when intermediate (`.dat`) files are read into RADMC-3D, the the extension of the file automatically tells if the file is formatted or unformatted: any file ending in `.uinp` or `.udat` or `.uout` is in unformatted style, while any file ending in `.inp` or `.dat` or `.out` is in ascii style. NOTE: The unformatted I/O is subtle, because of fortran-style record-based I/O. See the subsections on the various input/output files for details.

- `camera_tracemode` (default: 1)
  If `camera_tracemode`=-1, the images that are rendered by RADMC-3D will instead by the column depth traced along each ray. If `camera_tracemode`=-2, the images that are rendered by RADMC-3D will instead by the continuum optical depth traced along each ray. By default `camera_tracemode`=1, which is the normal mode, where real images are being created.

- `camera_nrrefine` (default: 100)
  For images: to assure that flux is correctly sampled, the image pixels will not just be rendered one ray per pixel. Instead, if necessary, a pixel will spawn 2x2 sub-pixels recursively (each of which can split again into 2x2 until the required resolution is obtained) so as to assure that the flux in each pixel is correct. Nrrefine tells how deep RADMC-3D is allowed to recursively refine. 100 is therefore effectively infinite. Putting this to 0 means that you go back to 1 ray per pixel, which is fast, but may seriously misrepresent the flux in each pixel. NOTE: The recursive pixel refinement is done internally and the user will not notice it except for getting better answers. In the output image only the original pixels are shown; all subpixels have been integrated over to get the flux of the original pixel. So you can keep this to the default value of 100 without having to worry about handling complex data structures. The only drawback is that it may take longer to compute. See Section 9.6 for more details.

- `camera_refine_criterion` (default: 1.0) *[Fine-tuning only]*
  Setting this value to smaller than 1 means that you refine the recursive pixeling until a tighter criterion is met. The smaller this value, the more accurate the fluxes in each pixel, but the longer it takes to render. See Section 9.6 for more details.

- `camera_incl_stars` (default: 1)
  If 0, then only the interstellar/circumstellar material is rendered for the images and spectra. If 1, then also the stellar flux is included in the spectra and images. So far, stars are treated always as point sources.

- `camera_starsphere_nrpix` (default: 20) *[Fine-tuning only]*
  For rectangular images and for the spectra/SEDs (but not for spectra/SEDs created with circular pixel arrangements), this number tells RADMC-3D how much it should do sub-pixeling over the stellar surface. That is: 20 means that at least 20 sub-pixels are assured over the stellar surface. This is important for flux conservation (see Section 9.6).

- `camera_spher_cavity_relres` (default: 0.05) *[Fine-tuning only]*
  Determines the size of sub-pixels inside the inner grid radius of spherical coordinates.

- `camera_localobs_projection` (default: 1)
  (Only for local observer mode) The type of projection on the sphere of observation.

- `camera_min_dangle` (default 0.05) *[Fine-tuning only]*
  Fine-tuning parameter for recursive subpixeling, for spherical coordinates, assuring that not too fine subpixeling would slow down the rendering of images or spectra too much.

- `camera_max_dangle` (default 0.3) *[Fine-tuning only]*
  Fine-tuning parameter for recursive subpixeling, for spherical coordinates, preventing that too coarse subpixeling would reduce the accuracy.

- `camera_min_dr` (default 0.003) *[Fine-tuning only]*
  Fine-tuning parameter for recursive subpixeling, for spherical coordinates, assuring that not too fine subpixeling would slow down the rendering of images or spectra too much.

- `camera_diagnostics_subpix` (default: 0)
  Setting this to 1 forces RADMC-3D to write out a file called `subpixeling_diagnostics.out` which contains four columns, for respectivly: `px,py,pdx,pdy`, i.e. the pixel position and its size. This is for all pixels, including the sub-pixels created during the recursive subpixeling procedure (Section 9.6.2). This allows the user to find out if the recursive subpixeling went well or if certain areas were over/under-resolved. This is really only meant as a diagnostic.

- `camera_secondorder` (default: 0)
  If set to 1, RADMC-3D will interpolate all emission/absorption quantities to the cell corners, and then use a second order integration routine with bilinear interpolation of the source terms to integrate the ray-tracing formal transfer equations. See Section 9.8 for more information about the second order integration: It is recommended to read it!

- `camera_interpol_jnu` (default: 0) *[Fine-tuning only]*
  Fine-tuning parameter for ray-tracing, only used for when second order integration is done (i.e. if `camera_secondorder`=1).

If 0 (default), then the source function S_ν is the one that is interpolated on the grid, while if 1, then the emissivity j_ν is the one that is interpolated on the grid. The differences are minimal, but if strange results appear (when using second order integration) then you may want to experiment a bit with this parameter.

- `mc_weighted_photons` (default: 1) *[Fine-tuning only]*
  If `mc_weighted_photons`=1 (default) then in Monte Carlo simulations not all photon packages will have the same energy. The energy will be weighted such that each star or emission mechanism will emit, on average, the same number of photon packages. As an example: If you have a stellar binary consisting of an O-star surrounded by a Brown Dwarf, but the Brown Dwarf is surrounded by a disk, then although the O star is much brighter than the O-star, the very inner regions of the Brown Dwarf disk is still predominantly heated by the Brown Dwarf stellar surface, because it is much closer to that material. If you do not have weighted photon packages, then statistically the Brown Dwarf would emit perhaps 1 or 2 photon packages, which makes the statistics of the energy balance in the inner disk very bad. By `mc_weighted_photons`=1 both the Brown Dwarf and the O-star will each emit the same number of photon packages; just the energy of the photon packages emitted by the Brown Dwarf are much less energetic than those from the O-star. This now assures a good photon statistics everywhere.

- `optimized_motion` (default: 0) *[Fine-tuning only]*
  If `optimized_motion` is set to 1, then RADMC-3D will try to calculate the photon motion inside cells more efficiently. This may save computational time, but since it is still not very well tested, please use this mode with great care! It is always safer not to use this mode.

- `lines_mode` (default: -1)
  This mode determines how the level populations for line transfer are computed. The default is -1, which means: Local Thermodynamic Equilibrium (LTE). For other modes, please consult Chapter 7.

- `lines_maxdoppler` (default: 0.3) *[Fine-tuning only]*
  If the doppler catching mode is used, this parameter tells how fine RADMC-3D must sample along the ray, in units of the doppler width, when a line is doppler-shifting along the wavelength-of-sight.

- `lines_partition_ntempint` (default 1000) *[Fine-tuning only]*
  Number of temperature sampling points for the internally calculated partition function for molecular/atomic lines.

- `lines_partition_temp0` (default 0.1) *[Fine-tuning only]*
  Smallest temperature sampling point for the internally calculated partition function for molecular/atomic lines.

- `lines_partition_temp1` (default 1E5) *[Fine-tuning only]*
  Largest temperature sampling point for the internally calculated partition function for molecular/atomic lines.

- `lines_show_pictograms` (default 0)
  If 1, then print a pictogram of the levels of the molecules/atoms.

- `tgas_eq_tdust` (default: 0)
  By setting `tgas_eq_tdust=1` you tell `radmc3d` to simply read the `dust_temperature.inp` file and then equate the gas temperature to the dust temperature. If multiple dust species are present, only the first species will be used.

- `subbox_nx, subbox_ny, subbox_nz, subbox_x0, subbox_x1, subbox_y0, subbox_y1, subbox_z0, subbox_z1`
  Parameters specifying the subbox size for the subbox extraction. See Section 15.1 for details.

A.2 INPUT (required): amr_grid.inp,amr_grid.uinp

This is the file that specifies what the spatial grid of the model looks like. See Chapter 10. This file is essential, because most other `.inp` and `.dat` files are simple lists of numbers which do not contain any information about the grid. All information about the grid is contained in the `amr_grid.inp`, also for non-AMR regular grids. Note that in the future we will also allow for unstructured grids. The corresponding grid files will then be named differently. Both

the formatted style of this file (`amr_grid.inp`) and the unformatted style (`amr_grid.uinp`) have the following information. Each line represents a row in the formatted style and a record in the unformatted style.

There are three possible AMR grid styles:

- Regular grid: No mesh refinement. This is grid style 0.
- Oct-tree-style AMR ("Adaptive Mesh Refinement", although for now it is not really "adaptive"). This is grid style 1.
- Layer-style AMR. This is grid style 10.

A.2.1 Regular grid

For a regular grid, without grid refinement, the `amr_grid.inp` looks like:

```
iformat                                   <=== Typically 1 at present
0                                         <=== Grid style (regular = 0)
coordsystem
gridinfo
incl_x       incl_y       incl_z
nx           ny           nz
xi[1]        xi[2]        xi[3]       ........  xi[nx+1]
yi[1]        yi[2]        yi[3]       ........  yi[ny+1]
zi[1]        zi[2]        zi[3]       ........  zi[nz+1]
```

The meaning of the entries are:

iformat: The format number, at present 1. For unformatted files this must be 4-byte integer.

coordsystem: If `coordsystem`<100 the coordinate system is cartesian. If 100<=`coordsystem`<200 the coordinate system is spherical (polar). If 200<=`coordsystem`<300 the coordinate system is cylindrical. For unformatted files this must be 4-byte integer.

gridinfo: If `gridinfo`==1 there will be abundant grid information written into this file, possibly useful for post-processing routines. Typically this is redundant information, so it is advised to set `gridinfo`=0 to save disk space. In the following we will assume that `gridinfo`=0. For unformatted files this must be 4-byte integer.

incl_x, incl_y, incl_z: These are either 0 or 1. If 0 then this dimension is not active (so upon grid refinement no refinement in this dimension is done). If 1 this dimension is fully active, even if the number of base grid cells in this direction is just 1. Upon refinement the cell will also be splitted in this dimension. For unformatted files these numbers must be 4-byte integer.

nx, ny, nz: These are the number of grid cells on the base grid in each of these dimensions. For unformatted files these numbers must be 4-byte integer.

xi[1] ... xi[nx+1]: The edges of the cells of the base grid in x-direction. For `nx` grid cells we have `nx+1` cell walls, hence `nx+1` cell wall positions. For unformatted files these numbers must be 8-byte reals (=double precision).

yi[1] ... yi[ny+1]: Same as above, but now for y-direction.

zi[1] ... zi[nz+1]: Same as above, but now for z-direction.

Example of a simple 2x2x2 regular grid in cartesian coordinates:

```
1
0
1
0
1  1  1
2  2  2
-1.  0. 1.
-1.  0. 1.
-1.  0. 1.
```

A.2.2 Oct-tree-style AMR grid

For a grid with oct-tree style grid refinement, the `amr_grid.inp` looks like:

```
iformat                                  <=== Typically 1 at present
1                                        <=== Grid style (1 = Oct-tree)
coordsystem
gridinfo
incl_x       incl_y       incl_z
nx           ny           nz
levelmax     nleafsmax    nbranchmax     <=== This line only if grid style == 1
xi[1]        xi[2]        xi[3]       ........  xi[nx+1]
yi[1]        yi[2]        yi[3]       ........  yi[ny+1]
zi[1]        zi[2]        zi[3]       ........  zi[nz+1]
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
(0/1)                   <=== 0=leaf, 1=branch (only if amrstyle==1)
...
...
```

The keywords have the same meaning as before, but in addition we have:

(0/1): *NOTE: Only for amrstyle==1.* These are numbers that are either 0 or 1. If 0, this means the current cell is a leaf (= a cell that is not refined and is therefore a "true" cell). If 1, the current cell is a branch with 2 (in 1-D), 4 (in 2-D) or 8 (in 3-D) daughter cells. In that case the next (0/1) numbers are for these daughter cells. In other words, we immediately recursively follow the tree. The order in which this happens is logical. In 3-D the first daughter cell is (1,1,1), then (2,1,1), then (1,2,1), then (2,2,1), then (1,1,2), then (2,1,2), then (1,2,2) and finally (2,2,2), where the first entry represents the x-direction, the second the y-direction and the third the z-direction. If one or more of the daughter cells is also refined (i.e. has a value 1), then first this sub-tree is followed before continuing with the rest of the daughter cells. If we finally return to the base grid at some point, the next (0/1) number is for the next base grid cell (again possibly going into this tree if the value is 1). The order in which the base grid is scanned in this way is from `1` to `nx` in the innermost loop, from `1` to `ny` in the middle loop and from `1` to `nz` in the outermost loop. For unformatted files these numbers must be 4-byte integers, one record per number.

NOTE: For this file the unformatted style is presumably not so useful, because for technical reasons each of the (0/1) numbers must be a separate record, requiring 12 bytes. The formatted version is smaller: each line being only 2 bytes (one character 0 or 1 and a return). In the future I will try to make this more efficient, but for now the user is advised to just use the unformatted style.

Example of a simple 1x1x1 grid which is refined into 2x2x2 and for which the (1,2,1) cell is refined again in 2x2x2:

```
1
1
1
0
1  1  1
1  1  1
10 100 100
-1.  1.
-1.  1.
-1.  1.
1
0
0
1
0
0
0
0
0
```

```
0
0
0
0
0
0
0
0
```

A.2.3 Layer-style AMR grid

For a grid with layer-style grid refinement, the `amr_grid.inp` looks like:

```
iformat                                  <=== Typically 1 at present
10                                       <=== Grid style (10 = layer-style)
coordsystem
gridinfo
incl_x       incl_y       incl_z
nx           ny           nz
nrlevels     nrlayers                    <=== This line only if grid style == 10
xi[1]        xi[2]        xi[3]       ........  xi[nx+1]
yi[1]        yi[2]        yi[3]       ........  yi[ny+1]
zi[1]        zi[2]        zi[3]       ........  zi[nz+1]
parentid     ix  iy  iz   nx  ny  nz
parentid     ix  iy  iz   nx  ny  nz
parentid     ix  iy  iz   nx  ny  nz
parentid     ix  iy  iz   nx  ny  nz
.
.
.
```

The keywords have the same meaning as before, but in addition we have:

nrlevels: How many levels you plan to go, where nrlevels==0 means no refinement, nrlevels==1 means one level of refinement (factor of 2 in resolution), etc.

nrlayers: How many layers do you have, with nrlayers==0 means no refinement, nrlayers==1 means one layer of refinement (factor of 2 in resolution), etc.

parentid: (For each layer) The parent layer for this layer. parentid==0 means parent is base grid. First layer has id==1.

ix, iy, iz: (For each layer) The location in the parent layer where the current layer starts.

nx, ny, nz: (For each layer) The size of the layer as measured in units of the the parent layer. So the actual size of the current layer will be (in 3-D): `2*nx, 2*ny, 2*nz`. In 2-D, with only the x- and y- dimensions active, we have a size of `2*nx, 2*ny` with of course size 1 in z-direction.

As you can see, this is a much easier and more compact way to specify mesh refinement. But it is also less "adaptive", as it is always organized in square/cubic patches. But it is much easier to handle for the user than full oct-tree refinement.

Note that this layer-style refinement is in fact, internally, translated into the oct-tree refinement. But you, as the user, will not notice any of that. The code will input and output entirely in layer style.

NOTE: The layers must be specify in increasing refinement level! So the first layer (layer 1) must have the base grid (layer 0) as its parent. The second layer can have either the base grid (layer 0) or the first layer (layer 1) as parent, etc. In other words: the parent layer must always already have been specified before.

Example of a simple 2-D 4x4 grid which has a refinement patch in the middle of again 4x4 cells (=2x2 on the parent grid), and a patch of 2x2 (=1x1 on the parent grid) starting in the upper left corner:

```
1
100
1
0
```

```
1  1  0
4  4  1
1  2
-2. -1. 0. 1. 2.
-2. -1. 0. 1. 2.
-0.5 0.5
0  2  2  1  2  2  1
0  1  1  1  1  1  1
```

This has just one level of refinement, but two patches at level 1.

Anothe example: two recursive layers. Again start with a 2-D 4x4 grid, now refine it in the middle with again a 4x4 sub-grid (=2x2 on the parent grid = layer 0) and then again a deeper layer of 4x4 (=2x2 on the parent grid = layer 1) this time starting in the corner:

```
1
100
1
0
1  1  0
4  4  1
2  2
-2. -1. 0. 1. 2.
-2. -1. 0. 1. 2.
-0.5 0.5
0  2  2  1  2  2  1
1  1  1  1  2  2  1
```

Note that with this layer-style grid, the input data will have to be speficied layer-by-layer: first the base grid, then the first layer, then the second etc. This is worked out in detail for `dust_density.inp` in Section A.3. This will include redundant data, because you specify the data on the entire base grid, also the cells that later will be replaced by a layer. Same is true for any layer that has sub-layers. The data that is specified in these regions will be simply ignored. But for simplicity we do still require it to be present, so that irrespective of the deeper layers, the data in any layer (including the base grid, which is layer number 0) is simply organized as a simple data cube. This redundancy makes the input and output files larger than strictly necessary, but it is much easier to handle as each layer is a datacube. For memory/hardisk-friendly storage you must use the oct-tree refinement instead. The layers are meant to make the AMR much more accessible, but are somewhat more memory consuming.

A.3 INPUT (required for dust transfer): dust_density.inp, dust_density.uinp

This is the file that contains the dust densities. It is merely a list of numbers. Their association to grid cells is via the file `amr_grid.(u)inp`. Each dust species will have its own density distribution, completely independently of the others. That means that at each position in space several dust species can exist, and the density of these can be fully freely specified. The structure of this file is as follows. For formatted style (`dust_density.inp`):

```
iformat                                  <=== Typically 1 at present
nrcells
nrspec
density[1,ispec=1]
..
density[nrcells,ispec=1]
density[1,1,1,ispec=2]
..
..
..
density[nrcells,ispec=nrspec]
```

Here `nrspec` is the number of independent dust species densities that will be given here. It can be 1 or larger. If it is 1, then of course the `density[1,1,1,ispec=2]` and following lines are not present in the file. The `nrcells` is the number of cells. For different kinds of grids this can have different meaning. Moreover, for different kinds of grids the order in which the density values are given is also different. So let us now immediately make the following distinction (See Chapter 10 on the different kinds of grids):

- *For regular grid and oct-tree AMR grids:*
  The value of `nrcells` denotes the number of *true* cells, excluding the cells that are in fact the parents of 2x2x2 subcells; i.e. the sum of the volumes of all true cells (=leafs) adds up to the volume of the total grid). The order of these numbers is always the same "immediate recursive subtree entry" as in the `amr_grid.(u)inp` (Section A.2).

- *For layer-style AMR grids:*
  The value of `nrcells` denotes the number of values that are specified. This is generally a bit more than the true number of cells specified in the oct-tree style AMR (see above). In the layer-style AMR mode you specify the dust density (or any other value) first at all cells of the base grid (whether a cell is refined or not does not matter), the at all cells of the first layer, then the second layer etc. Each layer is a regular (sub-)grid, so the order of the values is simply the standard order (same as for regular grids). This means, however, that the values of the density in the regular grid cells that are replaced by a layer are therefore redundant. See Section 10.4.1 for a discussion of this redundancy. The main advantage of this layer-style grid refinement is that the input and output always takes place on *regular* grids and subgrids (=layers). This is much easier to handle than the complexities of the oct-tree AMR.

A.3.1 Example: `dust_density.inp` for a regular grid

Now let us look at an example of a `dust_density.inp` file, starting with one for the simplified case of a regular 3-D grid (see Sections A.2.1 and 10.1):

```
iformat                                  <=== Typically 1 at present
nrcells
nrspec
density[1,1,1,ispec=1]
density[2,1,1,ispec=1]
..
density[nx,1,1,ispec=1]
density[1,2,1,ispec=1]
..
..
density[nz,ny,nz,ispec=1]
density[1,1,1,ispec=2]
..
..
..
density[nz,ny,nz,ispec=nrspec]
```

For unformatted style (`dust_density.uinp`) the structure is:

```
iformat       reclen
nrcells       nrspec
density[1,ispec=1]        ... density[reclen,ispec=1]
density[reclen+1,ispec=1] ... density[2*reclen,ispec=1]
..
..... density[nrcells,ispec=1] ... 0   0   0     <==== fill with 0 until end of record
density[1,ispec=2]        ... density[reclen,ispec=2]
density[reclen+1,ispec=2] ... density[2*reclen,ispec=2]
..
..... density[nrcells,ispec=2] ... 0   0   0     <==== fill with 0 until end of record
..
..
```

All integers (`iformat`, `reclen`, `nrcells` and `nrspec`) are 8-byte integers. Here `reclen` is a somewhat arbitrary number between 8 and 65536 which denotes the record length in bytes. It must be a multiple of 8 (which is the length of the double precision real). The data of the density is stored as series of double-precision (8-byte) reals organized in records of `reclen`/8 numbers long. Since the total number of cells `nrcells` is not necessarily divisible by `reclen`/8, it can be that the last record is not full. It will be padded with zeroes until the (fixed) record length is reached. Example: we have a 2x2x2 regular grid and two dust species. The grid contains 8 cells (i.e. `nrcells`=8). Suppose we choose `reclen`=48, i.e. each record contains 6 double precision numbers. Then the first record contains the densities of dust species 1 in cells (1,1,1), (2,1,1), (1,2,1), (2,2,1), (1,1,2), (2,1,2), and the second record will contain the density of dust species 1 in cells (1,2,2) and (2,2,2) and four double precision zeroes to pad the record to

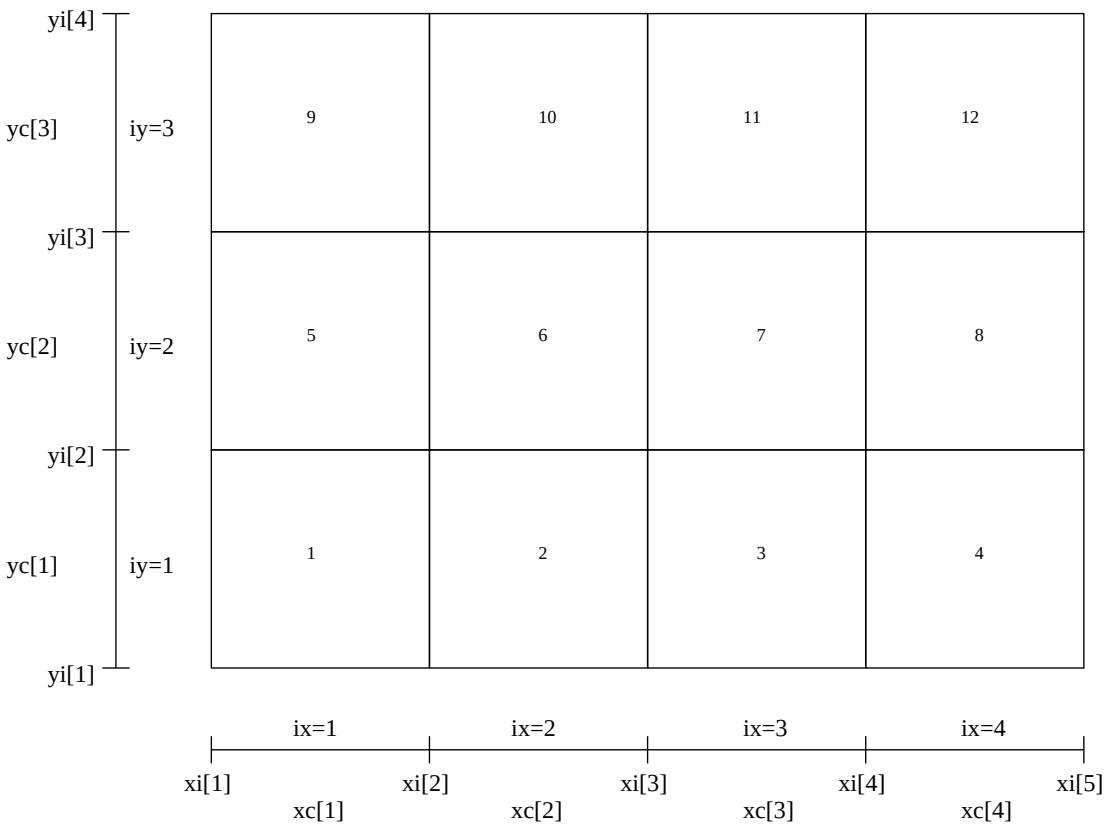


Figure A.1. Example of a regular 2-D grid with `nx`=4 and `ny`=3 (as Fig. 10.1), with the order of the cells shown as numbers in the cells.

6 numbers. Then we repeat the procedure for the second dust species, again yielding a record with 6 densities and one with 2 densities and padded with four zeroes.

A.3.2 Example: `dust_density.inp` for an oct-tree refined grid

For the case when you have an oct-tree refined grid (see Sections A.2.2 and 10.3), the order of the numbers is the same as the order of the cells as specified in the `amr_grid.(u)inp` file (Section A.2). Let us take the example of a simple 1x1x1 grid which is refined into 2x2x2 and for which the (1,2,1) cell is refined again in 2x2x2 (this is exactly the same example as shown in Section A.2.2, and for which the `amr_grid.inp` is given in that section). Let us also assume that we have only one dust species. Then the `dust_density.inp` file would be:

```
iformat                              <=== Typically 1 at present
15                                   <=== 2x2x2 - 1 + 2x2x2 = 15
1                                    <=== Let us take just one dust spec
density[1,1,1]                       <=== This is the first base grid cell
density[2,1,1]
density[1,2,1;1,1,1]                 <=== This is the first refined cell
density[1,2,1;2,1,1]
density[1,2,1;1,2,1]
density[1,2,1;1,2,1]
density[1,2,1;1,1,2]
density[1,2,1;2,1,2]
density[1,2,1;1,2,2]
density[1,2,1;1,2,2]                 <=== This is the last refined cell
density[2,2,1]
density[1,1,2]
density[2,1,2]
density[1,2,2]
density[2,2,2]                       <=== This is the last base grid cell
```

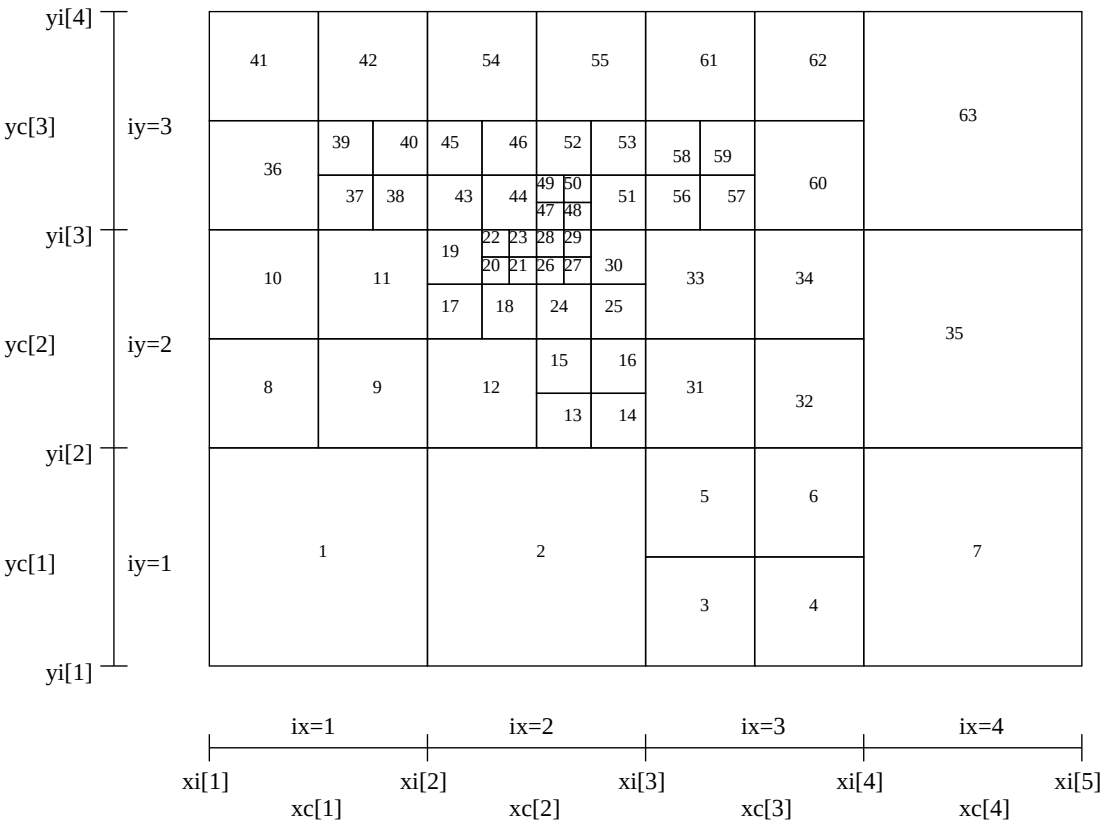


Figure A.2. Example of a 2-D grid with oct-tree refinement (as Fig. 10.2) with the order of the cells shown as numbers in the cells.

A more complex example is shown in Fig. A.2. An unformatted version is also available, in the standard way (see above).

A.3.3 Example: `dust_density.inp` for a layer-style refined grid

For the case when you have an layer-style refined grid (see Sections A.2.3 and 10.4) you specify the density in a series of regular boxes (=layers). The first box is the base grid, the second the first layer, the third the second layer etc. The value `nrcells` now tells the combined sizes of the all the boxes. If we take the second example of Section A.2.3: a simple 2-D 4x4 grid which has a refinement patch (=layer) in the middle of again 4x4 cells, and again one patch of 4x4 this time, however, starting in the upper left corner (see the `amr_grid.inp` file given in Section A.2.3), then the `dust_density.inp` file has the following form:

```
iformat                                 <=== Typically 1 at present
48                                      <=== 4x4 + 4x4 + 4x4 = 48
1                                       <=== Let us take just one dust spec
density[1,1,1,layer=0]
density[2,1,1,layer=0]
density[3,1,1,layer=0]
density[4,1,1,layer=0]
density[1,2,1,layer=0]
density[2,2,1,layer=0]                  <=== This a redundant value
density[3,2,1,layer=0]                  <=== This a redundant value
density[4,2,1,layer=0]
density[1,3,1,layer=0]
density[2,3,1,layer=0]                  <=== This a redundant value
density[3,3,1,layer=0]                  <=== This a redundant value
density[4,3,1,layer=0]
density[1,4,1,layer=0]
density[2,4,1,layer=0]
density[3,4,1,layer=0]
density[4,4,1,layer=0]
```

```
density[1,1,1,layer=1]                  <=== This a redundant value
density[2,1,1,layer=1]                  <=== This a redundant value
density[3,1,1,layer=1]
density[4,1,1,layer=1]
density[1,2,1,layer=1]                  <=== This a redundant value
density[2,2,1,layer=1]                  <=== This a redundant value
density[3,2,1,layer=1]
density[4,2,1,layer=1]
density[1,3,1,layer=1]
density[2,3,1,layer=1]
density[3,3,1,layer=1]
density[4,3,1,layer=1]
density[1,4,1,layer=1]
density[2,4,1,layer=1]
density[3,4,1,layer=1]
density[4,4,1,layer=1]
density[1,1,1,layer=2]
density[2,1,1,layer=2]
density[3,1,1,layer=2]
density[4,1,1,layer=2]
density[1,2,1,layer=2]
density[2,2,1,layer=2]
density[3,2,1,layer=2]
density[4,2,1,layer=2]
density[1,3,1,layer=2]
density[2,3,1,layer=2]
density[3,3,1,layer=2]
density[4,3,1,layer=2]
density[1,4,1,layer=2]
density[2,4,1,layer=2]
density[3,4,1,layer=2]
density[4,4,1,layer=2]
```

An unformatted version is also available, in the standard way (see above).

It is clear that 48 is now the total number of values to be read, which is 16 values for layer 0 (= base grid), 16 values for layer 1 and 16 values for layer 2. It is also clear that some values are redundant (they can have any value, does not matter). But it at least assures that each data block is a simple regular data block, which is easier to handle. Note that these values (marked as redundant in the above example) *must* be present in the file, but they can have any value you like (typically 0).

Note that if you have multiple species of dust then we will still have 48 as the value of `nrcells`. The number of values to be read, if you have 2 dust species, is then simply 2*`nrcells` = 2*48 = 96.

A.4 INPUT/OUTPUT: dust_temperature.dat, dust_temperature.udat

The dust temperature file is an intermediate result of RADMC-3D and follows from the thermal Monte Carlo simulation. It can be used by the user for other purposes (e.g. determination of chemical reaction rates), but also by RADMC-3D itself when making ray-traced images and/or spectra. The user can also produce his/her own `dust_temperature.(u)dat` file (without invoking the Monte Carlo computation) if she/he has her/his own way of computing the dust temperature.

The structure of this file is identical to that of `dust_density.inp` or `dust_density.uinp` (Section A.3), but with density replaced by temperature. We refer to section A.3 for the details. See Chapter B for more details on unformatted I/O.

A.5 INPUT/OUTPUT (only if required): electron_numdens.inp, electron_numdens.uinp

For various gasopacity issues (see Chapter 8) the number density of free electrons may be required. The structure of this file is identical to that of `dust_density.inp` or `dust_density.uinp` (Section A.3), but with density replaced by the electron number density in units of $1/\mathrm{cm}^3$. We refer to chapter 8 for the details. See Chapter B for more details on unformatted I/O.

A.6 INPUT/OUTPUT (only if required): ion_numdens.inp, ion_numdens.uinp,

For various gasopacity issues (see Chapter 8) the number density of ions may be required. The structure of this file is identical to that of `dust_density.inp` or `dust_density.uinp` (Section A.3), but with density replaced by the ion number density in units of $1/\text{cm}^3$. Here we need the overal ion number density. We refer to chapter 8 for the details. See Chapter B for more details on unformatted I/O.

A.7 INPUT (mostly required): stars.inp

This is the file that specifies the number of stars, their positions, radii, and spectra. Stars are sources of netto energy. For the dust continuum Monte Carlo simulation these are a source of photon packages. This file exists only in formatted (ascii) style. Its structure is:

```
iformat                          <=== Put this to 2 !
nstars        nlam
rstar[1]      mstar[1]      xstar[1]      ystar[1]      zstar[1]
  .             .             .             .             .
  .             .             .             .             .
rstar[nstars  mstar[nstars] xstar[nstars] ystar[nstars] zstar[nstars]
lambda[1]
  .
  .
lambda[nlam]
flux[1,star=1]
  .
  .
flux[nlam,star=1]
flux[1,star=2]
  .
  .
flux[nlam,star=2]
  .
  .
  .
  .
flux[nlam,star=nstar]
```

which is valid only if `iformat`==2. The meaning of the variables:

iformat: The format number, at present better keep it at 2. If you put it to 1, the list of wavelengths (see below) will instead be a list of frequencies in Herz.

nstars: The number of stars you wish to specify.

nlam: The number of frequency points for the stellar spectra. At present this must be identical to the number of walvelength points in the file `wavelength_micron.inp` (see Section A.11).

rstar[i]: The radius of star i in centimeters.

mstar[i]: The mass of star i in grams. This is not important for the current version of RADMC-3D, but may be in the future.

xstar[i]: The `x`-coordinate of star i in centimeters (in spherical or cylindrical coordinates this would be the r coordinate).

ystar[i]: The `y`-coordinate of star i (in cartesian coordinates: y in cm, in spherical coordinates: the θ coordinate, in cylindrical coordinates: the ϕ coordinate).

zstar[i]: The `z`-coordinate of star i (in cartesian coordinates: z in cm, in spherical coordinates: the ϕ coordinate, in cylindrical coordinates: the z coordinate in cm).

lambda[i]: Wavelength point i (where $i \in [1,$`nlam`$]$) in microns. This must be identical (!) to the equivalent point in the file `wavelength_micron.inp` (see Section A.11). If not, an error occurs.

flux[i,star=n]: The flux F_ν at wavelength point i for star n in units of erg cm^{-2} s^{-1} Hz^{-1} as seen from a distance of 1 parsec = 3.08572×10^{18} cm (for normalization).

Sometimes it may be sufficient to assume simple blackbody spectra for these stars. If for any of the stars the first (!) flux number (`flux[1,star=n]`) is negative, then the absolute value of this number is taken to be the blackbody temperature of the star, and no further values for this star are read. Example:

```
2
1            100
6.96e10      1.99e33        0.      0.    0.
0.1
  .
  .
1000.
-5780.
```

will make one star, at the center of the coordinate system, with one solar radius, one solar mass, on a wavelength grid ranging from 0.1 micron to 1000 micron (100 wavelength points) and with a blackbody spectrum with a temperature equal to the effective temperature of the sun.

A.8 INPUT (optional): stellarsrc_templates.inp

This is the file that specifies the template spectra for the smooth stellar source distributions. See Section 11.3. The file exists only in formatted (ascii) style. Its structure is:

```
iformat                          <=== Put this to 2 !
ntempl
nlam
lambda[1]
  .
  .
lambda[nlam]
flux[1,templ=1]
  .
  .
flux[nlam,templ=1]
flux[1,templ=2]
  .
  .
flux[nlam,templ=2]
  .
  .
  .
  .
flux[nlam,templ=ntempl]
```

which is valid only if `iformat==2`. The meaning of the variables:

iformat: The format number, at present better keep it at 2. If you put it to 1, the list of wavelengths (see below) will instead be a list of frequencies in Herz.

ntempl: The number of stellar templates you wish to specify.

nlam: The number of frequency points for the stellar template spectra. At present this must be identical to the number of walvelength points in the file `wavelength_micron.inp` (see Section A.11).

lambda[i]: Wavelength point i (where $i \in [1,$`nlam`$]$) in microns. This must be identical (!) to the equivalent point in the file `wavelength_micron.inp` (see Section A.11). If not, an error occurs.

flux[i,templ=n]: The "flux" at wavelength i for stellar template n. The units are somewhat tricky. It is given in units of erg / sec / Hz / gram-of-star. So multiply this by the density of stars in units of gram-of-star / cm^3, and divide by 4*pi to get the stellar source function in units of erg / src / Hz / cm^3 / steradian.

Sometimes it may be sufficient to assume simple blackbody spectra for these stellar sources. If for any of the stellar sources the first (!) flux number (`flux[1,templ=n]`) is negative, then the absolute value of this number is taken to be the blackbody temperature of the stellar source, and the following two numbers are interpreted as the stellar radius and stellar mass respectively. From that, RADMC-3D will then internally compute the stellar template. Example:

```
2
1
100
0.1
  .
  .
1000.
-5780.
6.9600000e+10
1.9889200e+33
```

will tell RADMC-3D that there is just one stellar template, assumed to have a blackbody spectrum with solar effective temperature. Each star of this template has one solar radius, one solar mass.

A.9 INPUT (optional): stellarsrc_density.inp, stellarsrc_density.uinp

This is the file that contains the smooth stellar source densities. If you have the file `stellarsrc_templates.inp` specified (see Section A.8) then you *must* also specify either `stellarsrc_density.inp` or `stellarsrc_density.uinp`. The format of these files are very similar to `dust_density.inp` or `dust_density.uinp` (Section A.3), but instead different dust species, we have different templates. For the rest we refer to Section A.3 for the format. Just replace `ispec` (the dust species) with `itempl` (the template). See Chapter B for more details on unformatted I/O.

A.10 INPUT (optional): external_source.inp

This is the file that specifies the spectrum and intensity of the external radiation field, i.e. the "interstellar radiation field" (see Section 11.4). Its structure is:

```
iformat                          <=== Put this to 2 !
nlam
lambda[1]
  .
  .
lambda[nlam]
Intensity[1]
  .
  .
Intensity[nlam]
```

which is valid only if `iformat`==2. The meaning of the variables:

iformat: The format number, at present better keep it at 2. If you put it to 1, the list of wavelengths (see below) will instead be a list of frequencies in Herz.

nlam: The number of frequency points for the stellar template spectra. At present this must be identical to the number of walvelength points in the file `wavelength_micron.inp` (see Section A.11).

lambda[i]: Wavelength point i (where $i \in [1,$`nlam`$]$) in microns. This must be identical (!) to the equivalent point in the file `wavelength_micron.inp` (see Section A.11). If not, an error occurs.

Intensity[i]: The intensity of the radiation field at wavelength i in units of erg / cm^2 / sec / Hz / steradian.

A.11 INPUT (required): wavelength_micron.inp

This is the file that sets the discrete wavelength points for the continuum radiative transfer calculations. Note that this is not the same as the wavelength grid used for e.g. line radiative transfer. See Section A.12 and/or Chapter 7 for that. This file is only in formatted (ascii) style. It's structure is:

```
nlam
lambda[1]
  .
  .
lambda[nlam]
```

where

nlam: The number of frequency points for the stellar spectra.

lambda[i]: Wavelength point i (where $i \in [1,$`nlam`$]$) in microns.

The list of wavelengths can be in increasing order or decreasing order, but must be monotonically increasing/decreasing.

IMPORTANT: It is important to keep in mind that the wavelength coverage must include the wavelengths at which the stellar spectra have most of their energy, and at which the dust cools predominantly. This in practice means that this should go all the way from 0.1 μm to 1000 μm, typically logarithmically spaced (i.e. equally spaced in logλ). A smaller coverage will cause serious problems in the Monte Carlo run and dust temperatures may then be severely miscalculated. Note that the 0.1 μm is OK for stellar temperatures below 10000 K. For higher temperatures a shorter wavelength lower limit must be used.

A.12 INPUT (optional): camera_wavelength_micron.inp

The wavelength points in the `wavelength_micron.inp` file are the global continuum wavelength points. On this grid the continuum transfer is done. However, there may be various reasons why the user may want to generate spectra on a different (usually more finely spaced) wavelength grid, or make an image at a wavelength that is not available in the global continuum wavelength grid. Rather than redoing the entire model with a different `wavelength_micron.inp`, which may involve a lot of reorganization and recomputation, the user can specify a file called `camera_wavelength_micron.inp`. If this file exists, it will be read into RADMC-3D, and the user can now ask RADMC-3D to make images in those wavelength or make a spectrum in those wavelengths.

If the user wants to make images or spectra of a model that involves gas lines (such as atomic lines or molecular rotational and/or ro-vibrational lines), the use of a `camera_wavelength_micron.inp` file allows the user to do the line+dust transfer (gas lines plus the continuum) on this specific wavelength grid. For line transfer there are also other ways by which the user can specify the wavelength grid (see Chapter 7), and it is left to the user to choose which method to use.

The structure of the `camera_wavelength_micron.inp` file is identical to that of `wavelength_micron.inp` (see Section A.11).

Note that there are also various other ways by which the user can let RADMC-3D choose wavelength points, many of which may be even simpler and more preferable than the method described here. See Section 9.4.

A.13 INPUT (required for dust transfer): dustopac.inp and dustkappa_*.inp or dust_optnk_*.inp

These files specify the dust opacities to be used. More than one can be specified, meaning that there will be more than one co-existing dust species. Each of these species will have its own dust density specified (see Section A.3). The opacity of each species is specified in a separate file for each species. The `dustopac.inp` file tells which file to read for each of these species.

A.13.1 The dustopac.inp file

The file `dustopac.inp` has the following structure, where an example of 2 separate dust species is used:

```
iformat                         <=== Put this to 2
nspec
----------------------------
inputstyle[1]
iquantum[1]                     <=== Put to 0 in this example
<name of dust species 1>
----------------------------
inputstyle[2]
iquantum[2]                     <=== Put to 0 in this example
<name of dust species 2>
```

where:

iformat: Currently the format number is 2, and in this manual we always assume it is 2.

nspec: The number of dust species that will be loaded.

inputstyle[i]: This number tells in which form the dust opacity of dust species i is to be read:

-1 Use the `dustopac_<name>.inp` input file style (see below). Here the opacity is specified at the exact wavelength points given in the file `wavelength_micron.inp`.

1 Use the `dustkappa_<name>.inp` input file style (see below). Here the opacity is specified on its own wavelength grid, and will then be mapped by RADMC-3D by use of interpolation onto the wavelength grid of `wavelength_micron.inp`. Typically one then specifies the opacity here on a fine wavelength grid, so that the mapping onto the (usually courser) global wavelength grid is easy and without artifacts.

100 Use the `dustoptnk_<name>.inp` input file style (see below). Here the optical constants are read on their own wavelength grid. Using Mie theory or CDE the opacities are then computed internally and mapped onto the global continuum wavelength grid from the `wavelength_micron.inp` file.

iquantum[i]: For normal thermal grains this is 0. If, however, this grain species is supposed to be treated as a quantum-heated grain, then non-zero values are to be specified. *NOTE: At the moment the quantum heating is not yet implemented. Will be done in the near future. Until then, please set this to 0.*

<name of dust species i>: This is the name of the dust species (without blank spaces). This name is then glued to the base name of the opacity file (see above). For instance, if the name is `enstatite`, and `inputstyle`==1, then the file to be read is `dustkappa_enstatite.inp`.

A.13.2 The dustopac_<name>.inp files

If for dust species <name> the `inputstyle` in the `dustopac.inp` file is set to -1, then the file dustopac_<name>.inp is sought and read. The structure of this file is:

```
nlam   dummy
kappa_abs[1]
   .
   .
kappa_abs[nlam]
kappa_scat[1]
   .
   .
kappa_scat[nlam]
```

The meaning of these entries is:

nlam: The number of frequency (wavelength) points. This must be *identical* to those of the `wavelength_micron.inp` file or else the code stops.

dummy: Put this number to 1. It is here for historic reasons (and backward compatibility with older RADMC incarnations).

kappa_abs[i]: The absorption opacity at wavelength point i of the `wavelength_micron.inp` wavelength grid, in units of cm^2 per gram of dust.

kappa_scat[i]: The scattering opacity at wavelength point i of the `wavelength_micron.inp` wavelength grid, in units of cm^2 per gram of dust. *NOTE: Here isotropic scattering is assumed.*

Note that the opacities listed in this kind of file belong to the wavelength points in the file `wavelength_micron.inp`. So if you change the `wavelength_micron.inp` file, you also must change the dustopac_<name>.inp files. This is why this kind of opacity specification is somewhat less flexible.

A.13.3 The dustkappa_<name>.inp files

If for dust species <name> the `inputstyle` in the `dustopac.inp` file is set to 1, then the file dustkappa_<name>.inp is sought and read. The structure of this file is:

```
# Any amount of arbitrary
# comment lines that tell which opacity this is.
# Each comment line must start with an # or ; or ! character
iformat                    <== This example is for iformat==2
nlam
lambda[1]        kappa_abs[1]       kappa_scat[1]
   .                  .                  .
   .                  .                  .
lambda[nlam]    kappa_abs[nlam]   kappa_scat[nlam]
```

The meaning of these entries is:

iformat: If iformat==1, then only the lambda and kappa_abs colums are present. In that case the scattering opacity is assumed to be 0, i.e. a zero albedo is assumed. If iformat==2 (which is what is used in the above example) also kappa_scat is read (third column). *In the future also iformat==3 will be active, in which a fourth column is read which lists the Henyey-Greenstein anisotropy factor.*

nlam: The number of wavelength points in this file. This can be any number, and does not have to be the same as those of the `wavelength_micron.inp`. It is typically advisable to have a rather large number of wavelength points.

lambda[i]: The wavelength point i in micron. This does not have to be (and indeed typically is not) the same as the values in the `wavelength_micron.inp` file. Also for each opacity this list of wavelengths can be different (and can be a different quantity of points).

kappa_abs[i]: The absorption opacity in units of cm^2 per gram of dust.

kappa_scat[i]: The scattering opacity in units of cm^2 per gram of dust. Note that this column should only be included if iformat==2 or higher.

Once this file is read, the opacities will be mapped onto the global wavelength grid of the `wavelength_micron.inp` file. Since this mapping always involve uncertainties and errors, a file `dustkappa_<name>.inp_used` is created which lists the opacity how it is remapped onto the global wavelength grid. This is only for you as the user, so that you can verify what RADMC-3D has internally done. Note that if the upper or lower edges of the wavelength domain of the `dustkappa_<name>.inp` file is within the domain of the `wavelength_micron.inp` grid, some extrapolation will have to be done. At short wavelength this will simply be constant extrapolation while at long wavelength a powerlaw extrapolation is done. Have a look at the `dustkappa_<name>.inp_used` file to see how RADMC-3D has done this in your particular case.

A.13.4 The dustoptnk_<name>.inp files

If for dust species <name> the `inputstyle` in the `dustopac.inp` file is set to 100, then the file dustoptnk_<name>.inp is sought and read.

NOTE: For now we discourage this mode as it is insufficiently tested.

A.14 OUTPUT: spectrum.out

Any spectrum that is made with RADMC-3D will be either called `spectrum.out` or *spectrum_<somename>.out* and will have the following structure:

```
iformat                     <=== For now this is 1
nlam

lambda[1]       flux[1]
   .              .
   .              .
lambda[nlam]   flux[nlam]
```

where:

iformat: This format number is currently set to 1.

nlam: The number of wavelength points in this spectrum. This does not necessarily have to be the same as those in the `wavelength_micron.inp` file. It can be any number.

lambda[i]: Wavelength in micron. This does not necessarily have to be the same as those in the `wavelength_micron.inp` file. The wavelength grid of a spectrum file can be completely independent of all other wavelength grids. For standard SED computations for the continuum typically these will be indeed the same as those in the `wavelength_micron.inp` file. But for line transfer or for spectra based on the `camera_wavelength_micron.inp` they are not.

flux[i]: Flux in erg cm^{-2} s^{-1} Hz^{-1} at this wavelength as measured at a standard distance of 1 parsec (just as a way of normalization).

NOTE: Maybe in the future a new iformat version will be possible where more telescope information is given in the spectrum file.

A.15 OUTPUT: image.out or image_****.out

Any images that are produced by RADMC-3D will be written in a file called `image.out` or `image_<somename>.out`. Unformatted versions are also possible (`.out` being then `.uout`). The unformatted **[CHECK THIS: SHOULD THIS NOT BE "FORMATTED"?]** versions have the following structure:

```
iformat                     <=== For now this is 1
im_nx        im_ny
nlam
pixsize_x    pixsize_y
lambda[1]  ......... lambda[nlam+1]

image[ix=1,iy=1,img=1]
image[ix=2,iy=1,img=1]
  .
  .
image[ix=im_nx,iy=1,img=1]
image[ix=1,iy=2,img=1]
  .
  .
image[ix=im_nx,iy=2,img=1]
image[ix=1,iy=im_ny,img=1]
  .
  .
  .
image[ix=im_nx,iy=im_ny,img=nlam]

image[ix=1,iy=1,img=1]
  .
  .
  .
  .
image[ix=im_nx,iy=im_ny,img=nlam]
```

In most cases the nr of images (nr of wavelengths) is just 1, meaning only one image is written (i.e. the img=2, ... img=nlam are not there, only the img=1). The meaning of the various entries is:

iformat: This format number is currently set to 1.

im_nx, im_ny: The number of pixels in x and in y direction of the image.

nlam: The number of images at different wavelengths that are in this file. You can make a series of images at different wavelengths in one go, and write them in this file. The wavelength belonging to each of these images is listed below. The `nlam` can be any number from 1 to however large you want. Mostly one typically just makes an images at one wavelength, meaning `nlam`=1.

pixsize_x, pixsize_y: The size of the pixels in cm (!!). This means that the size is given in model units (distance within the 3-D model) and the user can, for any distance, convert this into arcseconds: pixel size in arcsec = (pixel size in cm / 1.496E13) / (distance in parsec). The pixel size is the full size from the left of the pixel to the right of the pixel (or from bottom to top).

lambda[i]: Wavelengths in micron belonging to the various images in this file. In case `nlam`=1 there will be here just a single number. Note that this set of wavelengths can be completely independent of all other wavelength grids.

image[ix,iy,img]: Intensity in the image at pixel `ix`, `iy` at wavelength `img` (of the above listed wavelength points) in units of erg cm^{-2} s^{-1} Hz^{-1} ster^{-1}. The pixels are ordered from left to right (in x) and from bottom to top (in y).

A.16 INPUT: (minor input files)

There is a number of lesser important input files, or input files that are only read under certain circumstances (for instance when certain command line options are given). Here they are described.

A.16.1 The `color_inus.inp` file (required with comm-line option 'loadcolor')

The file `color_inus.inp` will only be read by RADMC-3D if on the command line the option `loadcolor` or `color` is specified, and if the main action is `image`.

```
iformat                     <=== For now this is 1
nlam
ilam[1]
  .
  .
ilam[nlam]
```

iformat: This format number is currently set to 1.

nlam: Number of wavelength indices specified here.

ilam[i]: The wavelength index for image i (the wavelength index refers to the list of wavelengths in the `wavelength_micron.inp` file.

A.16.2 INPUT: `aperture_info.inp`

If you wish to make spectra with wavelength-dependent collecting area, i.e. aperture (see Section 9.3.2), then you must prepare the file `aperture_info.inp`. Here is its structure:

```
iformat                     <=== For now this is 1
nlam
lambda[1]      rcol_as[1]
  .              .
  .              .
lambda[nlam]   rcol_as[nlam]
```

with

iformat: This format number is currently set to 1.

nlam: Number of wavelength indices specified here. This does *not* have to be the same as the number of wavelength of a spectrum or the number of wavelengths specified in the file `wavelength_micron.inp`. It can be any number.

lambda[i]: Wavelength sampling point, in microns. You can use a course grid, as long as the range of wavelengths is large enough to encompass all wavelengths you may wish to include in spectra.

rcol_as[i]: The radius of the circular image mask used for the aperture model, in units of arcsec.

A.17 For developers: some details on the internal workings

There are several input files that can be quite large. Reading these files into RADMC-3D memory can take time, so it is important not to read files that are not required for the execution of the particular command at hand. For instance, if a model exists in which both dust and molecular lines are included, but RADMC-3D is called to merely make a continuum SED (which in RADMC-3D never includes the lines), then it would be a waste of time to let RADMC-3D read all the gas velocity and temperature data and level population data into memory if they are not used.

To avoid unnecessary reading of large files the reading of these files is usually organized in a 'read when required' way. Any subroutine in the code that relies on e.g. line data to be present in memory can simply call the routine `read_lines_all(action)` with argument `action` being 1, i.e.:

```
call read_lines_all(1)
```

This routine will check if the data are present: if no, it will read them, if yes, it will return without further action. This means that you can call `read_lines_all(1)` as often as you want: the line data will be read once, and only once. If you look through the code you will therefore find that many `read_***` routines are called abundantly, whenever the program wants to make sure that certain data is present. The advantage is then that the programmer does not have to have a grand strategy for when which data must be read in memory: he/she simply inserts a call to the read routines for all the data she/he needs at that particular point in the program, (always with action=1), and it will organize itself. If certain data is nowhere needed, they will not be read.

All these `read_***` routines with argument `action` can also be called with `action=2`. This will force the routine to (re-)read these data. But this is rarely needed.

Appendix B

More information about fortran-style unformatted data files

B.1 Overview

Some input and output files are so big that it is useful to try to write them as compact as possible. Unformatted I/O is much more compact than formatted I/O, as we shall explain in Section B.2. RADMC-3D offers unformatted input and output formats for some of the largest files. Here is a (presumably incomplete) list of files that have unformatted versions:

Name	formatted	unformatted double-prec.	unformatted single-prec.
`dust_density`	`.inp`	`.uinp`	
`dust_temperature`	`.inp`	`.uinp`	
`dust_temperature`	`.dat`	`.udat`	
`gas_density`	`.inp`	`.uinp`	
`gas_temperature`	`.inp`	`.uinp`	
`electron_numdens`	`.inp`	`.uinp`	
`ion_numdens`	`.inp`	`.uinp`	
`levelpop_***`	`.inp`	`.uinp`	`.usinp`
`numberdens_***`	`.inp`	`.uinp`	`.usinp`
`gas_velocity`	`.inp`	`.uinp`	`.usinp`
`microturbulence`	`.inp`	`.uinp`	`.usinp`
`stellarsrc_density`	`.inp`	`.uinp`	

The `.inp` or `.dat` ending means that the file is in ASCII format. If the ending is `.uinp` or `.udat` then the file is FORTRAN-style unformatted (see below) and the data are in double precision. If the ending is `.usinp` or `.usdat` then the file is FORTRAN-style unformatted (see below) and the data are in single precision. The latter (single precision) saves roughly a factor of 2 in file size, so that can be useful for big simulations.

The unformatted file styles are complicated. In fact, they are "unnecessarily" complicated as a result of FORTRAN's old-fashioned record-based I/O. That is why we devote an extra chapter to this file format, because it can be very confusing for the RADMC-3D-user.

B.2 Why is unformatted I/O more compact than formatted?

In formatted I/O each number is represented as an ASCII string, where each digit of the number is a separate byte. The integer number 10398 is thus 5 bytes, plus a separator (a space or a return), totalling 6 bytes. If we would write 1000 such numbers to a file in ASCII format, then we would obtain a file of 6000 bytes, roughly 6 kB. However, numbers between -16535 and 16536 are stored, in computer memory, as 2-byte integers. If we could write these exact integers to the file, then the 1000 numbers would occupy only 2000 bytes and the file would only be 2 kB

large. That is a factor of 3 smaller. Note: normally integers are at least 4 bytes long in computer memory. You can also declare 8-byte integers for very large integer numbers.

Another example: a floating point number 3.59932E+03 is " 3.59932E+03" as a string variable, occupying 13 bytes as a string, but only 4 bytes as a floating point number in memory. Writing this number in formatted form (as ASCII) requires thus 13 bytes per number, while if we could write it directly in unformatted form, it would require only 4 bytes. Note: a double precision float requires 8 bytes in unformatted form.

For files that tend to become extremely large it is therefore worthwhile to use unformatted I/O to reduce the size of the files by a factor of about two to four.

B.3 What is FORTRAN-style record-based unformatted I/O?

In most programming languages you can read and write unformatted data exactly as you think it should be: If you write a single double-precision number to a file, the file length would be exactly 8 bytes long. Unfortunately, FORTRAN is an exception. Fortran uses a record-based I/O style which originates still from the old days, which we will discuss below. When they drastically improved Fortran to Fortran-90 and Fortran-95, most problems of Fortran-77 (such as lack of pointers, lack of allocatable arrays) were solved. But they forgot to solve the problem with record-based I/O. This problem was only solved in the Fortran-2003 version, but this fortran version has not yet become full standard yet. For reasons of portability we do not want to move to Fortran-2003 until Fortran-2003 has become the defacto standard of Fortran on all platforms. Therefore we are stuck, for the moment, with old-style Fortran I/O, which is record-based.

So what is record-based unformatted I/O? The idea is that one reads or writes data in blocks, called records, from/to files. Each record starts with a four-bytes integer[1] that tells how long the record is (in units of bytes). Then the data of the record comes, and after the end of the data the same four-byte integer is written again. The next record then starts also with a four-byte integer, giving *its* length, followed by the data, and ending again with the four-byte integer. This file structure was useful in the days of tape devices, but is no longer useful today. Typically each time you write some data to a file, e.g.

```
write(1) a,b,c
```

where a, b and c are single-precision floats, Fortran writes twice two extra bytes. The file would thus have a length of 4+3*4+4=20 bytes. Try the following fortran-90 program:

```
program testwrite
  real :: a,b,c
  a=30487003.0
  b=45645.2
  c=-234.0
  open(unit=1,file='myfile',form='unformatted')
  write(1) a,b,c
  write(1) a,b,c
  close(1)
end program testwrite
```

This gives a file of 40 bytes long. A similar output with a C-program would give a 24 bytes long file.

Normally you do not need to worry about these records or these 4-byte headers and footers. You will not notice it at all under normal circumstances. For instance, the `myfile` file from the above program can be read with the following program:

```
program testread
  real :: a,b,c,d,e,f
  open(unit=1,file='myfile',form='unformatted')
  read(1) a,b,c
  read(1) d,e,f
  close(1)
  write(*,*) a,b,c,d,e,f
end program testread
```

Things go wrong, however, if you do not read according to records. For instance, if we would have done

[1] Four bytes for 64-bit computers and two bytes for 32-bit computers, as far as I know.

```
program testread
  real :: a,b,c,d,e,f
  open(unit=1,file='myfile',form='unformatted')
  read(1) a
  read(1) b
  read(1) c
  read(1) d
  read(1) e
  read(1) f
  close(1)
  write(*,*) a,b,c,d,e,f
end program testread
```

Then the program aborts due to end-of-file at the point where it tries to read 'c'. This is the curse of fortran record-based I/O.

B.4 Strategy of writing FORTRAN-unformatted files

To keep files small, we want to use unformatted I/O. But we have seen that Fortran uses records and this can make things complicated. The simplest solution, technically, would be to write each number as a separate record. In that way we never get into troubles as shown in the last example program of Section B.3. But it would make the file unnecessarily long: for each 4-byte float we would have 2x4=8 bytes extra: the record header and footer. This means that the file would become 3x longer than needed, and we could have just as well written ASCII.

Therefore: we want to make records relatively long, so that the addition of 2x4 bytes is, percentally, only a small addition. Example: If we pack a 1000 single-precision floats into a single record, then on-disk this leads to a file of length 1000x4+2x4=4008 bytes. This is almost as long as the 4000 bytes that C would have used. Therefore: the longer the record, the more compact the data storage.

Ideally you may want to make the record as long as the entire data you want to write or read. That is in principle possible. For 64-bit compilers the record length integer is 4-bytes, so this allows up to 4 GB of data to be stored in a single record. But for very large data files even this may be not enough. Also, as far as I know, there may still be compilers that use the 2-byte record header/footer. In that case the limit of the data in a single record is just 65536. For that reason we allow in RADMC-3D that the data is written in several records. The record length can be set by the user, and will be written as an integer somewhere near the start of the file (for details, see below).

B.5 General unformatted file structure used by RADMC-3D

All of the unformatted files listed in Section B.1 follow the same general structure.

- The very first record contains two 8-byte integers: The first is the format number (`iformat`), which is usually simply 1, but in later developments of RADMC-3D when certain file structures may change, could be upgraded to 2 or 3 or whatever. This number is just there for RADMC-3D to recognize old file structures (a backward compatibility feature). The second integer (`reclen`) is the general record length (in bytes) used for the main data of this file. See below for more information.
- The second record contains either one or two 8-byte integers. The first of these is (always) the number of cells of the model (i.e. the number of cells for which this file contains data). For some files there is this second integer. For instance, for `dust_density.uinp` this second integer tells the number of dust species.
- For `levelpop_**.uinp` (or `.usinp`) there is a next record containing a single 8-byte integer that gives the number of levels for which the populations are given in each cell. For other files no such record exists.
- Now the main data records follow.

The number of records that follow can be calculated as follows. Let `nfloats` be the number of floating point (either double or single) numbers per grid point for this file. Let `reclend` be the number of cells for which the data fits in a record (please make sure that this fits exactly!). For double precision one thus has `reclend` =

`reclen`/(`nfloats*8`) (relevant for files ending in `.uinp` or `.udat`). For single precision one thus has `reclend` = `reclen`/(`nfloats*4`) (relevant for files ending in `.usinp` or `.usdat`). For a double precision velocity field we have three double precision numbers per cell, so we get `reclend` = `reclen`/(`3*8`). Let `ncells` be the number of cells. Then the number of required records is then

```
nrecords = ( (ncells-1) / reclend ) + 1
```

where *integer division* is used, in which e.g. 5/4=1 while 3/4=0. The -1 and +1 are there to assure that we have enough space. For instance if we have 5 cells (`ncells=5`) and we have `reclend=4` (i.e. the data of 4 cells fit into a single record) then we need 2 records, the second of which will contain only one cell (the rest being padded with 0).

Here are a few things to keep in mind:

1. Be sure that `reclen` (the length of each data record in units of bytes) is exactly an integer number times the required data storage of each cell. Example: For the `dust_density.uinp` file each cell contains only a single number: the density of dust[2]. If we want to pack 1024 cells into a single record, then `reclen` must be exactly 8192.
2. It can happen that the last record is not fully filled. For instance, if we have 5 cells, but records of 4 cells each, then the file contains 2 data records, the first one filled with data of 4 cells, the second one only containing data of 1 cell. In the current version of RADMC-3D *you must still write the full record length: so simply pad the unused part with 0 or whatever.*
3. Because of the previous point (the padding) it is wise to do EITHER one of the following:
   - Make the record length exactly as big as needed to fit in all data in a single record.
   - Take the record length moderate so that if you have a nearly empty record at the end you won't waste too much space (but again, don't take it too small either, so that you don't waste too much space with the record headers and footers). If you take `reclend`=32 or so (meaning `reclen`=256 for a double precision scalar field or `reclen`=768 for a double precision vector field) then you are on the safe side.

I apologise for the complicated stuff here. As soon as fortran-2003 is fully the standard on all platforms and for the GNU fortran compilers, then I will simplify the unformatted I/O.

[2] Different dust species are written in the outermost loop, see Section A.3.

Appendix C

Command-line options

This chapter deals with all the possible command-line options one can give when calling the `radmc3d` code.

C.1 Main commands

In addition to the radmc3d.inp file, which contains many 'steering' parameters, one can (and even must) give RADMC-3D also command-line options. The most important (and compulsory) options are the 'command' what RADMC-3D should do. At the moment you can choose from:

mctherm: Runs RADMC-3D for computing the dust temperatures using the Monte Carlo method. See chapter 6.

spectrum: Runs RADMC-3D for making a spectrum based on certain settings. This option requires further command-line specifications. See chapter 9.

sed: Runs RADMC-3D for making a SED based on certain settings. This option requires further command-line specifications. Note that a SED is like a spectrum, but for continuum processes only (no lines). See chapter 9 for more details.

image: Runs RADMC-3D for making an image. This option requires further command-line specifications. See chapter 9.

movie: Like `image`, but now for a series of different vantage points. Useful for making movies in one go, without having to call RADMC-3D time and again. *NOTE: This command is still under development*. See chapter 9.

mcmono: (Only expect use). Runs RADMC-3D for computing the local radiation field at each location in the model. This is only useful for when you wish to couple RADMC-3D to models of chemistry or so, which need the local radiation field. See Section 6.4.

Example:

```
radmc3d mctherm
```

runs the RADMC-3D code for computing the dust temperatures everywhere using the Monte Carlo method.

There are also some additional commands that may be useful for diagnostics:

subbox_**: where **** is one of the following: `dust_density`, `dust_temperature`. But other quantities will follow in later versions. However, it may be better to use this option from within IDL. See Section 15.1.

linelist: Write a list of all the lines included in this model.

C.2 Additional arguments: general

Here is a list of command line options, on top of the above listed main commands (Note: We'll try to be complete, but as the code develops we may forget to list new options here):

npix: [for images] The next number specifies the number of pixels in both x and y direction, assuming a square image.

npixx: [for images] The next number specifies the number of pixels in x direction only.

npixy: [for images] The next number specifies the number of pixels in y direction only.

nrrefine: [for images and spectra] Specifies a maximum depth of refinement of the pixels (see Section 9.6).

fluxcons: [for images and spectra] Puts nrrefine (see above) to a large value to assue flux conservation (see Section 9.6).

norefine: [for images and spectra] Puts nrrefine (see above) to 0 so that each pixel of the image corresponds only to 1 ray. This is fast but not reliable and therefore not recommended (see Section 9.6).

nofluxcons: [for images and spectra] As `norefine` above.

noscat: This option makes RADMC-3D ignore the dust scattering process (though not the scattering extinction!) in the images, spectra and Monte Carlo simulations. For images and spectra this means that no scattering Monte Carlo run has to be performed before each image ray tracing (see Section 6.5.4). This can speed up the making of images or spectra enormously. This is even more so if you make images/spectra of gas lines with LTE, LVG or ESCP methods, because if no scattering Monte Carlo needs to be made, ray-tracing can be done multi-frequency for each ray, and the populations can be calculated once in each cell, and used for all frequencies. That can speed up the line rendering enormously – of course at the cost of not including dust scattering. For lines in the infrared and submillimeter, if no large grains are present, this is usually OK, because small grains (smaller than about 1 micron) have very low scattering albedos in the infrared and submillimeter.

ilambda or **inu**: [for images] Specify the index of the wavelength from the `wavelength_micron.inp` file for which a ray-trace image should be made.

color: [for images] Allows you to make multiple images (each at a different wavelength) in one go. This will make RADMC-3D read the file `color_inus.inp` (see Section A.16) which is a list of indices `i` referring to the `wavelength_micron.inp` file for which the images should be made. See Section 9.4 for details.

loadcolor: [for images] Same as `color`.

loadlambda: [for images] Allows you to make multiple images (each at a different wavelength) in one go. This will make RADMC-3D read the file `camera_wavelength_micron.inp` or `camera_frequency.inp` (whichever is present) to read the precise wavelength points at which you wish to make the images. In contrast to `loadcolor`, which only allows you to pick from the global set of wavelength used by the Monte Carlo simulation (in the file `wavelength_micron.inp` or `frequency.inp`), with the `camera_wavelength_micron.inp` or `camera_frequency.inp` files you can specify any wavelength you want, and any number of them. See Section 9.4 for details.

sizeau: [for images and spectra] The next number specifies the image size in model space in units of AU (=1.496E13 cm). This image size is measured from the image center to the left or right or top or bottom. This gives always square images. This image half size in au is observer distance independent. The corresponding image half size in arcsec is: image half size in arcsec = image half size in AU / (distance in parsec).

sizepc: [for images and spectra] Same as `sizeau`, but now in parsec units.

zoomau: [for images and spectra] The next four numbers set the image window precisely by specifying the xleft, xright, ybottom, ytop of the image in units of AU. The zero point of the image (the direction of the 2-D image point located at (0.0,0.0) in image coordinates) stays the same (i.e. it aims toward the 3-D point in model space given by `pointau` or `pointpc`). In this way you can move the image window left or with or up or down without having to change the `pointau` or `pointpc` 3-D locations. Also for local perspective images it is different if you move the image window in the image plane, or if you actually change the direction in which you are looking (for images from infinity this is the same). *Note that if you use this option without the* `truepix` *option RADMC-3D will always make square pixels by adapting* `npixx` *or* `npixy` *such that together with the* `zoomau` *image size you get approximately square pixels. Furthermore, if* `truezoom` *is not set, RADMC-3D will alleviate the remaining tiny deviation from square pixel shape by slightly (!) adapting the* `zoomau` *window to obtain exactly square pixels.*

zoompc: [for images and spectra] Same as `zoomau`, but now the four numbers are given in units of parsec.

truepix: [for images and spectra] If with `zoomau` or `zoompc` the image window is not square then when specifying `npix` one gets non-square pixels. Without the `truepix` option RADMC-3D will adapt the `npixx` or `npixy` number, and subsequently modify the zoom window a bit such that the pixels are square. With the `truepix` option RADMC-3D will not change `npixx` nor `npixy` and will allow non-square pixels to form.

truezoom: [for images and spectra] If set, RADMC-3D will always assure that the exact zoom window (specified with `zoomau` or `zoompc`) will be used, i.e. if `truepix` is *not* set but `truezoom` is set, RADMC-3D will only (!) adapt `npixx` or `npixy` to get *approximately* square pixels.

pointau: [for images and spectra] The subsequent three numbers specify a 3-D location in model space toward which the camera is pointing for images and spectra. The (0,0) coordinate in the image plane corresponds by definition to a ray going right through this 3-D point.

pointpc: [for images and spectra] Same as `pointau` but now in units of parsec.

incl: [for images and spectra] For the case when the camera is at infinity (i.e. at a large distance so that no local perspective has to be taken into account) this inclination specifies the direction toward which the camera for images and spectra is positioned. Incl = 0 means toward the positive z-axis (in cartesian space), incl=90 means toward a position in the x-y-plane and incl=180 means toward the negative z-axis. The angle is given in degrees.

phi: [for images and spectra] Like `incl`, but now the remaining angle, also given in degrees. Examples: `incl`=90 and `phi`=0 means that the observer is located at infinity toward the negative y axis; `incl`=90 and `phi`=90 means that the observer is located at infinity toward the negative x axis; `incl`=90 and `phi`=180 means that the observer is located at infinity toward the positive y axis (looking back in negative y direction). Rotation of the observer around the object around the z-axis goes in clockwise direction. The starting point of this rotation is such that for `incl`=0 and `phi=0` the (x, y) in the image plane correspond to the (x, y) in the 3-D space, with x pointing toward the right and y pointing upward. Examples:] if we fix the position of the observer at for instance `incl`=0 (i.e. we look at the object from the top from the positive z-axis at infinity downward), then increasing `phi` means rotating the object counter-clockwise in the image plane.

posang: [for images] This rotates the camera itself around the $(0, 0)$ point in the image plane.

imageunform: Write out images in unformatted form

imageformatted: Write out images in text form (default)

circ: When spectra or SEDs are made, and when spherical coordinates are used, then this option will make RADMC-3D use a circular arrangement of pixels. This is an emulation of RADMC (the predecessor code), and has some advantages in terms of speed over RADMC-3D sub-pixeling method. See Section 9.9.

tracetau: [for images] If this option is set, then instead of ray-tracing a true image, the camera will compute the optical depth at the wavelength given by e.g. `inu` and puts this into an image output as if it were a true image. Can be useful for analysis of models.

tracecolumn: [for images] Like `tracetau` but instead of the optical depth the simple column depth is computed in g/cm^2. *NOTE: for now only the column depth of the dust.*

tracenormal: [for images: Default] Only if you specified `tracetau` or `tracecolumn` before, and you are in child mode, you may sometimes want to reset to normal imaging mode.

apert or **useapert**: [for images/spectra] Use the image-plane aperture information from the file `aperture_info.inp`.

noapert: [for images/spectra] Do *not* use an image-plane aperture.

nphot_therm: [for MC] The nr of photons for the thermal Monte Carlo simulation. But it is better to use the `radmc3d.inp` for this (see Section A.1), because then you can see afterward with which photon statistics the run was done.

nphot_scat: [for MC] The nr of photons for the scattering Monte Carlo simulation done before each image (and thus also in the spectrum). But it is better to use the `radmc3d.inp` for this (see Section A.1), because then you can see afterward with which photon statistics the run was done.

nphot_mcmono: [for MC] The nr of photons for the monochromatic Monte Carlo simulation. But it is better to use the `radmc3d.inp` for this (see Section A.1), because then you can see afterward with which photon statistics the run was done.

C.3 Switching on/off of radiation processes

You can switch certain radiative processes on or off with the following commands (though often the `radmc3d.inp` file also allows this):

inclstar: [for images and spectra] Include stars in spectrum or images.

nostar: [for images and spectra] Do *not* include stars in spectrum or images. Only the circumstellar / interstellar material is imaged as if a perfect coronograph is used.

inclline: Include line emission and extinction in the ray tracing (for images and spectra).

noline: Do not include line emission and extinction in the ray tracing (for images and spectra).

incldust: Include dust emission, extinction and (unless it is switched off) dust scattering in ray tracing (for images and spectra).

nodust: Do not include dust emission, extinction and scattering in ray tracing (for images and spectra).

inclfreefree: Include the gas continuum free-free emission (Bremsstrahlung). See chapter 8.

nofreefree: Do not include the gas continuum free-free emission.

inclgascont: Include all gas continuum processes known by RADMC-3D (right now this is only free-free, as of 04.07.2010, but this could become more in later versions).

nogascont: Do not include the gas continuum.

C.4 Commands for child mode

Here is a list of options that are only useful for when you use RADMC-3D in child mode (Chapter 12):

child: This prevents RADMC-3D from exiting after each main command is done. Instead, RADMC-3D will wait for further commands being given on the RADMC-3D internal command line. This can be useful if multiple actions are to be taken, and the user does not want to wait for long file input reading. It is in fact used by the `viewimage.pro` GUI (see chapter 14) for making images. Note that with RADMC-3D command line each command has to be on a separate line (i.e. ending with a return).

exit or **quit**: In `child` mode you can finish the command-line mode by entering exit or quit.

enter: In `child` mode enter says RADMC-3D that it can start executing the last set of commands. After it is done a new set of commands (each on a new line) can be given, again ending with the word enter (on a separate line).

writeimage: In `child` mode an image can be written to the standard output (in ascii form) with this command.

writespec: In `child` mode a spectrum can be written to the standard output (in ascii form) with this command.

Appendix D

Which options are mutually incompatible?

For algorithmic reasons not all options / coordinate systems and all grids are compatible with each other. Here is an overview of which options/methods work when. Note that only options/methods for which this is a possible issue are listed.

D.1 Coordinate systems

Some coordinate systems exclude certain possibilities. Here is a list.

Option/Method:	Cart 3
Second order ray-tracing (Sec. 9.8)	yes
Isotropic scattering	yes
An-isotropic scattering for thermal Monte Carlo	yes
An-isotropic scattering for monochromatic Monte Carlo	yes
An-isotropic scattering for images and spectra	yes
Gas lines	yes
Gas lines and Doppler-shift line catching	yes
Circular images (backward compatibility with RADMC)	no

D.2 Scattering off dust grains

The inclusion of the effect of scattering off dust grains in images and spectra typically requires a separate Monte Carlo computation for each image. This is done automatically by RADMC-3D. But it means that there are some technical limitations.

Option/Method:	No scattering	Isotropic approximation	Full anisotropic scattering
Fast multi-frequency ray tracing for spectra (auto)	yes	no	no
Multiple images at different vantage point at once (Sec. 9.11)	yes	yes	yes
Local observer (Sec. 9.10)	yes	yes	no

Whereever there is "(auto)" this means that the user does not need to set/choose anything: RADMC-3D will automatically make the choice correctly. It is listed here just to make clear to the user why things may work differently under different circumstances.

D.3 Local observer mode

The local observer mode (Sect. 9.10) is a special mode for putting the observer in the near-field of the object, or even right in the middle of the object. It is not meant to be really for science use (though it can be used for it, to a certain extent), but instead for public outreach stuff. However, it is kept relatively basic, because to make this mode compatible with all the functions of RADMC-3D would require much more development and that is not worth it at the moment. So here are the restrictions:

Option/Method:	Local observer mode
Dust isotropic scattering	yes
Dust an-isotropic scattering	no
Multiple images at different vantage point at once (Sec. 9.11)	yes
Second-order ray-tracing (Sec. 9.8)	yes
Doppler-catching of lines (Sec. 7.6)	no