DISKLAB
A protoplanetary disk modeling package in Python

Cornelis Dullemond & Til Birnstiel

(2018, 2019)

Contributions by: Carsten Dominik, Andrej Hermann

very minor changes by Oliver Porth, 2022

Contents

Chapter 1

Preliminaries

1.1 Purpose and philosophy

The DISKLAB code package is *not* meant as a blackbox model set. It is not supposed to be the kind of model behind a widget interface, where one merely has to change a few parameters.

Instead it is meant as a *library* of subroutines / methods to very easily program your own axisymmetric disk models and experiments in Python. That is also the idea behind the name: a laboratory in which you have all the tools to perform your experiments, without having to start from scratch to implement all the standard stuff.

The package includes (among other things);

- A set of standard basic models, which you can then modify at will.
- Subroutines to compute standard stuff from your disk model (e.g. disk mass, Toomre Q value, temperature, pressure scale height etc).
- Subroutines to compute standard stuff for the dust (e.g. stopping time and Stokes number for given grain size or vice versa, dust layer vertical thickness, optical depth, simple formal radiative transfer).
- A method for time-dependent evolution of the viscous disk equation.
- A method for time-dependent evolution of the dust radial drift and radial mixing.
- Subroutines to calculate stability criteria.

There are numerous examples in this tutorial (the *code snippets*). These are not supposed to be physically fully realistic model setups: they are meant as simple starting points from where you, the user, can pick up and do your own experiments. They are merely meant to demonstrate how to use the code. **We do not take responsibility for any science produced with this package: that responsibility lies solely with the user of this package.**

1.2 Installing

The installation is kept as simple as possible. You need to have an up-to-date Python version installed with all the usual libraries such as `numpy` and `scipy`. Both Python 2.7 and Python 3 should work. On several systems the Anaconda Python installation is recommended, since it includes all you need.

We assume here that you execute Python scripts directly from the command-line (i.e. a Linux-like bash shell or tcsh shell). While DISKLAB can be used from within the Jupyter environment, and most (all?) of the stuff done here should be Jupyter-compatible, the tutorial is written under the assumption that you execute everything from the terminal. Users who are unfamiliar with Linux/Unix-like environments (which includes Max OS X) may want to read up a little on concepts such as: terminals, bash shells and environment variables, i.e. the basic Unix concepts.

Also, if you are unfamiliar with it, you may want to get up to date with the concept of text editors (as opposed to Word or the like), which allow you to edit text in a computer-readable (i.e. primitive, non-marked-up) way.

There are two main ways to install DISKLAB: the automated way using `pip install`, and the primitive way by hand (by setting the appropriate environment variables).

1.2.1 Installation using pip (preferred)

In principle it should be possible to install DISKLAB in an automatic way. Typically you would install it by executing this in the top-level directory of this repository:

```
pip install -e .
```

This installs a development version, i.e. if you modify the code, those changes will be reflected if you `import disklab`. To install a static version of the current code, leave out the `-e` option.

What this will do is: (1) It will let Python know that this new library exists and where it can be found, so that you can simply include it in your Python code by typing `import disklab`, (2) it will use `f2py` to compile some fortran code parts into Python libraries, so that these can be directly used from within Python.

All examples of how to use DISKLAB, including those from this tutorial, can be found in the `snippets/` directory.

If you forgot (or never knew) where the DISKLAB main directory is, but you know that DISKLAB is succesfully installed, then you can find its path using from within Python:

```
import disklab
dir =disklab.utilities.get_disklab_main_directory()
```

The `dir` is then a string which contains the path.

1.2.2 Installation "by hand" (the primitive way)

If the above `pip install` method does not work, you can also install DISKLAB the primitive way. Here is how it works. The Python executables of the DISKLAB package are all in the `disklab/` directory. To be able to use them from any directory on your computer system you need to tell Python where to search. This is done (on Linux/Unix-like systems) with the environment variable `PYTHONPATH`. This should point to the top-most directory of the DISKLAB package. Let us call this top-most directory for convenience `$HOME/disklab_py/`, but of course this should be the actual directory where you are now installing this package. Just to be sure you and I are "on the same page": if you type (in a linux shell)

```
ls $HOME/disklab_py/disklab/*.py
```

you should see something like

```
disklab/__init__.py        disklab/natconst.py
disklab/diskradial.py       disklab/solvediffonedee.py
disklab/grainmodel.py       disklab/tridag.py
```

which are the executables of the DISKLAB package.

To set the `PYTHONPATH` environment variable for the current `bash` shell you type (on the command-line):

```
export PYTHONPATH='$HOME/disklab_py':$PYTHONPATH
```

To check if you have been successfull, you can type:

```
echo $PYTHONPATH
```

This prints the content of this environment variable. Dependent on what you already had in the `PYTHONPATH` variable before, it could something like this:

```
/Users/cornelisdullemond/science/pyprog/disklab_py:/Users/cornelisdullemond/bin/python
```

where the part before : is the new thing we added, while the part after : is the PYTHONPATH value before we added the new path.

One problem is that every time we open a new terminal, the PYTHONPATH is again reset to the old value. To *permanently add* the DISKLAB model to the PYTHONPATH, you have to add the above line (the one starting with export) to the .bashrc file in your home directory, i.e. to the $HOME/.bashrc file. The best way to do this is to use your favorite text editor and edit this file, go to the very end of the file, and add the above line.

From that point onward, every time you open a *new* shell (terminal window) your PYTHONPATH variable should contain the path to the $HOME/disklab_py/ directory. You can verify this by typing echo $PYTHONPATH, as above. If it is not there, then presumably you still are in an 'old' shell (from before you edited the .bashrc file). Just close the shell (type exit) and open a new one.

1.2.3 Your first steps...

With a bit of luck you are now all set to start using DISKLAB.

We now recommend you to make a model directory in which you will do your experiments. For instance:

```
cd $HOME/disklab_py
mkdir mydiskmodels
cd mydiskmodels
```

Here you can write the Python scripts for your models and create the figures you want. This will then not accidently interfere or change the main code of the package because it will be nicely isolated. You can now try out if the installation was successfull (if not, try to open a new shell and try again). Type

```
python
```

in the shell, and you should see some text followed by something like >>>. Type:

```
import disklab
```

This should do ... nothing (or so it seems). It simply returns another >>>. If so, then you have succesfully installed DISKLAB. You can leave Python again by pressing control-D. If, however, it complains that it cannot find the disklab module, then Python has not managed to learn where the DISKLAB code is. Something of the above stuff with the environment variables must have gone wrong. Try closing and opening a new shell. If that still does not work, ask a Linux-expert.

In case the above was not successful: Another way to let Python know where the DISKLAB package is located is to start every Python session with the following commands:

```
import sys
sys.path.append("<directory path to disklab_py>")
```

Where you should replace the stuff between < and > with the directory of DISKLAB (which we referred to above as $HOME/disklab_py). This method works, but it is cumbersome, since you have to always copy-paste this in at the start of each Python session.

For Jupyter Notebooks, by the way, instead of PYTHONPATH you should use JUPYTER_PATH. But if that somehow does not work, a quick-fix could be to start each Jupyter Notebook with the above sys.path.append() trick.

If you still have problems: There is an even simpler way to use DISKLAB, but that is rather clunky: simply copy everything you need into your mydiskmodels directory:

```
cp src/*.py mydiskmodels/
cp snippets/*.py mydiskmodels/
```

This is not the recommended way, but it works. The main thing to keep in mind is that if you use any of the code examples and/or snippets, you must remove the "`disklab.`" in the lines that say `from disklab.diskradial import *` and similar lines. This is because if you simply copy all python codes into your current working directory (the `mydiskmodels` directory), then the DISKLAB codes are stand-alone, and no longer part of a "package" called `disklab`.

1.3 Executing the example code snippets

In this tutorial there are many code snippets that you can simply cut & paste into python. The best way to do this is to start `ipython` (instead of the standard `python`) with the extension `---matplotlib`:

```
ipython --matplotlib
```

That should get you a prompt looking like:

```
In [1]:
```

You can cut & paste the code snippets of the tutorial in there. However, if you want to keep your work, it is better to paste the snippets into a file, for example into `mytest.py`, and then run it from within python:

```
%run mytest.py
```

Some of the longer code snippets are already available as files in the directory `snippets/`, see Section 1.4.

1.4 The code snippets in the `snippets/` directory

The code snippets in the `snippets/` directory are meant as starting points for models that *you* wish to develop. They are templates, so to speak. They demonstrate what DISKLAB can do. If you want to make your own model, typically you copy one of the snippets to your own model directory and start developing from there. You can also copy bits and pieces from different snippets.

All the snippets in the `snippets/` directory start with an `import` statement from a file called `snippet_header.py`. This is just a convenient way to import all the modules needed for this snippet. For instance,

```
from snippet_header import DiskRadialModel, np, plt, MS, year, au
```

is just a shorthand for

```
import numpy as np
import matplotlib.pyplot as plt
from disklab import DiskRadialModel
from disklab.natconst import MS, year, au
```

where the first two lines should be familiar to all Python users, and the third line imports the `DiskRadialModel` class from the DISKLAB package, and the fourth line imports the values of the solar mass (in gram), the year (in seconds) and the astronomical unit (in cm).

All snippets from the `snippets/` directory end with `finalize()`. This is just a convenience for us (the authors) to automatically re-create all plots. For you (the user) you *can* also replace `finalize()` with `plt.show()`, if you like.

Note: To run the code snippets of the `snippets/` directory you would have to first enter that directory before you start Python. For example:

```
cd snippets/
ipython --matplotlib
%run snippet_dustdrift_1.py
```

Heads up:
The snippets in the `snippets/` directory, and the figures they produce, are directly imported into this latex tutorial.

So if you want to experiment with the snippets by *modifying* them, we *strongly recommend* that you copy them to another directory first:

```
mkdir mysnippets
cp snippets/snippet*.py mysnippets/
cd mysnippets
< now you can modify the snippets at will >
ipython --matplotlib
...
```

Then you can modify at will without running the risk of overwriting the original snippets (and thus without the risk of accidently modifying your copy of this tutorial).

1.5 If you want to modify the DISKLAB code safely

Most of the experimentation should be doable without modifying the actual DISKLAB package. But given that the main idea behind DISKLAB is to create as much experimentation-flexibility as possible for the user, it might sometimes be desirable to modify some piece of the DISKLAB code.

It is recommended *never* to change directly the code in the `disklab/` directory (except for updates). Instead you can copy a version to a subdirectory in your local working directory and rename it:

```
cd mydiskmodels
mkdir mydisklab
cp ../disklab/*.py mydisklab/
< now change whatever you want in the mydisklab/ directory >
ipython --matplotlib
from mydisklab.diskradial import *
```

So instead of importing `disklab` you now import the local `mydisklab`, which is your own private (and modified) copy of the DISKLAB package. The nice thing is that your modifications will then be only effective in that specific working directory (here: `mydiskmodels`). So you can safely change the DISKLAB codes without changing the main package.

1.6 Default units are CGS - how to switch to SI

The `natconsts/` directory contains two files:

```
natconst_CGS.py
natconst_SI.py
```

These are simple list of natural constants in CGS and SI units respetively. The `disklab` directory contains the `natconst.py` file, which is the natural constants file it uses for all its internal calculations.

If you want to **permanently** change to SI, you can simply copy the `natconst_SI.py` file to `disklab/natconst.py`. Although in the code all comments are assuming CGS, the code gets all its unit-dependent constants from the `disklab/natconst.py` file, so just replacing that file with the SI version should do the trick.

However, a **safer** way of doing this would be to make the local modified version of DISKLAB, as explained in Section 1.5. This is how it would work:

```
cd mydiskmodels
mkdir mydisklab
cp ../disklab/*.py mydisklab/
cp ../natconsts/natconst_SI.py mydisklab/natconst.py
ipython --matplotlib
```

```
from mydisklab.diskradial import *
from mydisklab.natconst import *
d=DiskRadialModel()
d.r.rmin()/au
```

This should give the value 0.1, because the default inner radius of the disk is 0.1 au. You can verify that the astronomical unit is now in meters instead of cm.

1.7 Naming conventions

We follow the standard Python naming conventions. That means that classes (such as the `DiskRadialModel` class) are written in CapsWord convention, while modules (such as the `diskradial` module) are written with small letters. For example, the file `diskradial.py` contains the definition of the classes `DiskRadialModel` and `DiskRadialComponent`.

Part I

1-D Radial Disk Models

Chapter 2

Introduction: Disk Radial Model Python Class

The centerpiece of the DISKLAB package is the `diskradial.py` module. It defines the `DiskRadialModel` Python class on which everything is based. The basic `DiskRadialModel` class is meant as a simple and quick approximate disk model that you can use to analyze e.g. observational data or you can use as a backdrop for other calculations. It is not meant as a detailed and accurate disk model. Many things are very approximate, such as the use of a fixed irradiation angle and such. Also, the model is in continuous development. This document is a description of the model, but cannot be guaranteed to be always up to date.

The model is in CGS units. A file `natconst.py` is provided which contains shortcuts for many of the standard natural constants. Note that one may want to be careful not to accidently overwrite those (if you import natconst.py using `from disklab.natconst import *`), or import it as `import disklab.natconst as nc`. Internally in the DISKLAB package the natural constants are used as `import disklab.natconst as nc`, but the `from disklab.natconst import *` can be convenient for the user so that natural constants such as an astronomical unit are simply `au` instead of `nc.au`. In the code snippets we will always use the `from disklab.natconst import *` method for convenience.

Many of the methods in this class have examples given in their documentation string.

2.1 Set up the model

The model is set up in Python by:

```
from disklab.diskradial import *
d = DiskRadialModel()
```

This sets up the standard model with default values. But you can change the default values of stellar parameters, as well as the radial grid of the disk model:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mstar=2*MS,lstar=10*LS,rin=0.1*au,rout=1000*au,nr=300)
```

where `MS` and `LS` are the solar mass and luminosity, respectively, and `au` is an astronomical unit. If you set the value of `mdisk` you automatically choose the standard powerlaw disk model (see below), for example:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mstar=2*MS,lstar=10*LS,rin=0.1*au, \
              rout=1000*au,nr=300,mdisk=0.01*MS,plsig=-1)
```

The `d` object now contains already a full 1-D disk model (no vertical structure, though). The key contents are the following variables

```
d.alpha     # Viscous alpha parameter (can be scalar or array)
d.cs        # Isothermal sound speed [cm/s]
d.flang     # Angle of incidence of stellar radiation (flaring angle)
d.hp        # Pressure scale height of the disk [cm]
d.lstar     # Luminosity of the star [erg/s]
d.mass      # Mass of the disk [g]
d.mstar     # Mass of the star [g]
d.omk       # Array of the Kepler frequency [1/s]
d.plsig     # Powerlaw index of gas surface density (only for information)
d.r         # Array of the radial grid coordinate [cm]
d.rhomid    # Array of the midplane gas density [g/cm^3]
d.rstar     # Radius of the star [cm]
d.Sc        # Schmidt number (only important for turbulent mixing)
d.sigma     # Array of gas surface density [g/cm^2]
d.sigmin    # Lower bound to gas surface density [g/cm^2]
d.tmid      # Midplane disk temperature [K]
d.tstar     # Stellar effective temperature [K]
d.mu        # Mean molecular mass [mp]
```

Note that these are only the variables after the above simple commands. If you add dust or do other stuff (see further down the tutorial), new variables and arrays will be added. So the above list is not a complete list.

But often you can also make a disk model in two steps:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mstar=2*MS,lstar=10*LS,rin=0.1*au,rout=1000*au,nr=300)
d.plsig = -1
d.make_disk_from_m_pl(mdisk=0.01*MS)
```

which is essentially the same as above, but here we explicitly used one of the build-in standard disk model setups described in Chapter 3. By default, the initialization method of `DiskRadialModel` calls the simplest model setup of Chapter 3, namely `make_disk_from_m_pl()`, described in Section 3.1.2. But you can choose, of course, any of the built-in models of Chapter 3.

Note that by making such a disk model, only the *radial* disk structure is set up, i.e. the surface density profile. The vertical structure is not set up with any of the `d.make_***` methods (for the vertical structure, see part II of this tutorial). There is a series of standard disk models: check out the `d.make_***` methods, and their corresponding sections below.

You can plot the results and print some diagnostics with e.g.:

```
from disklab.diskradial import *
import matplotlib.pyplot as plt
from disklab.natconst import *
d = DiskRadialModel(rout=500*au)
d.make_disk_from_lbp_alpha(1e-2*MS,1*au,0.001,1e6*year)
plt.plot(d.r/au,d.sigma)
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$\Sigma [\mathrm{g}\,\mathrm{cm}^{-2}]$')
plt.xscale('log')
plt.yscale('log')
plt.ylim(1e-5,1e4)
plt.show()
print('Mdisk = {} Msun'.format(d.mass/MS))
```

Note that the model also computes the *midplane* volume density given by `rhomid`. This is *not* the 2-D or 3-D gas density, but only the midplane one.

2.2 Computing the midplane temperature

The method `compute_disktmid()` is used to compute the midplane temperature of the disk. This method is *automatically called* by the `__init__()` method of this class. The temperature is computed using a simple irradiation recipe. The irradiation heating rate is:

$$Q_{\rm irr}(r) = 2\varphi(r)\frac{L_*}{4\pi r^2} \tag{2.1}$$

where $\varphi(r)$ is the irradiation angle (`flang`), which can either be given as a global value φ or as an array $\varphi(r)$. The factor 2 is due to the fact that the disk has two sides. The cooling rate of a disk with effective temperature $T_{\rm eff}$ at the surface is:

$$Q_{\rm cool}(r) = 2\sigma_{\rm SB}T_{\rm eff}(r)^4 \tag{2.2}$$

where $\sigma_{\rm SB}$ is the Stefan-Boltzmann constant. For irradiation the midplane temperature will be equal to $T_{\rm mid} = T_{\rm eff}/2^{0.25}$, so by setting $Q_{\rm cool}(r) = Q_{\rm irr}(r)$ we obtain:

$$\sigma_{\rm SB}T_{\rm mid}(r)^4 = \frac{1}{4}Q_{\rm irr}(r) = \frac{1}{2}\varphi(r)\frac{L_*}{4\pi r^2} \tag{2.3}$$

The factor of $1/2^{0.25}$ comes in because the primary stellar radiation is first absorbed by the surface layer: One half is emitted away from the disk, the other half down into the disk. Only the latter is heating up the midplane. Note that this temperature recipe is independent of $\Sigma(r)$. See Chiang & Goldreich (1997) for details. The resulting midplane temperature is stored inside the object as `tmid`. Automatically also computed is the isothermal sound speed

$$c_s = \sqrt{\frac{k_{\rm B}T}{\mu m_p}} \tag{2.4}$$

stored as `cs`, with $\mu = 2.3$ and m_p the proton mass and $k_{\rm B}$ the Boltzmann constant. And the vertical pressure scale height of the disk

$$H_p = \frac{c_s}{\Omega_K} \tag{2.5}$$

is also computed, and stored as `hp`, where the Kepler frequency is calculated as:

$$\Omega_K(r) = \sqrt{\frac{GM_*}{r^3}} \tag{2.6}$$

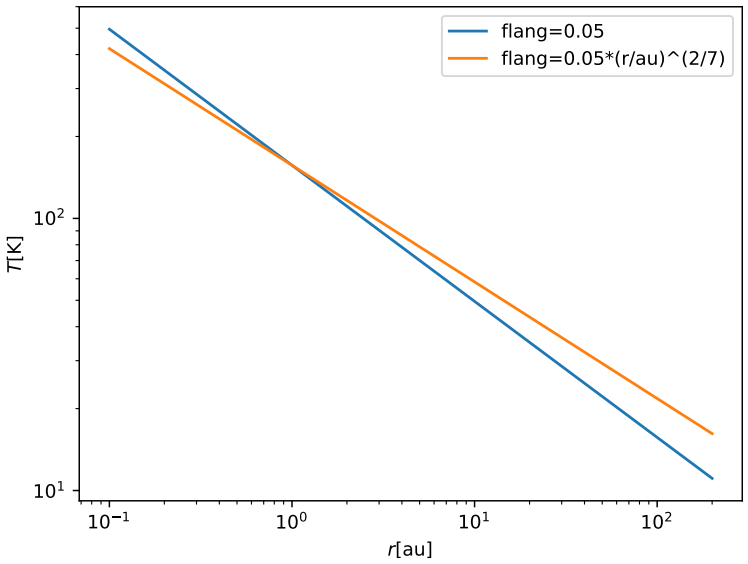
which is stored in the array `omk`.

Example: a simple powerlaw disk with a flaring (i.e. irradiation) angle varying with radius. The code snippet is `snippet_simple_rt_1.py`. In Python run it as:

```
%run snippet_simple_rt_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, plt, au, finalize

d1 = DiskRadialModel(flang=0.05)
flang = 0.05 * (d1.r / au)**(2. / 7.)
d2 = DiskRadialModel(flang=flang)
plt.plot(d1.r / au, d1.tmid, label='flang=0.05')
plt.plot(d2.r / au, d2.tmid, label='flang=0.05*(r/au)^(2/7)')
plt.legend()
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$T [\mathrm{K}]$')
plt.xscale('log')
plt.yscale('log')

finalize(results=(d1.tmid,d2.tmid))
```



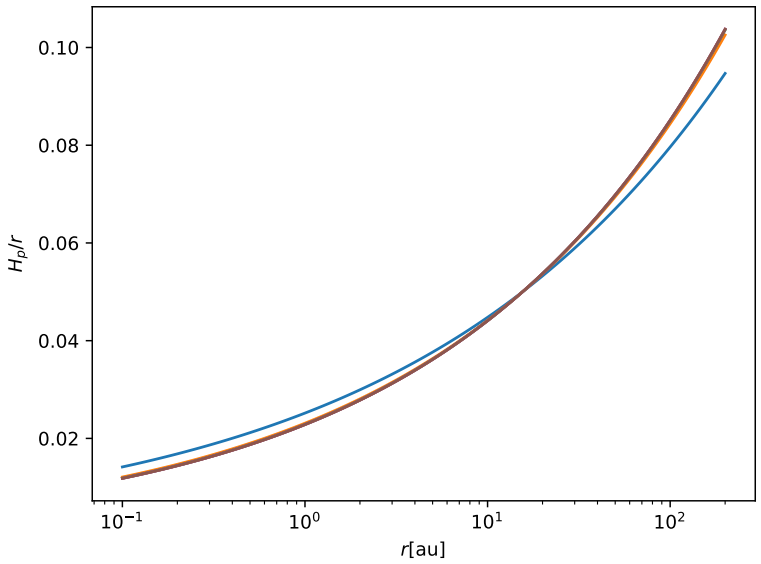
To explain what happens here: We make disk models without specifying the surface density, but we specify the flaring angle $\varphi = 0.05$ for the first model. For the second model we specify a radius-dependent flaring angle $\varphi(r) = 0.05\,(r/\mathrm{au})^{2/7}$. The initialization of the object automatically calls the `compute_disktmid()` method. In neither of the two cases the surface density has to be specified.

A slightly more sophisticated example: Instead of setting the flaring angle directly, we set the *flaring index*, which is the ratio of φ to H_p/r. This requires iteration, which quickly converges. The code snippet is `snippet_iterate_hp_1.py`. In Python run it as:

```
%run snippet_iterate_hp_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, plt, au, finalize

flindex = 1.0   # Flaring index defined as Flaring angle = flindex * H_p/r
d = DiskRadialModel()
plt.plot(d.r / au, d.hp / d.r)
for iter in range(5):
    d.flang = flindex * d.hp / d.r
    d.compute_disktmid()
    plt.plot(d.r / au, d.hp / d.r)
plt.xscale('log')
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$H_p/r$')

finalize(results=(d.hp))
```



A self-consistent model, however, requires the self-consistent computation of the surface height H_s (see Chiang & Goldreich 1997; Chiang et al. 2001; and Dullemond & Dominik 2001). Then, however, the surface density of the gas (or more importantly: that of the dust) as well as the dust opacities are required. See Subsection 4.1 where this is worked out.

So far we have not included the viscous heating into the computation of the midplane temperature. We refer this to Subsection 4.2.

2.3 Computing the disk mass

The method `compute_mass()` computes the disk mass in gram, if the disk is already installed.

$$M_{\rm disk} = 2\pi \int_{r_{\rm in}}^{r_{\rm out}} r\Sigma_g(r) dr \tag{2.7}$$

where the integral is computed as a numerical sum. This value is stored as `mass`. This mass includes only the gas. The disk mass is automatically computed when one of the standard disk models of Chapter 3 are set up. But sometimes one may want to recompute the disk mass. Example:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(rout=500*au,mdisk=0.01*MS)
factor=np.exp(-d.r/(200*AU))
d.sigma *= factor
d.compute_mass()
print('New disk mass = {} Msun'.format(d.mass/MS))
```

2.4 Computing the Toomre Q value

The method `compute_qtoomre()` compute the Toomre Q value for the gas:

$$Q_g(r) = \frac{c_s \Omega_K}{\pi G \Sigma_g} \tag{2.8}$$

stored as `Q`. If a dust component is present, the same will be computed for the dust and stored in `Qdust`, where c_s is then replaced by $\Omega_K H_d$ with H_d the vertical scale height of the dust layer. Example for gas:

```
from disklab.diskradial import *
import matplotlib.pyplot as plt
from disklab.natconst import *
d1=DiskRadialModel(mdisk=0.1*MS,rout=50*au,plsig=-1)
d1.compute_qtoomre()
d2=DiskRadialModel(mdisk=0.1*MS,rout=50*au,plsig=-1.5)
d2.compute_qtoomre()
plt.plot(d1.r/au,d1.qtoomre,label='plsig=-1')
plt.plot(d2.r/au,d2.qtoomre,label='plsig=-1.5')
plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$Q$')
plt.show()
```

2.5 Computing the deviation from Kepler rotation

For most purposes a protoplanetary disk can be considered to be perfectly Keplerian. However, when it comes to dust radial drift (see Chapters 6 and 6) even a small deviation from Kepler rotation can be important. And when the pressure in the disk midplane varies too strongly with radius (e.g. if one has dense gas rings or deep disk gaps) then the disk may become unstable and produce Rossby wave vortices.

So `DiskRadialModel` has some routines for computing these things.

The method `compute_omega()` computes the double-logarithmic derivative of the gas pressure $d\ln p/d\ln r$, where the gas pressure $p=\rho c_s^2$ is at the midplane. Optionally one can also use the vertically-integrated gas pressure $P=\Sigma c_s^2$, which is the relevant one for some 2-D (r,ϕ) hydrodynamic codes of disks. This option can be selected by setting `vertint=True`. But the default is the gas pressure at the midplane. This double-logarithmic derivative is stored as `dlnpdlnr`.

The Ω is then computed as

$$\Omega=\Omega_K\sqrt{1+\left(\frac{c_s^2}{v_K^2}\right)\frac{d\ln p}{d\ln r}} \tag{2.9}$$

where $v_K=\Omega_K r$. Furthermore following things are computed and stored:

$$v_\phi = \Omega r \tag{2.10}$$
$$l_\phi = \Omega r^2 \tag{2.11}$$
$$\delta v_\phi = v_\phi - v_K \tag{2.12}$$

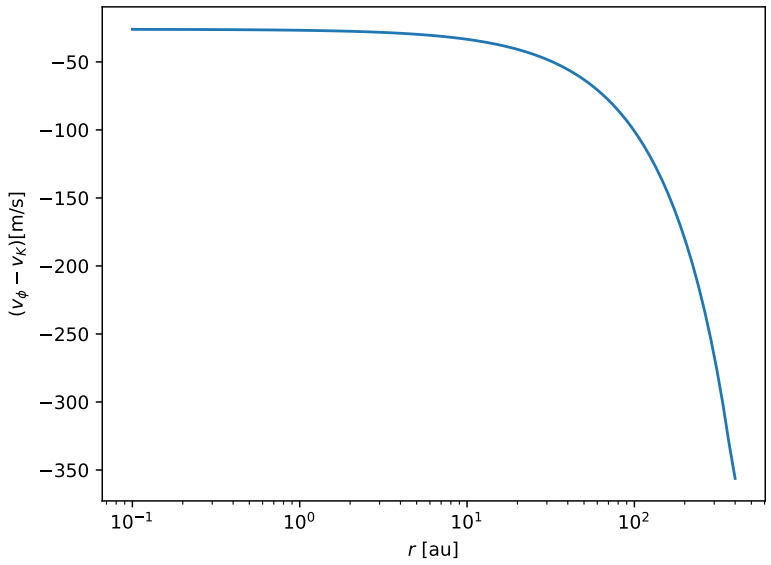
Example: let us compute the deviation from Kepler for a Lynden-Bell & Pringle model (we will use one of DISKLAB's standard disk models, of Chapter 3, the one described in Section 3.1.6). The code snippet is `snippet_deviation_kepler_1.py`. In Python run it as:

```
%run snippet_deviation_kepler_1.py
```

Here is the listing:

```
from snippet_header import plt, DiskRadialModel, MS, year, au, finalize

d = DiskRadialModel(rout=400 * au)
d.make_disk_from_lbp_alpha(1e-2 * MS, 1 * au, 1e-2, 1e5 * year)
d.compute_omega()
plt.plot(d.r / au, d.dvphi / 1e2)
plt.xscale('log')
plt.xlabel(r'$r$ [au]')
plt.ylabel(r'$(v_\phi-v_K) [\mathrm{m}/\mathrm{s}]$')
```

```
finalize(results=(d.dvphi))
```



The various things this method computes are:

```
d.omega       # Orbital angular frequency [1/s]
d.vphi        # Azimuthal gas velocity [cm/s]
d.dvphi       # Azimuthal gas velocity - Kepler velocity [cm/s]
d.lphi        # Specific angular momentum [cm^2/s]
d.dlnpdlnr    # Double-logarithmic derivative of the pressure
```

2.6 Computing the Solberg-Hoiland frequency

The method `compute_solberg_hoiland()` computes the value of the Solberg Hoiland criterion number

$$\mathrm{SH} = \kappa^2 + N^2 \tag{2.13}$$

If $\mathrm{SH} > 0$ then the disk is stable. If $\mathrm{SH} < 0$ then the disk is Rayleigh unstable. We follow Li, Finn, Lovelace & Colgate (2000) ApJ 533, 1023, Equation 22.

The κ is given by the derivative of the specific angular momentum in the following way:

$$\kappa^2 = \frac{1}{r^3}\frac{dl^2}{dr} \tag{2.14}$$

The Brunt-Vaisala frequency is given by:

$$N^2 = \frac{1}{\rho}\frac{dp}{dr}\left(\frac{1}{\rho}\frac{d\rho}{dr} - \frac{1}{\gamma p}\frac{dp}{dr}\right) \tag{2.15}$$

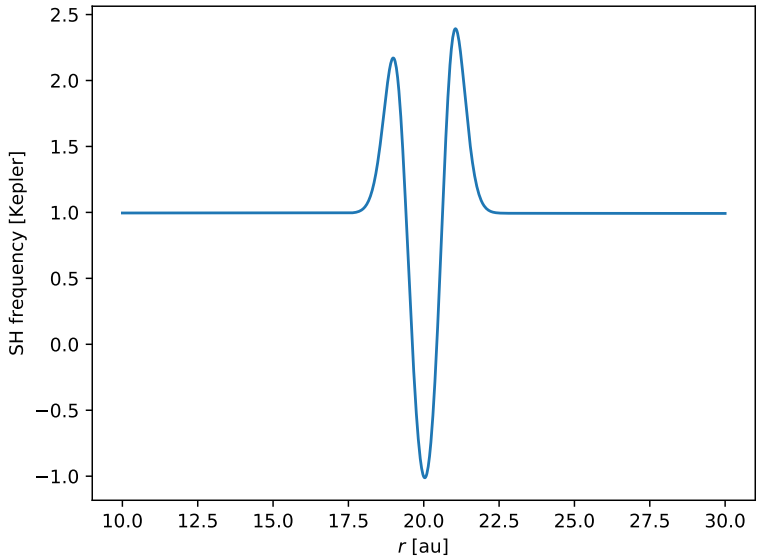
where γ is the adiabatic index.

Example where we add a bump to the disk and see if it is stable or not: The code snippet is `snippet_deviation_kepler_2.py`. In Python run it as:

```
%run snippet_deviation_kepler_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, year, au, finalize

d = DiskRadialModel(mdot=1e-8 * MS / year, rin=10 * au, rout=30 * au, nr=1000)

rbump    = 20. * au                      # Radial location of bump
abump    = 1.0                           # Amplitude of bump (relative)
hpbump   = np.interp(rbump, d.r, d.hp)   # Pressure scale height
wbump    = 0.5 * hpbump                  # Width (stand dev) of Gaussian bump
fact     = 1 + abump * np.exp(-0.5 * ((d.r - rbump) / wbump)**2)
d.sigma *= fact
d.rhomid *= fact

# Compute SH with Sigma and P=Sigma*c_s^2

d.compute_solberg_hoiland(vertint=True)
SHsqdimless = d.SHsq / d.omk**2      # SH freq in units of Kepler freq

# plotting

plt.plot(d.r / au, SHsqdimless)
plt.xlabel(r'$r$ [au]')
plt.ylabel('SH frequency [Kepler]')

finalize(results=(SHsqdimless))
```



There where the curve drops below 0 the disk is unstable. This is because the bump (i.e. ring) is too narrow.

2.7 For convenience: the `DiskRadialModel.plot()` method

Sometimes it might be useful to be able to plot certain results with a single command, rather than a series of `plt.xxxxx()` commands. The `DiskRadialModel.plot()` method allows you to make standard plots of any content of the `DiskRadialModel` object. Here is an example:

```
from disklab.diskradial import *
from disklab.natconst import *
d=DiskRadialModel(mdisk=0.01*MS)
```

```
d.plot(d.sigma,ylabel=r'$\Sigma [\mathrm{g/cm}^2]$')
```

You can, in fact, plot any kind of stuff, e.g.

```
d.plot(d.sigma*d.r**2,ylabel=r'$\Sigma\,r^2 [\mathrm{g}]$')
```

There are several keywords that can be set: type `d.plot?` for more information. *Note:* This method is just offered for convenience. There is no need to use it other than convenience. For the rest of this tutorial we will not use it, so that we can keep as closely to `matplotlib` style as possible to not complicate things unnecessarily.

Chapter 3

Built-in standard 1-D radial disk model setups

The class `DiskRadialModel` has several standard simple disk models that you can choose from. These are created by the methods with names starting with `make_disk_from_***`. The available models are descibed in the next subsections. In each of these subsections an example is given. If you want to make a plot of them: see Section 2.7. Note that in all the disk models described here you can also adapt the stellar parameters and grid parameters. Example:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(tstar=1e4,mstar=2.5*MS,lstar=40.*LS,rin=0.5*au,rout=300*AU,nr=1000)
d.make_disk_from_m_pl(mdisk=0.01*MS)
```

3.1 Standard 1-D radial disk models

3.1.1 Disk model: Powerlaw disk

The method `make_disk_from_powerlaw()` makes a powerlaw disk model according to the formula:

$$\Sigma_g(r) = \Sigma_0 \left(\frac{r}{r_0}\right)^p \tag{3.1}$$

Example

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel()
d.make_disk_from_powerlaw(1e2,1*au,-1.5)
```

3.1.2 Disk model: Powerlaw disk of given mass

The method `make_disk_from_m_pl()` makes a powerlaw disk model according to Eq. (3.1) where Σ_0 is computed such that the total disk mass is the value that is given as a parameter: `mdisk` (in gram).

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel()
d.make_disk_from_m_pl(mdisk=0.01*MS,plsig=-1.5)
```

3.1.3 Disk model: Fixed Toomre Q value

The method `make_disk_from_toomre()` makes a disk model by demanding that everywhere the Toomre Q value is as given:

$$\Sigma_g(r) = \frac{c_s \Omega_K}{\pi G Q} \tag{3.2}$$

Typically one would want to set $Q = 2$, so that you get the maximum surface density of the disk for marginal gravitational stability:

```
from disklab.diskradial import *
d = DiskRadialModel()
d.make_disk_from_toomre(2.0)
```

3.1.4 Disk model: Constant accretion rate disk

The method `make_disk_from_mdot()` creates a disk model that has a constant $\dot{M}$ throughout the grid. The standard viscous disk model is used, where the accretion rate is given by

$$\dot{M} = -2\pi r \Sigma_g(r) v_r(r) \tag{3.3}$$

where v_r is the radial velocity. Note that the minus sign is a convention: it is chosen such that inward motion ($v_r < 0$) is, by definition, a positive accretion rate. Again here: only the gas is accounted for. The radial velocity in steady state is:

$$v_r = -\frac{3}{2}\frac{\nu(r)}{r} \tag{3.4}$$

so that we obtain

$$\dot{M} = 3\pi \Sigma_g(r) \nu(r) \tag{3.5}$$

And so the formula for the surface density is then:

$$\Sigma_g(r) = \frac{\dot{M}}{3\pi\nu} \tag{3.6}$$

The viscosity ν is given by the standard Shakura & Sunyaev viscosity recipe:

$$\nu = \alpha \frac{c_s^2}{\Omega_K} \tag{3.7}$$

where the value of α is the variable `alpha` of the object, and can be given as an array (i.e. $\alpha = \alpha(r)$) or as a global constant (default is global constant at value `alpha = 0.01`). Example:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel()
d.make_disk_from_mdot(mdot=1e-8*MS/year)
```

3.1.5 Disk model: Lynden-Bell & Pringle

The method `make_disk_from_lyndenbellpringle()` sets up the most famous time-dependent disk model: the analytic solution by Lynden-Bell & Pringle.

We follow here the description from the paper Lodato, Scardoni, Manara & Testi (2017), MNRAS 472, 4700, their Eqs. 2, 3 and 4.

$$\Sigma_g(r) = \frac{M_0}{2\pi R_0^2}(2-\gamma)\left(\frac{R_0}{r}\right)^{\gamma} T^{-\eta} \exp\left(-\frac{(r/R_0)^{(2-\gamma)}}{T}\right) \tag{3.8}$$

with

$$\nu = \nu_0 \left(\frac{r}{R_0}\right)^{\gamma} \tag{3.9}$$

$$t_\nu = \frac{R_0^2}{3(2-\gamma)^2\nu_0} \tag{3.10}$$

$$T = 1+\frac{t}{t_\nu} \tag{3.11}$$

$$\eta = \frac{2.5-\gamma}{2-\gamma} \tag{3.12}$$

The R_0 variable sets the *initial* radius of the disk, the γ sets the powerlaw of the viscosity (for a "normal" constant α disk this is $\gamma = 1$), and M_0 sets the initial disk mass.

A key property of this model setup is that you have to specify the *time* at which you want to have the disk model. This time is the time after $t = 0$ of the Lynden-Bell & Pringle model. Example

```
from disklab.diskradial import *
from disklab.natconst import *
d    = DiskRadialModel()
M0   = 1e-2*MS
R0   = 1.0*au
nu0  = 1e10
gam  = 1.0
time = 1e5*year
d.make_disk_from_lyndenbellpringle(M0,R0,nu0,gam,time)
```

3.1.6 Disk model: Lynden-Bell & Pringle as an α-disk

The method `make_disk_from_lbp_alpha()` sets up a Lynden-Bell & Pringle model in which the ν is according to this α-recipe, with viscosity ν set according to Eq. (3.7),

Important: When using this method, the α must be a global constant or a global powerlaw. The γ of the Lynden-Bell & Pringle model will be computed as the double-logarithmic numerical derivative only from the first two grid points:

$$\gamma = \left.\frac{d\ln\nu}{d\ln r}\right|_{r=r_{\rm in}} \tag{3.13}$$

and ν_0 is obtained from

$$\nu_0 = \nu(r = R_0) \tag{3.14}$$

using numerical interpolation.

```
from disklab.diskradial import *
from disklab.natconst import *
d    = DiskRadialModel()
M0   = 1e-2*MS
R0   = 1.0*au
alpha= 1e-2
time = 1e5*year
d.make_disk_from_lbp_alpha(M0,R0,alpha,time)
```

3.1.7 Disk model: Simplified Lynden-Bell & Pringle

The method `make_disk_from_simplified_lbp()` computes a simplified Lynden-Bell & Pringle model often used in the literature. The time-dependent part is removed and only the following formula given:

$$\Sigma_g(r) = \Sigma_c \left(\frac{R_c}{r}\right)^{\gamma} \exp\left(-\left(\frac{r}{R_c}\right)^{2-\gamma}\right) \tag{3.15}$$

```
from disklab.diskradial import *
from disklab.natconst import *
d    = DiskRadialModel()
Rc   = 1.0*au
Sigc = 200.
gam  = 1.0
d.make_disk_from_simplified_lbp(Sigc,Rc,gam)
```

3.2 Modifying a standard disk model to your wishes

Standard disk models (such as those from Section 3.1) are all nice and good, but DISKLAB is meant for you to experiment with your own models. Usually this means you start with a standard model, and then modify it to your needs. The nice thing of the `DiskRadialModel` class is that you can modify the disk in any way you like.

3.2.1 Modifying a disk model "by hand"

The simplest form is simply to first create a disk model, and then modify it by hand. Example:

```
from disklab.diskradial import *
from disklab.natconst import *
import matplotlib.pyplot as plt
d = DiskRadialModel(mdisk=0.01*MS)
gapr = 10*au
gapw = 2*au
gapd = 0.9
factor = 1. - gapd*np.exp(-0.5*(d.r-gapr)**2/gapw**2)
d.sigma *= factor
d.add_dust(agrain=1e-4)
lam = 0.13
d.compute_onezone_intensity(lam)
plt.plot(d.r/au,d.intensity)
plt.xscale('log')
plt.yscale('log')
plt.show()
```

where we already used some elements from Section 6.1 and Section 9.2.

3.2.2 Built-in analytic planet gap models

Although the simple gap model built in by hand in Section 3.2.1 works fine, it can be sometimes useful to implement slightly more sophisticated planetary gap models. The `DiskRadialModel` class has a built-in routine for that, which contains some models and new models may be added later. Example

```
from disklab.diskradial import *
from disklab.natconst import *
import matplotlib.pyplot as plt
d = DiskRadialModel(mdisk=0.01*MS,nr=1000)
d.add_dust(agrain=1e-4)
d.add_planet_gap(5*au,'duffell',mpl=2e-4*MS,smooth=2.)
lam = 0.13
d.dust[0].grain.compute_simple_opacity(lam,tabulatemean=False)
d.compute_onezone_intensity(lam)
plt.plot(d.r/au,d.intensity[0,:])
plt.xlim(3,7)
plt.ylim(0,2e-12)
plt.show()
```

This punches a gap into a disk according to the model of Duffell (2015) ApJL 807, 11. To make the gap a bit more smooth (rather than the sudden flat-bottom gap of Duffell) a smoothing parameter is added.

Chapter 4

Disk midplane temperature: Irradiation, disk flaring, and viscous dissipation

So far we have assumed, for simplicity, that the incidence angle of the stellar radiation (the flaring angle $\varphi(r)$ stored as `d.flang`) is a constant throughout the disk, and that the midplane temperature is only given by the irradiation of the disk by the central star (see section 2.2). These assumptions made it easy to compute $T_{\rm mid}(r)$, by simply using Eq. (2.3).

However, the assumption that $\varphi(r)$ is a constant is not a very good assumption. For quick testing purposes it is ok, but for more accurate calculations it is not recommended. We want to compute it self-consistently. We follow Chiang & Goldreich (1997, henceforth CG97) here. The main new concept we introduce here is the *surface height* $H_s(r)$, which is typically larger than the pressure scale height $H_p(r)$, and represents the height above the midplane where the disk becomes optically thick to the stellar radiation.

4.1 Computing $H_s(r)$ and $\varphi(r)$ mutually self-consistently

First let us do the exercise of computing $H_s(r)$ and $\varphi(r)$ self-consistently for a fixed $H_p(r)$. That is: we do not treat (for now) the pressure scale height $H_p(r)$ to be dependent on $\varphi(r)$, even though it is through the midplane temperature and sound speed.

The flaring angle is, according to CG97, their Eq. 5, given by

$$\varphi(r) = \frac{0.4R_*}{r} + r\frac{d}{dr}\left(\frac{H_s}{r}\right) \tag{4.1}$$

where H_s is the surface height of the disk. The surface height is defined by the height above the midplane $z = H_s$ where the optical depth with respect to stellar radiation is unity. Let us compute this height.

Let us first define the dust opacity at stellar wavelength to be κ_*, and that only the dust produces opacity (not the gas). Let us assume that the dust and the gas are well mixed, so that the dust-to-gas ratio is the same at all z. Let us call this dust-to-gas ratio dtg. Then the vertical optical depth to stellar radiation τ_* (without the flaring angle) is

$$\tau_* = {\rm dtg}\,\Sigma_g\,\kappa_* \tag{4.2}$$

Next we assume that the gas density is vertical distributed according to the following Gaussian:

$$\rho_g(r,z) = \frac{\Sigma_g(r)}{\sqrt{2\pi}H_p}\exp\left(-\frac{z^2}{2H_p^2}\right) \tag{4.3}$$

where H_p is the pressure scale height given by Eq. (2.5). We assume that the dust density follows the gas density: $\rho_d(r,z) = {\rm dtg}\,\rho_g(r,z)$, by virtue of the assumption of perfect vertical mixing.

Given the irradiation angle $\varphi(r)$, which we assume to *not* depend on z, we can compute the optical depth to stellar radiation along the incident angle as a function of z:

$$\tau_{*,\mathrm{irr}}(r,z) = \frac{\mathrm{dtg}\,\kappa_*}{\varphi(r)} \int_z^\infty \rho_g(r,z)dz \tag{4.4}$$

We now wish to find the z (for given r) at which $\tau_{*,\mathrm{irr}}(r,z) = 1$. We follow Dullemond & Dominik (2001), appendix A2, their Eq. A9. In our form this amounts to solving

$$1 - \mathrm{erf}\left(\frac{H_s}{\sqrt{2}H_p}\right) = \frac{2\varphi}{\tau_*} \tag{4.5}$$

for H_s.

Now that we have $H_s(r)$ we can integrate Eq. (4.1) to obtain $\varphi(r)$ and then we iterate until convergence. Unfortunately this procedure is numerically unstable. Chiang et al. (2001) ApJ 547, 1077 devised a method to keep it stable. Our version of it is to define the *flaring index* ξ as follows:

$$\xi(r) = \frac{d\ln(H_s/r)}{d\ln r} \tag{4.6}$$

so that the *flaring angle* (Eq. 4.1) becomes:

$$\varphi(r) = \frac{0.4R_*}{r} + \xi(r)\frac{H_s}{r} \tag{4.7}$$

The flaring index $\xi(r)$ is now computed in a pairwise fashion, where gridpoints `i` and `i+1` store the flaring index computed from gridpoints `i-1` and `i-2`. This means $\xi_{i+1} = \xi_i$ for all even `i`. The inner two gridpoints have, by this procedure, no value. We take them to be copies of the next two. This procedure turns out to converge quickly.

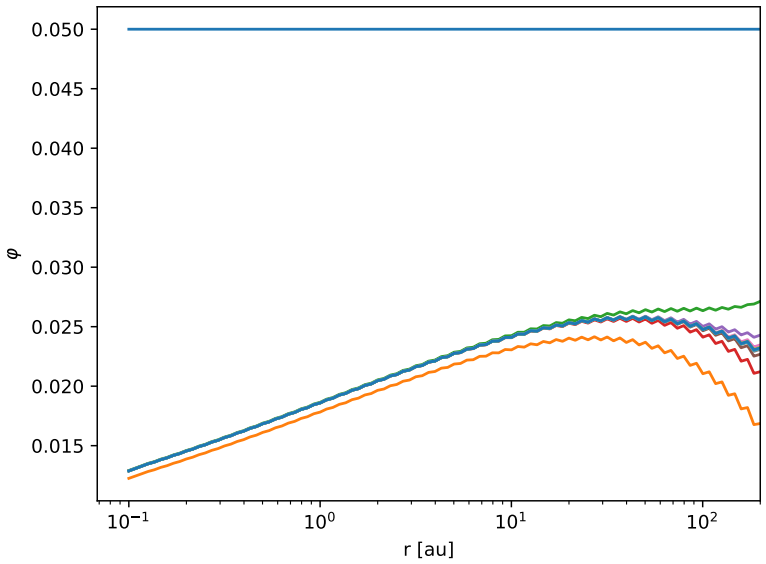
The procedure to compute H_s by solving Eq. (4.5) is called `compute_hsurf()`. The procedure for computing the flaring index according to Eq. (4.6) (with the pairwise method) is `compute_flareindex()`. Finally, the procedure to compute $\varphi(r)$ from $\xi(r)$ according to Eq. (4.7) is called `compute_flareangle_from_flareindex()`. By iterating these you can obtain the self-consistent flaring geometry.

The code snippet is `snippet_iterate_flang_1.py`. In Python run it as:

```
%run snippet_iterate_flang_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, au, finalize

d = DiskRadialModel(mdisk=0.01 * MS)
kappa = 1e3    # Take a somewhat low kappa for now
d.meanopacitymodel = ['supersimple', {'dusttogas': 0.01, 'kappadust': kappa}]
d.compute_mean_opacity()

plt.figure()
plt.plot(d.r / au, d.flang + np.zeros(len(d.r)))
for iter in range(10):
    d.compute_hsurf()
    d.compute_flareindex()
    d.compute_flareangle_from_flareindex(inclrstar=False)
    plt.plot(d.r / au, d.flang)
plt.xscale('log')
plt.xlabel('r [au]')
plt.ylabel(r'$\varphi$')
plt.xlim(right=200)

finalize(results=(d.flang))
```

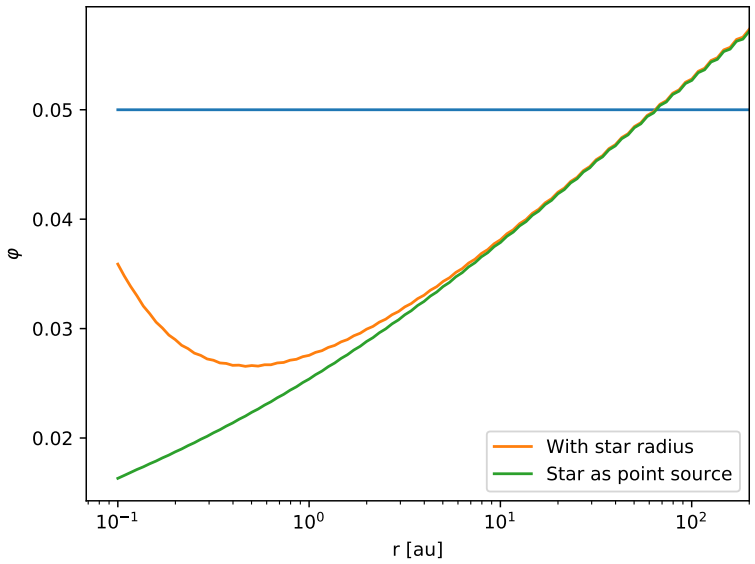


Rather than doing the iteration by hand as above, you can also use `iterate_flaringangle()`. The code snippet is `snippet_iterate_flang_2.py`. In Python run it as:

```
%run snippet_iterate_flang_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, au, finalize

d = DiskRadialModel(mdisk=0.01 * MS)
kappa = 1e5
d.meanopacitymodel = ['supersimple', {'dusttogas': 0.01, 'kappadust': kappa}]
d.compute_mean_opacity()

plt.figure()
plt.plot(d.r / au, d.flang + np.zeros(len(d.r)))
iter1 = d.iterate_flaringangle(inclrstar=True, itermax=20, convcrit=1e-2, iterhp=False)
flangwithstar = d.flang
iter2 = d.iterate_flaringangle(inclrstar=False, itermax=20, convcrit=1e-2, iterhp=False)
flangpointstar = d.flang

plt.plot(d.r / au, flangwithstar, label='With star radius')
plt.plot(d.r / au, flangpointstar, label='Star as point source')
plt.xscale('log')
plt.xlim(right=200)
plt.xlabel('r [au]')
plt.ylabel(r'$\varphi$')
plt.legend(loc='lower right')
print('Nr of iterations = {}, {}'.format(iter1, iter2))

finalize(results=(flangwithstar,flangpointstar))
```



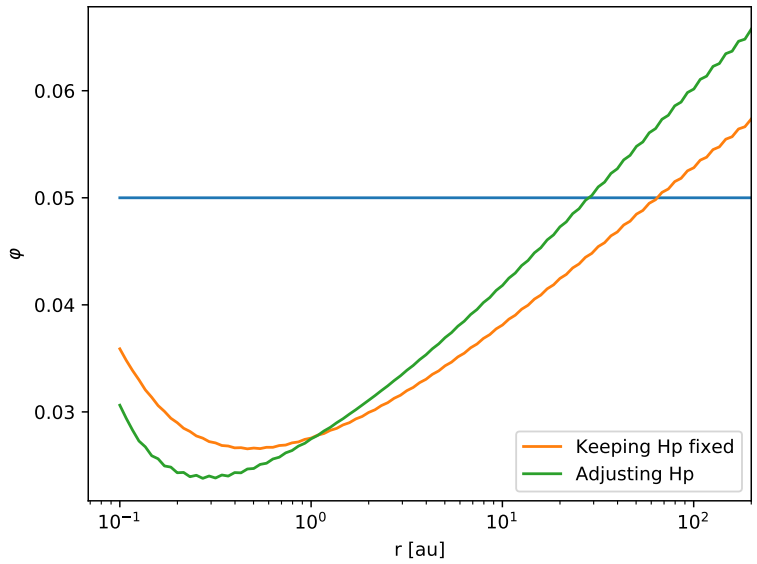
Here we also compare the effect when we include or not include the $0.4R_*/r$ term in Eq. (4.7), which is the term that takes care of the fact that the star is extended (i.e. not a point source) and can thus irradiation even a disk that is not flaring.

Finally, we can now include the self-consistent determination of the pressure scale height $H_p(r)$ into the iteration, because the flaring angle determines $T_{\rm mid}$, which determines c_s, which determines H_p. The code snippet is `snippet_iterate_flang_3.py`. In Python run it as:

```
%run snippet_iterate_flang_3.py
```

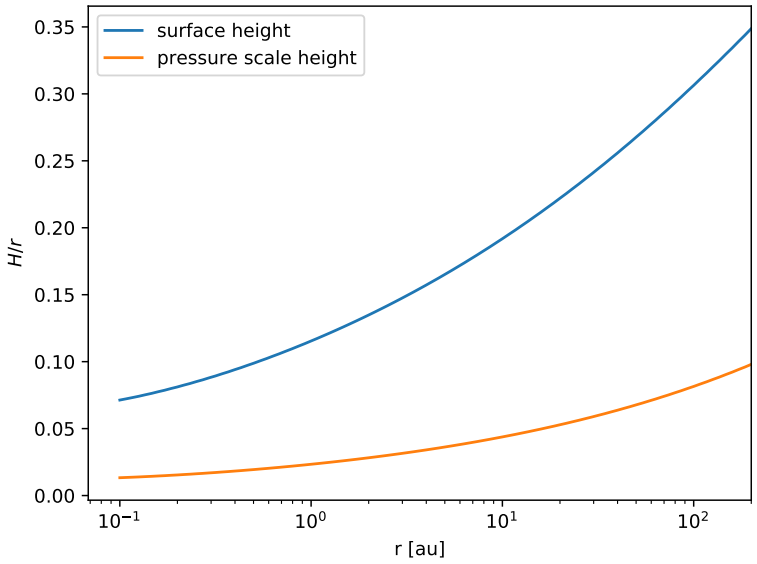
Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, au, finalize

d = DiskRadialModel(mdisk=0.01 * MS)
kappa = 1e5
d.meanopacitymodel = ['supersimple', {'dusttogas': 0.01, 'kappadust': kappa}]
d.compute_mean_opacity()

plt.figure()
plt.plot(d.r / au, d.flang + np.zeros(len(d.r)))

iter1 = d.iterate_flaringangle(inclrstar=True, itermax=20, convcrit=1e-2, iterhp=False)
flanghpfixed = d.flang
iter2 = d.iterate_flaringangle(inclrstar=True, itermax=20, convcrit=1e-2, iterhp=True)
flanghpvary = d.flang

plt.plot(d.r / au, flanghpfixed, label='Keeping Hp fixed')
plt.plot(d.r / au, flanghpvary, label='Adjusting Hp')
plt.xscale('log')
plt.xlim(right=200)
plt.xlabel('r [au]')
plt.ylabel(r'$\varphi$')
plt.legend(loc='lower right')
print('Nr of iterations = {}, {}'.format(iter1, iter2))

finalize(results=(flanghpfixed,flanghpvary))
```



You can also now inspect how the new value of $H_p(r)$ is. This is usually best done by plotting the ratio $H_p(r)/r$. The code snippet is `snippet_iterate_flang_4.py`. In Python run it as:

```
%run snippet_iterate_flang_4.py
```



IMPORTANT: For disks that are not very optically thick anymore, and/or for the very outer regions of the disk, the disk is no longer flaring. The disk becomes *self-shadowed*, and thus the above mathematics will fail. The only real solution would then be to resort to actual 2-D/3-D radiative transfer, which is not included in the `diskradial` module. It is also important to keep in mind that the puffed-up inner rim is not included in these calculations, and also there the only real solution is to resort to 2-D/3-D radiative transfer. See Section 12.2 for some more notes on this.

4.2 Including viscous heating in midplane temperature computation

In Subsection 2.2 and in Section 4.1 we compute the midplane temperature from the irradiation by the central star. However, a viscously accreting disk also produces its own heating. Including this into the computation of $T_{\rm mid}(r)$ is both easy and hard. It is easy because one can easily add a recipe to do that. It is hard, because (a) it is not really well known at which height in the disk the heat is deposited and (b) the midplane temperature depends on the dust opacity. The dust can sublimate if the temperature at the midplane gets too high, which leads to a kind of "thermostat effect".

The viscous heating rate is well-defined:

$$Q_{\rm visc}(r) = \frac{9}{4}\Sigma_g(r)\nu(r)\Omega_K(r)^2 \tag{4.8}$$

The effective temperature at the surface of the disk (assuming the disk is optically thick) is easily computed, by setting $Q_{\rm visc}(r) = Q_{\rm cool}(r)$, with $Q_{\rm cool}(r)$ given by Eq. (2.2):

$$2\sigma_{\rm SB}T_{\rm eff}(r)^4 = \frac{9}{4}\Sigma_g(r)\nu(r)\Omega_K(r)^2 \tag{4.9}$$

At this point it becomes important to realize that if the Rosseland mean optical depth of the disk $\tau_{\rm Ross}$ becomes lower than about unity, Eq. (4.9) becomes invalid. In such a case one should replace Eq. (2.2) with

$$Q_{\rm cool}(r) = 2\sigma_{\rm SB}T_{\rm eff}(r)^4\left(1-e^{-2\tau_{\rm Ross}}\right) \tag{4.10}$$

where $\tau_{\rm Ross}$ is the Rosseland optical depth of the disk:

$$\tau_{\rm Ross} = \Sigma_d\,\kappa_{d,\rm Ross} \tag{4.11}$$

where $\kappa_{d,\rm Ross}$ is the Rosseland opacity of the dust and Σ_d the dust surface density. The cooling rate formula of Eq. (4.10) is approximately valid for any value of $\tau_{\rm Ross}$. For $\tau_{\rm Ross}\gg 1$ it converges to the familiar Eq. (2.2). For $\tau_{\rm Ross}\ll 1$ it converges to $Q_{\rm cool}\rightarrow 4\sigma_{\rm SB}\tau_{\rm Ross}T_{\rm eff}^4$, which is the optically thin cooling rate. With these changes, Eq. (4.9) changes to

$$2\sigma_{\rm SB}T_{\rm eff}(r)^4 = \frac{9}{4}\left(1-e^{-2\tau_{\rm Ross}}\right)^{-1}\Sigma_g(r)\nu(r)\Omega_K(r)^2 \tag{4.12}$$

Another difficulty lies in the fact that the midplane temperature is not equal to the effective temperature. Assuming that all the viscous heat is deposited at the midplane, the relation between $T_{\rm eff}$ and $T_{\rm mid}$ is

$$T_{\rm mid}^4 = \left(\frac{1}{2}\tau_{\rm Ross}+1\right)T_{\rm eff}^4 \tag{4.13}$$

The factor 1/2 in Eq. (4.13) comes in because $\tau_{\rm Ross}$ is the *total* vertical optical depth, while we need the optical depth from the midplane to the surface. The $+1$ inside the brackets of Eq. (4.13) is a approximation to keep the equation also valid for $\tau_{\rm Ross}\ll 2$.

In principle Eqs. (4.12) and (4.13) complete the formula for the midplane temperature:

$$2\sigma_{\rm SB}T_{\rm mid}(r)^4 = \frac{9}{4}\left(\tfrac{1}{2}\tau_{\rm Ross}+1\right)\left(1-e^{-2\tau_{\rm Ross}}\right)^{-1}\Sigma_g(r)\nu(r)\Omega_K(r)^2 \tag{4.14}$$

Although it seems that Eq. (4.14) gives the final answer, there are several loose ends:

1. The value of $\nu(r)$ depends on the midplane temperature itself (Eq. 3.7). This means iteration is required.
2. The Rosseland opacity $\kappa_{d,\rm Ross}(r)$ also depends on the midplane temperature, because it is the Rosseland-average at the temperature of the infrared radiation inside the disk. It is certainly not the same as the κ we used in Section 4.1 for computing the flaring angle (which is the opacity at stellar wavelengths).
3. To do it right, one should spatially resolve the disk vertically, because the temperature is a gradial function of vertical coordinate z. But usually this is ignored for simplicity.
4. Usually *both* viscous heating and irradiation take place. The formula Eq. (4.14) only treats the viscous heating.

5. The viscous heating can lead to the inner disk regions becoming *self-shadowed*. This means that the irradiation by the central star of this region is zero. In reality it is, however, not exactly zero because the disk surface is not infinitely geometrically thin. Also radial radiative diffusion will transport heat into this region, but treating this requires full 2-D/3-D radiative transfer.

Concerning point 1: In principle one could solve Eq. (4.14) in the simplest possible iterative way: keep $\tau_{\rm Ross}$ and ν from the previous iteration, and evaluate the right-hand-side of Eq. (4.14), thus obtaining $T_{\rm mid}$, and then re-evaluate $\tau_{\rm Ross}$ and ν for this new $T_{\rm mid}$ and repeat the process until convergence. This is the solution method used when you call `compute_disktmid()` with the keyword `simple=True`. But this simple iteration can fail, in particular if dust sublimation is included in the mean opacity model.

Instead, by default the `compute_disktmid()` method uses Brent's root-finding algorithm[1] to solve for $T_{\rm mid}$. This is a much more robust method.

Concerning point 4: combining viscous heating and irradiation. There is no unique way, because in reality one would have to resolve the vertical structure. We do this here in the following way: we compute the midplane temperature due to iraddiation-only and call this $T_{\rm irrad}$, and compute the midplane temperature due to viscous-heating-only and call this $T_{\rm visc}$. Then we combine them in the following way:

$$T_{\rm mid} = \left(T_{\rm irrad}^4 + T_{\rm visc}^4\right)^{1/4} \tag{4.15}$$

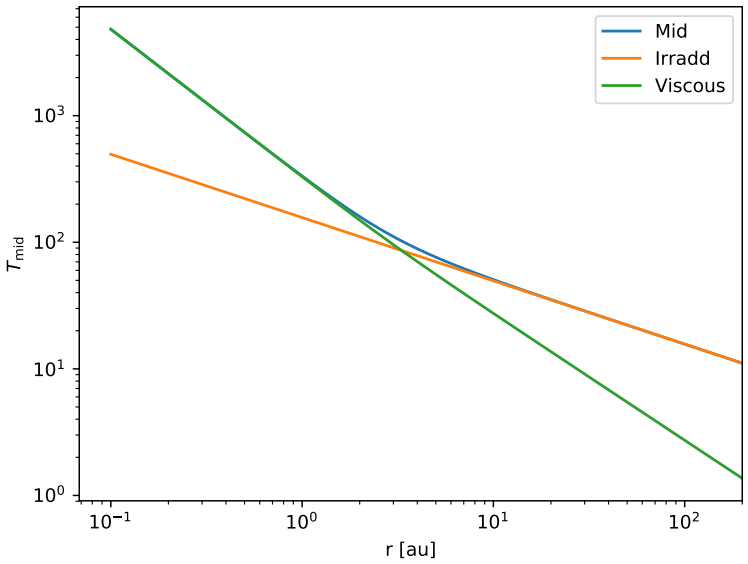
Let's compute this in an example. The code snippet is `snippet_tmid_viscousheating_1.py`. In Python run it as:

```
%run snippet_tmid_viscousheating_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, plt, MS, au, finalize

d = DiskRadialModel(mdisk=0.01 * MS)
d.add_dust(agrain=1e-4)
kappa = 1e3
d.meanopacitymodel = ['supersimple', {'dusttogas': 0.01, 'kappadust': kappa}]
d.compute_mean_opacity()
d.compute_disktmid(vischeat=True)

plt.figure()
plt.plot(d.r / au, d.tmid, label='Mid')
plt.plot(d.r / au, d.tirr, label='Irradd')
plt.plot(d.r / au, d.tvisc, label='Viscous')
plt.xscale('log')
plt.yscale('log')
plt.xlim(right=200)
plt.xlabel('r [au]')
plt.ylabel(r'$T_{\mathrm{mid}}$')
plt.legend()

finalize(results=(d.tmid,d.tirr,d.tvisc))
```

[1] https://en.wikipedia.org/wiki/Brent's_method



Note that here we kept the flaring angle fixed at $\varphi(r) = 0.05$.

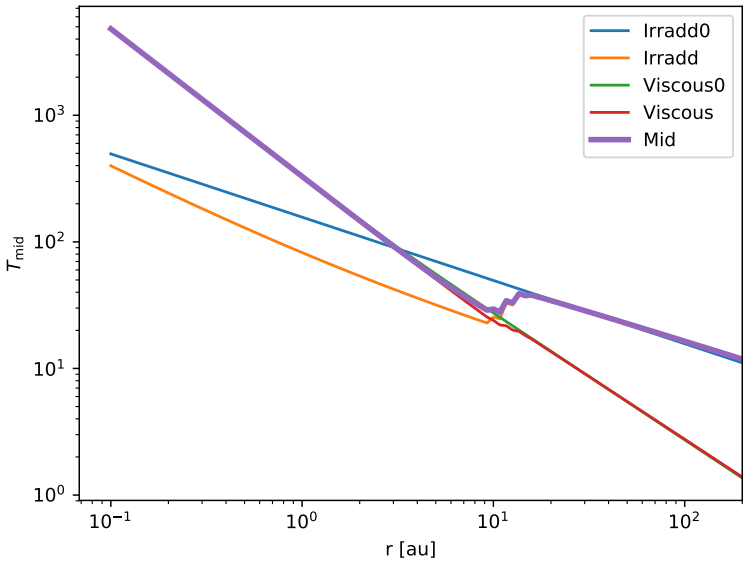
If we re-compute the flaring angle after the inclusion of the viscous heating, we will encounter the effect of self-shadowing in the viscously heated part of the disk. Since self-shadowing can not be properly modeled using the flaring angle recipe, it can lead to somewhat odd behavior of the result. In the following example such behavior of the model is shown. The code snippet is `snippet_tmid_viscousheating_2.py`. In Python run it as:

```
%run snippet_tmid_viscousheating_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, plt, MS, au, finalize

d = DiskRadialModel(mdisk=0.01 * MS)
d.add_dust(agrain=1e-4)
kappainfrared = 1e3
kappastellar  = 1e5
d.meanopacitymodel = ['supersimple', {'dusttogas': 0.01, 'kappadust': kappainfrared}]
d.compute_mean_opacity()
d.compute_disktmid(vischeat=True)
tirr0  = 1.0 * d.tirr
tvisc0 = 1.0 * d.tvisc
d.meanopacitymodel = ['supersimple', {'dusttogas': 0.01, 'kappadust': kappastellar}]
d.compute_mean_opacity()
d.iterate_flaringangle(inclrstar=True, itermax=20, convcrit=1e-2, iterhp=True, keeptvisc=True)
d.meanopacitymodel = ['supersimple', {'dusttogas': 0.01, 'kappadust': kappainfrared}]
d.compute_mean_opacity()
d.compute_disktmid(vischeat=True)

plt.figure()
plt.plot(d.r / au, tirr0, label='Irradd0')
plt.plot(d.r / au, d.tirr, label='Irradd')
plt.plot(d.r / au, tvisc0, label='Viscous0')
plt.plot(d.r / au, d.tvisc, label='Viscous')
plt.plot(d.r / au, d.tmid, label='Mid', linewidth=3)
plt.xscale('log')
plt.yscale('log')
plt.xlim(right=200)
plt.xlabel('r [au]')
plt.ylabel(r'$T_{\mathrm{mid}}$')
```

```
plt.legend()

finalize(results=(tirr0,tvisc0,d.tmid,d.tirr,d.tvisc))
```



Until about 10 au the disk midplane is dominated by the viscous heating (compare the "Irrad" curve to the "Viscous" curve: the latter being much larger than the former). This make the disk more "conical" in shape, and thus aggravates the effect of the reduced contribution of the irradiation: The disk wants to become self-shadowed here. Beyond 10 au, however, the irradiation dominates over the viscous heating, making the disk strongly flaring, which increases the flaring angle, and thus reinforces the effect. The dramatic step in temperature around 10 au is likely an artifact of the iteration of the flaring angle recipe in part of the disk that "wants" to become self-shadowed. In reality the jump is not expected to be so strong. But a proper treatment requires 2D/3D radiative transfer.

Before concluding this section, let us go back to the previous simple case, but now replace the supersimple opacity with a more realistic opacity: the Bell & Lin opacity model (see Section 8.5).

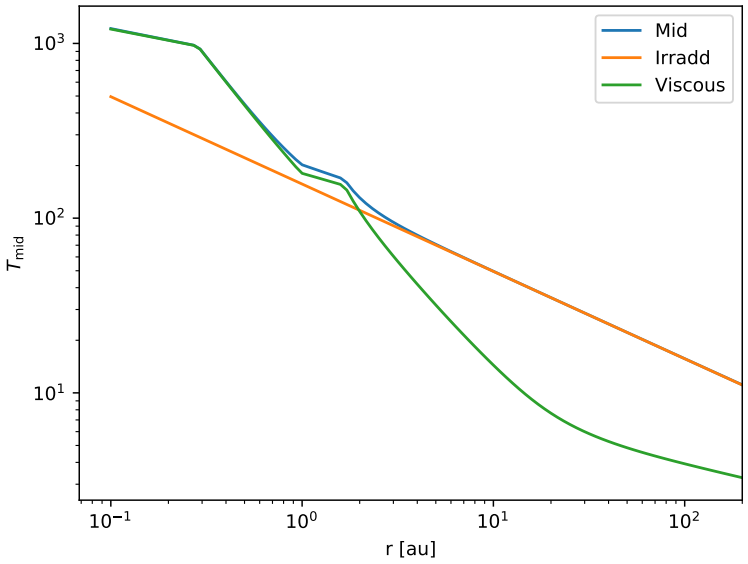
The code snippet is `snippet_tmid_viscousheating_3.py`. In Python run it as:

```
%run snippet_tmid_viscousheating_3.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, plt, MS, au, finalize

d = DiskRadialModel(mdisk=0.01 * MS)
d.add_dust(agrain=1e-4)
kappa = 1e3
d.meanopacitymodel = ['belllin']
d.compute_mean_opacity()
d.compute_disktmid(vischeat=True)

plt.figure()
plt.plot(d.r / au, d.tmid, label='Mid')
plt.plot(d.r / au, d.tirr, label='Irradd')
plt.plot(d.r / au, d.tvisc, label='Viscous')
plt.xscale('log')
plt.yscale('log')
plt.xlim(right=200)
plt.xlabel('r [au]')
plt.ylabel(r'$T_{\mathrm{mid}}$')
plt.legend()
```

```
finalize(results=(d.tmid,d.tirr,d.tvisc))
```



Compared to the `snippet_tmid_viscousheating_1.py` case, which had a smooth transition from irradiated (in the outer region) to viscous (inner region), we now see several kinks in the temperature profile. The temperature seems to nearly level off to constant within two regions. These are the ice line (outer leveled-off region, between about 1 and 2 au) and the dust sublimation line (inner leveled-off region, inward of about 0.3 au). This leveling off is a kind of "thermostat effect" when the ice or dust starts to sublimate. In the Bell & Lin opacity model the sublimation of ice and silicates occur gradually over an extended range in temperature (see the figure showing the Bell & Lin opacity model of Section. 8.3.4), whereas in the opacity model used in `snippet_viscdiskevol_4.py` (Section 5.1) the sublimation occurs at exactly a single temperature (leading to the exactly leveled temperature profiles in that model). Reality probably lies in between these two extremes: At higher densities the sublimation temperature is at somewhat higher temperature.

4.3 Some cautionary comments about the midplane temperature

As you can see, the computation of the midplane temperature in a self-consistent manner is not quite easy. The self-consistent computation of the flaring angle is not always really possible, because the flaring-angle-recipe itself may break down due to multi-dimensional radiative transfer effects. To include such multi-dimensional radiative transfer effects you either have to resort to full-scale radiative transfer codes such as `RADMC-3D` (see e.g. Chapter 13), or to simplified 2-D radiative diffusion-based methods such as described in Section 12.4 in the Chapter on vertical structure models (Chapter 11).

Also the viscous heating may not be well understood: Is the heat deposited always near the midplane (proportional to the density) or is it deposited higher up in the disk? And what is the effect of thermal sublimation of dust on the midplane temperature?

And will the environment affect the temperature? Surely the temperature should not be allowed to drop below the 2.73 microwave background temperature. But given that stars form in star formation regions, the cumulative radiation from the nearby young stars will likely provide a higher background temperature. It may therefore be necessary to set a lower limit to the disk temperature you compute from the above simple estimates, in particular for very large disks that tend to become very cold in the outer regions. By default, DISKLAB uses a background temperature of $T_{\rm bg} = 2.725$, i.e. the cosmic microwave background temperature. But you can set this to a higher value if the protoplanetary disk object resides in a cluster of stars. The best is to specify this at the very start by setting the `tbg` keyword in your call to `DiskRadialModel`. For instance:

```
d = DiskRadialModel(mstar=2*MS,lstar=10*LS,rin=0.1*au,rout=1000*au,nr=300,tbg=20.)
```

Please keep in mind the simplicity of this method. A more self-consistent radiative transfer computation can be done with codes such as RADMC-3D (see e.g. Chapter 13).

Another potential problem with the determination of the midplane temperature can emerge when we combine viscous heating with time-dependent viscous evolution (see Section 5.5). It often works out of the box, but sometimes, when the opacity model is complex (e.g. has dust evaporation jumps), it can become numerically unstable. To solve this, please refer to Section 5.5.

4.4 A note on constant aspect ratio models

In many papers on the hydrodynamics of planet-disk interaction and/or magnetohydrodynamic disk behavior a set of standard assumptions are made that will make the comparison to the models of DISKLAB a bit difficult.

First, these models often assume that the disk has a "constant aspect ratio" of, say, 0.05. This is another way of saying that, by assumption, $h_p/r = 0.05$. This means that the midplane disk temperature as a function of radius is $T(r) = (\mu m_p/k_{\rm B})(0.05)^2 v_K^2(r) \propto r^{-1}$. This is a steeper temperature profile than we find in the above sections, at least for the irradiation-dominated part of the disk. The irradiation-dominated disks have a flaring geometry, i.e. $d(h_p/r)/dr > 0$, i.e. a less steep temperature decline with radius. If the disk is viscous-heating dominated, the opposite can be the case. If you wish to make DISKLAB models that can be compared to constant aspect ratio models from the literature, you can impose this by hand, for instance like this:

```
hr      = 0.05                # Choice of aspect ratio h_p/r
d.cs    = hr*d.r*d.omk
d.hp    = d.cs/d.omk
d.tmid  = 2.3*mp*d.cs**2/kk   # 2.3 is the mean molecular weight
```

Second, these hydrodynamics models are often expressed in "code units". In constrast, DISKLAB does not use code units: everything is in real units (CGS). Any comparison requires a proper conversion.

Chapter 5

Time-dependent viscous disk evolution

5.1 Time-dependent viscous disk

The method `get_viscous_evolution_next_timestep()` allows the disk model to be viscously advanced in time by a single time step Δt. This time-advancement is done using an *implicit integration method*, and thus is stable for any value of Δt. But of course it is better (more accurate) to do several small time steps than one huge time step. Experimentation may be required.

The time-dependent viscous disk equation is:

$$\frac{\partial \Sigma_g}{\partial t} + \frac{1}{r}\frac{\partial}{\partial r}\left(r\Sigma_g v_r\right) = \dot{\Sigma}_g \tag{5.1}$$

where $\dot{\Sigma}_g$ is a possible source term (e.g. infall of gas from an envelope, see Section 5.4). Also a negative source term (e.g. photoevaporation) is possible, but that could easily lead to negative surface densities, which would then also mess up the mass conservation. The radial velocity due to the viscosity is:

$$v_r = -\frac{3}{\sqrt{r}\Sigma_g}\frac{\partial(\sqrt{r}\Sigma_g \nu)}{\partial r} \tag{5.2}$$

where ν is the viscosity coefficient given by Eq. (3.7), but the viscosity could have any shape. If we insert Eq. (5.2) into Eq. (5.1) we obtain

$$\frac{\partial \Sigma_g}{\partial t} - \frac{3}{r}\frac{\partial}{\partial r}\left(\sqrt{r}\frac{\partial(\sqrt{r}\Sigma_g \nu)}{\partial r}\right) = \dot{\Sigma}_g \tag{5.3}$$

We can now resort to a general-purpose Python subroutine `solvediffonedee` for solving (or advancing in time) a standard diffusion equation. This method is described in appendix A.

The viscous disk equation Eq. (5.1) can be cast into the standard form of Eq. (A.1), so that we can use the subroutine `solvediffonedee` to advance the viscous disk in time.

If define $\sigma = 2\pi\Sigma_g r$ and $\dot{\sigma} = 2\pi\dot{\Sigma}_g r$ we obtain:

$$\frac{\partial \sigma}{\partial t} - 3\frac{\partial}{\partial r}\left(\sqrt{r}\frac{\partial}{\partial r}\left(\sigma\frac{\nu}{\sqrt{r}}\right)\right) = \dot{\sigma} \tag{5.4}$$

Next define $g(r) = \sqrt{r}/\nu$:

$$\frac{\partial \sigma}{\partial t} - 3\frac{\partial}{\partial r}\left(\nu g\frac{\partial}{\partial r}\left(\frac{\sigma}{g}\right)\right) = \dot{\sigma} \tag{5.5}$$

Next define $D = 3\nu$:

$$\frac{\partial \sigma}{\partial t} - \frac{\partial}{\partial r}\left(D g\frac{\partial}{\partial r}\left(\frac{\sigma}{g}\right)\right) = \dot{\sigma} \tag{5.6}$$

This means that we can now use the standard diffusion solver with $y = \sigma$ and $x = r$.

For completeness: the radial velocity is:

$$v_r = -\frac{3}{\Sigma\sqrt{r}}\frac{\partial}{\partial r}\left(\sqrt{r}\Sigma\nu\right) = -\frac{D}{\sigma}g\frac{\partial}{\partial r}\left(\frac{\sigma}{g}\right) \tag{5.7}$$

The accretion rate is then:

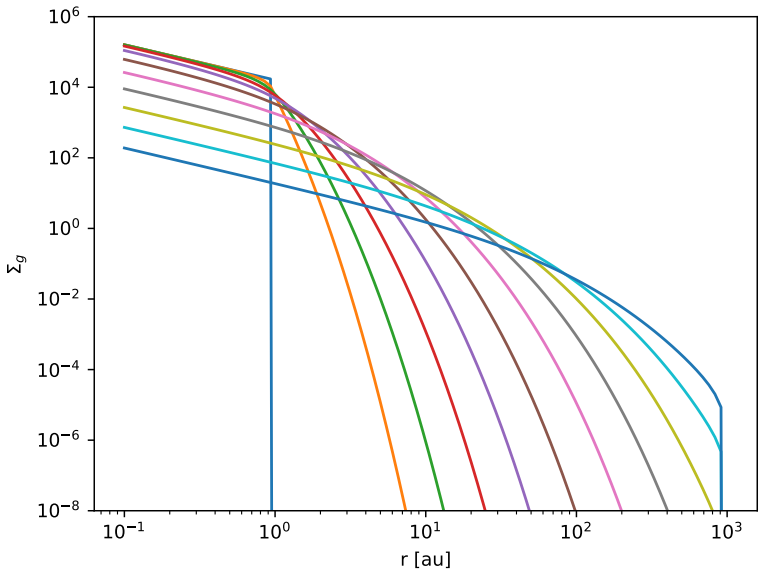
$$\dot{M}(r,t) = -\sigma v_r = Dg\frac{\partial}{\partial r}\left(\frac{\sigma}{g}\right) \tag{5.8}$$

Example: we set up an initial disk model and let it evolve in time. The code snippet is `snippet_viscdiskevol_1.py`. In Python run it as:

```
%run snippet_viscdiskevol_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, year, au, finalize

d = DiskRadialModel(rout=1000 * au, alpha=1e-3)
d.make_disk_from_m_pl(mdisk=0.01 * MS, rdisk=1 * au)

plt.figure()
plt.plot(d.r / au, d.sigma)
plt.xscale('log')
plt.yscale('log')

ntime = 10
tstart = 1e3 * year     # Initial time at 1000 year (for log-spaced timesteps)
tend = 1e7 * year       # Final time at 10 Myr
time = tstart * (tend / tstart)**(np.linspace(0., 1.,
                                              ntime + 1))   # Logarithmic spacing
for itime in range(1, ntime + 1):
    d.sigma = d.get_viscous_evolution_next_timestep(
        time[itime] - time[itime - 1])
    plt.plot(d.r / au, d.sigma)
plt.ylim(1e-8, 1e6)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_g$')

finalize(results=(d.sigma))
```



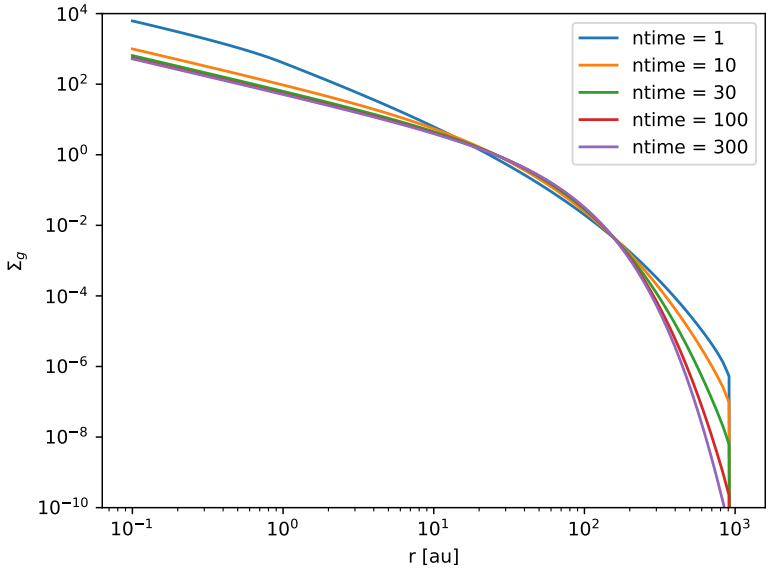
The different lines are different time snapshots, with logarithmic time intervals.

Ten time steps, logarithmic in time, as shown here is a very coarse time-resolution. Let us check how the time step size affects the final result (here taken at 3 Myr). The code snippet is `snippet_viscdiskevol_2.py`. In Python run it as:

```
%run snippet_viscdiskevol_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, year, au, finalize

ntime  = np.array([1, 10, 30, 100, 300])
tstart = 1e3 * year
tend   = 3e6 * year

plt.figure()
for iter in range(len(ntime)):
    d = DiskRadialModel(rout=1000 * au, alpha=1e-3)
    d.make_disk_from_m_pl(mdisk=0.01 * MS, rdisk=1 * au)
    time = tstart * (tend / tstart)**(np.linspace(0., 1., ntime[iter] + 1))
    for itime in range(1, ntime[iter] + 1):
        d.sigma = d.get_viscous_evolution_next_timestep(
            time[itime] - time[itime - 1])
    plt.plot(d.r / au, d.sigma, label='ntime = {}'.format(ntime[iter]))

plt.xscale('log')
plt.yscale('log')
plt.ylim(1e-10, 1e4)
plt.legend()
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_g$')

finalize(results=(d.sigma))
```

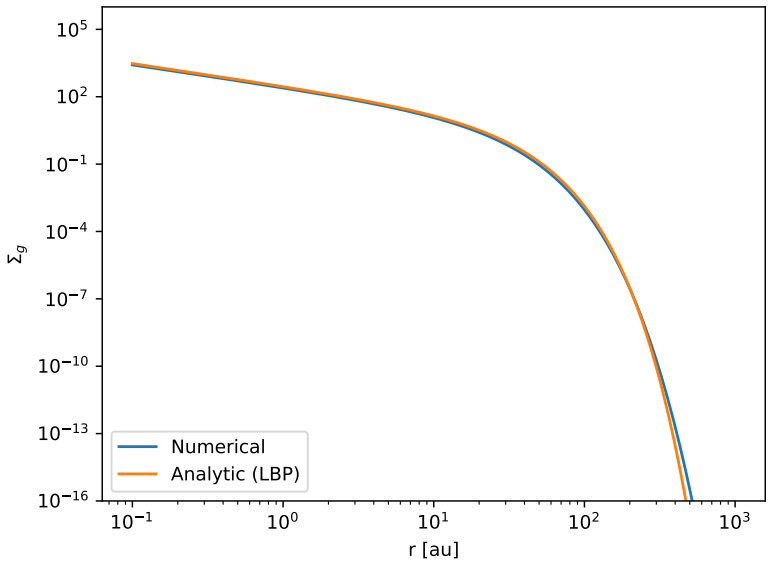


Comparison to Lynden-Bell & Pringle solution. The code snippet is `snippet_viscdiskevol_3.py`. In Python run it as:

```
%run snippet_viscdiskevol_3.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, year, au, finalize

alpha  = 1e-3
mdisk0 = 0.01 * MS
rdisk0 = 1 * au
tstart = 1e3 * year
tend   = 1e6 * year
ntime  = 300
nr     = 1000
time = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))
d = DiskRadialModel(rout=1000 * au, alpha=alpha, nr=nr)
d.make_disk_from_m_pl(mdisk0, rdisk=rdisk0)
for itime in range(1, ntime + 1):
    d.sigma = d.get_viscous_evolution_next_timestep(
        time[itime] - time[itime - 1])
lbp = DiskRadialModel(rout=1000 * au, nr=nr)
lbp.make_disk_from_lbp_alpha(mdisk0, rdisk0, alpha, tend)
plt.figure()
plt.plot(d.r / au, d.sigma, label='Numerical')
plt.plot(lbp.r / au, lbp.sigma, label='Analytic (LBP)')
plt.xscale('log')
plt.yscale('log')
plt.legend(loc='lower left')
plt.ylim(1e-16, 1e6)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_g$')

finalize(results=(d.sigma))
```



It is important to note that `get_viscous_evolution_next_timestep()` only returns the gas surface density $\Sigma_g(r)$. The midplane density is not updated, unless you explicitly do so by calling `d.compute_rhomid_from_sigma()` after te surface density update. Another approach is to use the `compute_viscous_evolution_next_timestep()` method (note: `compute` instead of `get`), which computes the surface density Σ_g and automatically stores it into `d.sigma`, and also recomputes `d.rhomid` from that. Simply replace the time loop in the above snippet(s) by:

```
for itime in range(1,ntime):
   d.compute_viscous_evolution_next_timestep(time[itime]-time[itime-1])
```

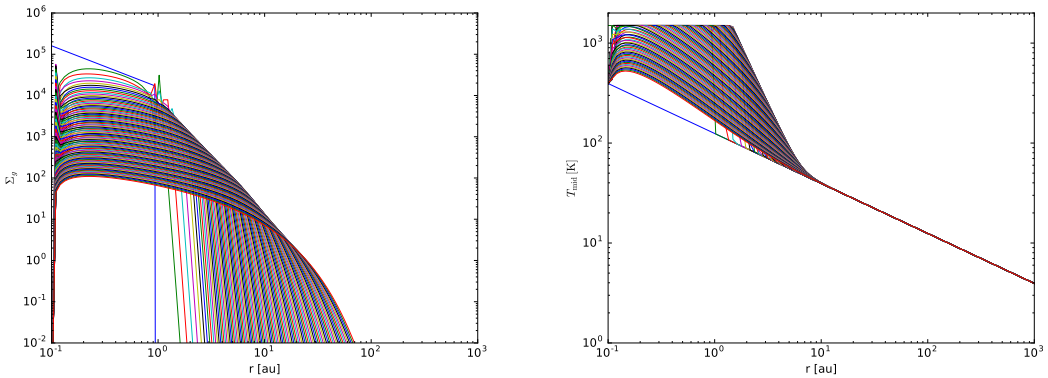
As a slightly more sophisticated example of a viscous accretion model we update the midplane temperature at every time step using `compute_disktmid(vischeat=True)`, i.e. with the viscous heating included. To mimick the thermostat effect due to the dust sublimation we simply limit the temperature by hand to $<$1500 K. We should in principle also check if the disk becomes superadiabatic (in which case convection would set in and limit the temperature further), but in this example we will not do this.

The code snippet is `snippet_viscdiskevol_4.py`. In Python run it as:

```
%run snippet_viscdiskevol_4.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, year, au, finalize

d=DiskRadialModel(rout=1000*au,alpha=1e-3,flang=0.02)
d.make_disk_from_m_pl(mdisk=0.01*MS,rdisk=1*au)
d.add_dust(agrain=1e-4)
kappa=1e3
d.meanopacitymodel=['supersimple',{'dusttogas':0.01,'kappadust':kappa}]
plt.figure(1)
plt.plot(d.r/au,d.sigma)
plt.xscale('log')
plt.yscale('log')
plt.ylim((1e-2,1e6))
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_g$')
plt.figure(2)
plt.plot(d.r/au,d.tmid)
plt.xscale('log')
plt.yscale('log')
plt.ylim((1,2e3))
plt.xlabel('r [au]')
plt.ylabel(r'$T_{\mathrm{mid}}\,[\mathrm{K}]$')
ntime=100
tstart=1e3*year     # Initial time at 1000 year (for log-spaced timesteps)
tend=1e6*year       # Final time at 10 Myr
time=tstart * (tend/tstart)**(np.linspace(0.,1.,ntime+1))   # Logarithmic spacing
for itime in range(1,ntime+1):
   d.compute_mean_opacity()
   d.compute_disktmid(vischeat=True)
   d.tmid[d.tmid>1.5e3]=1.5e3
   d.compute_cs_and_hp()
   d.sigma=d.get_viscous_evolution_next_timestep(time[itime]-time[itime-1],sigma_innerbc=1e-4)
   plt.figure(1)
   plt.plot(d.r/au,d.sigma)
   plt.figure(2)
   plt.plot(d.r/au,d.tmid)

finalize(results=(d.sigma,d.tmid))
```



One can see that initially the temperature near the inner edge saturates at 1500 K. As time progresses the temperature drops. The numerical wiggles early on are because the initial disk was taken to have an abrupt outer edge. As the disk becomes smoother this problem disappears.

Important: In this example we set the inner boundary condition in a different way than before: we fixed the surface density to some arbitrary low value. This is necessary because the default self-adjusting boundary condition causes an instability for this example. The fixed inner boundary is set using the keyword `sigma_innerbc`.

5.2 Computation of $\dot{M}$ and v_r for viscous accretion

Whether or not one uses the time-dependent viscous disk integration capability or not, it can sometimes be useful to compute the accretion rate $\dot{M}(r)$ and/or the radial velocity of the gas $v_r(r)$ of the disk model. In principle one could use the formula Eq. (5.2) for the radial velocity. But sometimes it is important to know the values of $\dot{M}(r)$ and/or $v_r(r)$ exactly, i.e. numerically consistent with the time-dependent evolution from Section 5.1. This is necessary, for instance, when computing the radial transport of dust in the disk: a small numerical mismatch could jeopardize the dust-to-gas ratio.

So in `DiskRadialModel` these values are computed directly from the same algorithm that advances the disk in time. Note that this also works fine if the model is not evolved in time. The computation of $\dot{M}$ and v_r are consistent with, but not dependent on, the time-evolution method.

The method `compute_mdot_at_interfaces()` essentially uses Eq. (5.8), where the derivative is computed numerically exactly in the same way as in the implicit time evolution.

The method `compute_vr_at_interfaces()` first calls the `compute_mdot_at_interfaces()` method to compute $\dot{M}$ at the interfaces. Then it divides by $-2\pi r\Sigma_r$ to obtain v_r. One can choose which value of $-2\pi r\Sigma_r$ to take: the average of the values in the two bordering cells, or the upwind value. This is important if you use v_r elsewhere for e.g. the advection of a dust species: if you use upwinding for that, you should also use the upwind method here (which is, in fact, anyway the default).

Note that both methods return the values *at the interfaces*, not at the cell centers.

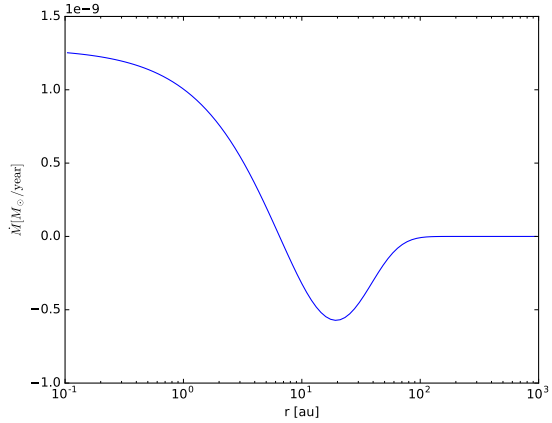
The code snippet is `snippet_compute_vr_mdot_1.py`. In Python run it as:

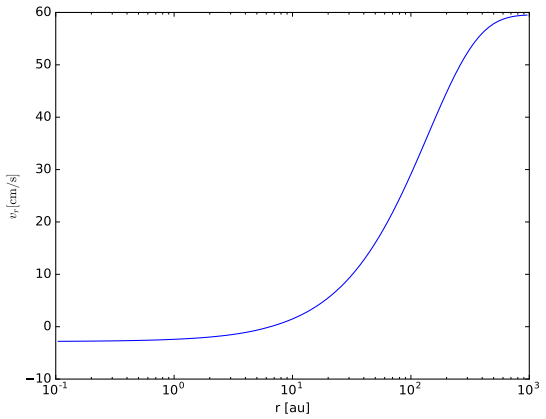
```
%run snippet_compute_vr_mdot_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, au, MS, year, plt, finalize

a = DiskRadialModel(rout=1000 * au)
a.make_disk_from_lbp_alpha(1e-2 * MS, 1 * au, 1e-3, 1e6 * year)
a.compute_mdot_at_interfaces()
a.compute_vr_at_interfaces()

ri = a.get_ri()    # Get the interface radii
```

```
plt.figure()
plt.plot(ri / au, a.mdot / (MS / year))
plt.xscale('log')
plt.xlabel('r [au]')
plt.ylabel(r'$\dot M [M_\odot\mathrm{/year}]$')

plt.figure()
plt.plot(ri / au, a.vr)
plt.xscale('log')
plt.xlabel('r [au]')
plt.ylabel(r'$v_r [\mathrm{cm/s}]$')

finalize(results=(a.mdot,a.vr))
```





You can see that in the inner disk (inward of about 7 au) the gas flows inward and the accretion rate is positive (as expected). Beyond about 7 au the gas flows outward, which is because it absorbs the angular momentum from the inner disk and thus expands the outer disk. The accretion rate is there negative (outflowing). At very large radii the $\dot{M}$ is going back to 0, because there is no material at those radii.

5.3 How to handle gravitational instability

Under certain conditions a protoplanetary disk can become so massive, that the Toomre Q parameter (Section 2.4) becomes smaller than 2. This is in particular to be expected when the disk is still being fed by an infalling envelope (see Section 5.4). If this happens, a gravitational instability (GI) sets in. It is impossible to model the evolution of the disk when such an instability operates. The GI produces non-axially symmetric waves in the disk (spiral waves) that strongly redistribute mass and angular momentum. Since the models in DISKLAB are axially symmetric, a proper treatment of GI is impossible.

It has been suggested in the literature (e.g. Armitage, Livio & Pringle 2001, MNRAS 324, 705) that the GI can be mimicked by an artificially enlarged α-parameter in the region where $Q < 2$. Experiments with this method using DISKLAB have yielded violently unstable solutions for the time step sizes we usually employ.

In DISKLAB we therefore use a different method to mimick the effect of GI. The idea is that we assume that GI will always try to redistribute the mass in the disk in such a way as to keep $Q \geq 2$: i.e. keeping the disk at marginal stability. We assume that this happens on a time scale faster than the usual viscous time scale of the disk, i.e. basically instantly.

The `DiskRadialModel` class of DISKLAB therefore offers a method called `gravinstab_flattening()`. It instantly redistributes the gas surface density in the disk whereever $Q < 2$, such that $Q \geq 2$ is restored everywhere. The $Q < 2$ region thus becomes a $Q = 2$ region. The redistributed mass is dumped on the inside and outside edges of the $Q < 2$ (now $Q = 2$) region in such a way as to increase the local surface density to its maximum, leading to $Q = 2$ there as well. This essentially enlarges the $Q = 2$ region. This redistribution is done in a way as to ensure mass conservation and angular momentum conservation. The enlarging of the $Q = 2$ region on the inside and outside

of the original $Q < 2$ region by the dumping of mass is, under the assumption of mass and angular momentum conservation, uniquely defined.

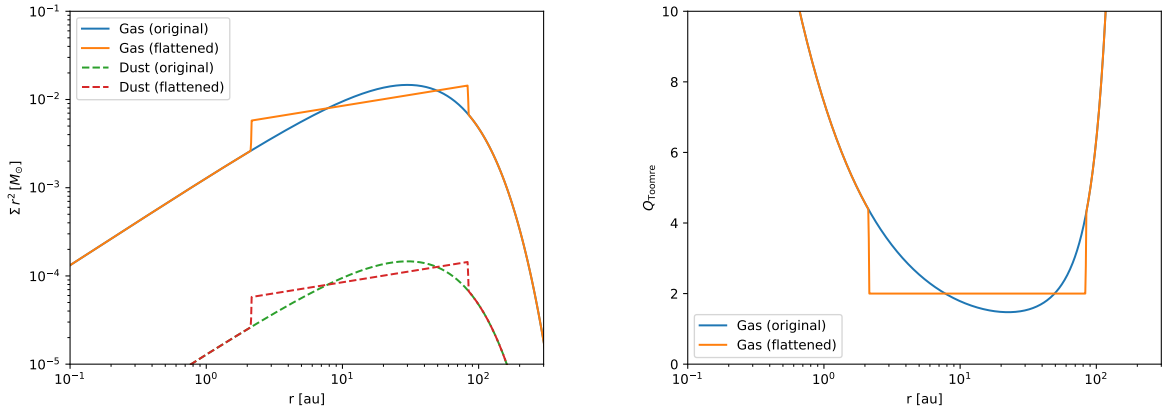
When calling `gravinstab_flattening()` if the disk has $Q > 2$ everywhere, nothing happens. But if there is a region with $Q < 2$, the `gravinstab_flattening()` method will fix the $Q < 2$ issue by generating instead a region with $Q = 2$. This region has an abrupt inner and outer edge, meaning that the surface density will also have a jump at the inner and outer edge of the $Q = 2$ region. This is not a very realistic effect, but in practice even a small α-viscosity will smooth these edges out a bit.

Note that if the disk has a dust component (see Chapter 6), the redistribution of the gas somehow also has to drag the dust along with it. It would be unphysical to have strong redistribution of gas, while leaving the dust untouched. After applying `gravinstab_flattening()` to the gas surface density, you can apply `gravinstab_apply_flattening()` to each dust component, to assure that the dust moves along with the gas during the redistribution.

The idea of using `gravinstab_flattening()` is to apply it (if necessary) at each time step of a viscous evolution model. Whenever the model tends to push Q below 2, the `gravinstab_flattening()` will push it back to 2. In Section 5.4 an example of this is given.

Here we show an extreme example of a disk that is "flattened". The code snippet is `snippet_gravinst_flattening_1.py`. In Python run it as:

```
%run snippet_gravinst_flattening_1.py
```



Please keep in mind that this example is extreme in the sense that it starts with a disk that has a region well below $Q = 2$, leading to a drastic redistribution of mass. If the method `gravinstab_flattening()` is applied every time step, then each such call only redistributes mass moderately.

It can happen that, as a result of the flattening, a lot of mass is collected in the very innermost radial zone. This is, of course, unphysical. The mass at the inner zone should be added to the star. To enforce this, you can set the keyword `flushgridedges=True` in the call to `gravinstab_flattening()`. Note that this option also removes any excessive mass from the outermost zone. But if mass is accumulated in the outermost zone, it is better to extend the radial grid outward.

It should be kept in mind that the instant redistribution of mass by the GI is a pretty strong assumption that may not be correct. If mass loading onto the disk (see Section 5.4) is too excessive, the disk will likely fragment into clumps rather than spread. However, within the context of DISKLAB this is, of course, not possible to model. Also the GI-induced spiral shock waves will heat the disk. This is, so far, not included in the model.

5.4 Viscous disk evolution with Shu-Ulrich infall

A protoplanetary disk does not simply get born in its final state. It gets built-up due to the collapse of a molecular cloud core. This is a complex process that is presumably not well represented by a simple model. But it is nevertheless very instructive to include this build-up phase into the viscous disk evolution.

We follow for most part the paper by Hueso & Guillot (2005) A&A 442, 703, and the implementation by Dullemond,

Natta & Testi 2006 A&A 645, 69 and Dullemond, Apai & Walch 2006 A&A 640, 67. The infall model is that of Shu (1977) ApJ 214, 488, and Ulrich (1976) ApJ 210, 377. These two models are combined where the Shu model treats the radial infall, while the Ulrich model handles the effect of the rotation. Without the rotation the disk would not form, so the Ulrich part is the essential part that creates the disk.

The way we implement the infall is to calculate, at every time step, the $\dot{\Sigma}_g(r,t)$ in Eq. 5.1, which is the infall from the cloud. We do not treat the 3-D effects such as infall onto the outer edge of the disk.

The computation of $\dot{\Sigma}_g(r,t)$ is done by the method `compute_shuulrich_infall()`.

Let us run such a model. We will reproduce the "Example 1" model of Hueso & Guillot (2005), were we get the parameters from their Table 4. The plot of star- and disk-mass as a function of time is their Fig. 5.

The code snippet is `snippet_infall_1.py`. In Python run it as:

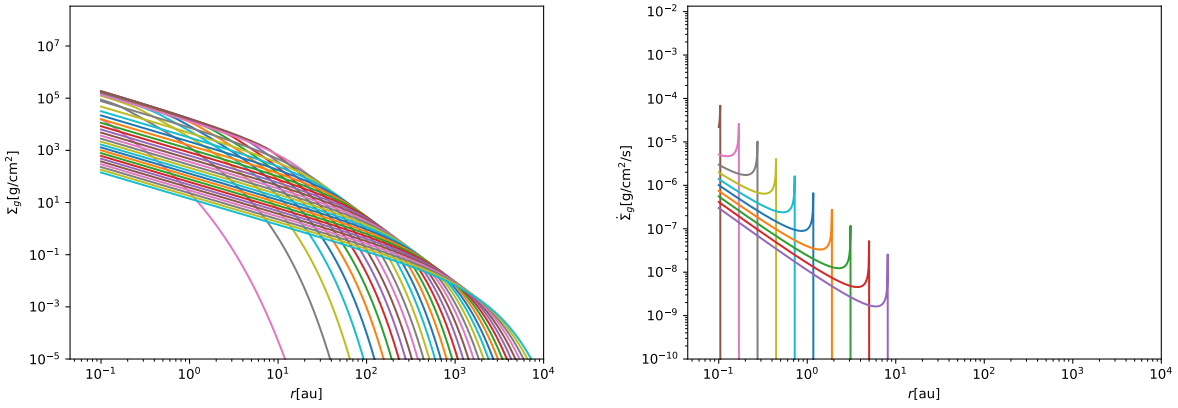
```
%run snippet_infall_1.py
```

Here is the listing:

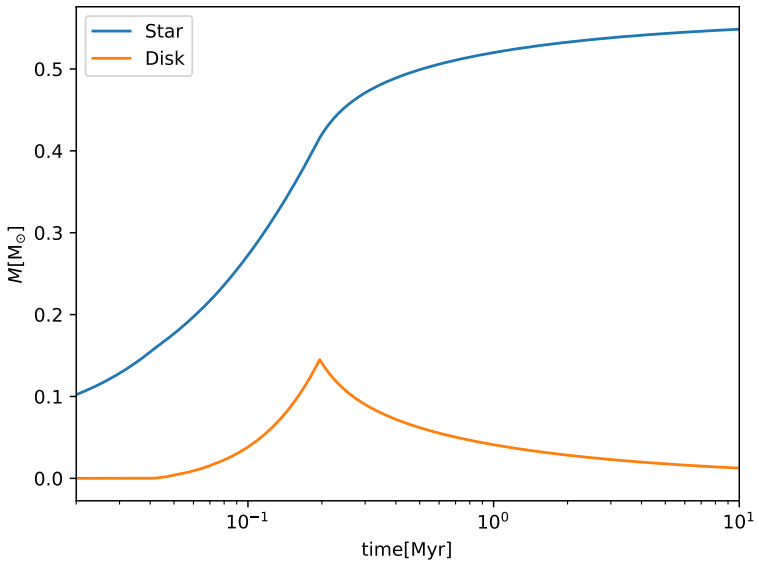
```
from snippet_header import DiskRadialModel, np, plt, MS, year, au, kk, mp, finalize
import copy
rin     = 0.1 * au
alpha   = 1e-2
mdisk0  = 1e-20 * MS
rdisk0  = 1. * au
mcloud  = 0.515 * MS
mstar0  = 1e-4*mcloud
tcloud  = 14.
omcloud = 2.3e-14
tstart  = 1e0 * year
tend    = 1e7 * year
ntime   = 1000
nr      = 1200

cscloud = np.sqrt(kk * tcloud / (2.3 * mp))
time = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))
d = DiskRadialModel(mstar=mstar0, rin=rin, rout=1e6 * au, alpha=alpha, nr=nr)
d.make_disk_from_lbp_alpha(mdisk0, rdisk0, alpha, tstart)
dlist = [copy.deepcopy(d)]
mdisk = np.zeros(ntime + 1)
mstar = np.zeros(ntime + 1)
rcentr = np.zeros(ntime + 1)
mdot = np.zeros(ntime + 1)
mdotdsk = np.zeros(ntime + 1)
mdotcap = np.zeros(ntime + 1)
mdisk[0] = mdisk0
mstar[0] = mstar0
rcentr[0] = 0.
mdot[0] = 0.
mdotdsk[0] = 0.
mdotcap[0] = 0.
for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.compute_shuulrich_infall(mcloud, cscloud, omcloud, time[itime], idistr=1)
    d.sigdot = d.infall_sigdot
    d.compute_viscous_evolution_next_timestep(dt)
    d.compute_mdot_at_interfaces()
    mdt = d.infall_mdotcap + d.mdot[0]  # Accretion rate onto star
    mst = d.mstar + mdt * dt
    d.update_stellar_mass(mst)
    d.compute_mass()
    dlist.append(copy.deepcopy(d))
    mstar[itime] = d.mstar
    mdisk[itime] = d.mass
```

```
    rcentr[itime] = d.infall_rcentr
    mdot[itime] = mdt
    mdotdsk[itime] = d.mdot[0]
    mdotcap[itime] = d.infall_mdotcap

print("Mass conservation: (Mstar+Mdisk)/(Mcloud+Mstar0+Mdisk0) = {}".format(
    (d.mstar + d.mass) / (mcloud + mstar0 + mdisk0)))

plt.figure()
for itime in range(0, ntime, 10):
    plt.plot(dlist[itime].r / au, dlist[itime].sigma)
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$\Sigma_g [\mathrm{g/cm^2}]$')
plt.xscale('log')
plt.yscale('log')
plt.xlim(right=1e4)
plt.ylim(bottom=1e-5)


plt.figure()
for itime in range(10, ntime, 10):
    plt.plot(dlist[itime].r / au, dlist[itime].infall_sigdot + 1e-50)
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$\dot\Sigma_g [\mathrm{g/cm^2/s}]$')
plt.xscale('log')
plt.yscale('log')
plt.xlim(right=1e4)
plt.ylim(bottom=1e-10)

plt.figure()
plt.plot(time / year / 1e6, mstar / MS, label='Star')
plt.plot(time / year / 1e6, mdisk / MS, label='Disk')
plt.xlabel(r'$\mathrm{time} [\mathrm{Myr}]$')
plt.ylabel(r'$M [\mathrm{M}_{\odot}]$')
plt.xscale('log')
plt.xlim(left=0.02, right=10.)
plt.legend(loc='upper left')

finalize(results=(dlist[-1].sigma,mstar,mdisk))
```

Note that this listing has a lot of additional infrastructure to make useful plots. The actual "bare" code that performs the Hueso&Guillot-like infall and viscous evolution is only the part inside the time loop.



Left is the surface density profile $\Sigma_g(r,t)$ for various time snapshots. Right the $\dot\Sigma_g(r,t)$.

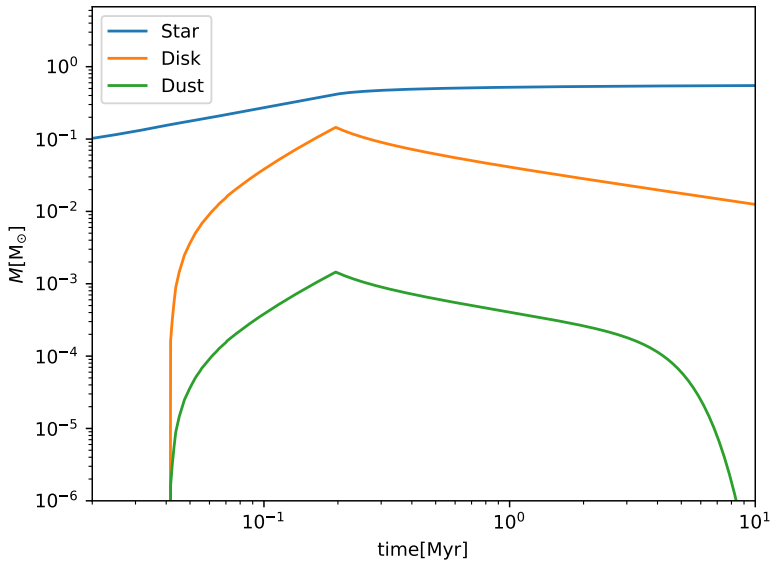


The time evolution of stellar and disk mass. Comparing this plot to Hueso & Guillot's Fig. 5 we see that the match is reasonable. There are some slight differences, but they are minor. For instance, in their plot the disk only starts gaining mass after 0.09 Myr, while in our case this happens already at 0.04 Myr because that is about the time when the centrifugal radius of the infalling material reaches the inner edge of the grid at 0.1 au. We find that mass conservation (where the final stellar + disk mass equals the initial stellar + disk + cloud mass) requires that the first time steps are fairly small. But since we use, as usual, logarithmically spaced time steps, that is not a problem. We find 99% mass conservation.

You can also add a dust component to the infalling gas. A simple example is to have only 1 grain size of 10 μm radius with a dust-to-gas ratio of 0.01. The code snippet is `snippet_infall_2.py`. In Python run it as:

```
%run snippet_infall_2.py
```

The star, disk and dust mass as a function of time is shown here:



The decline of the dust compared to the gas for large times is due to the dust radial drift.

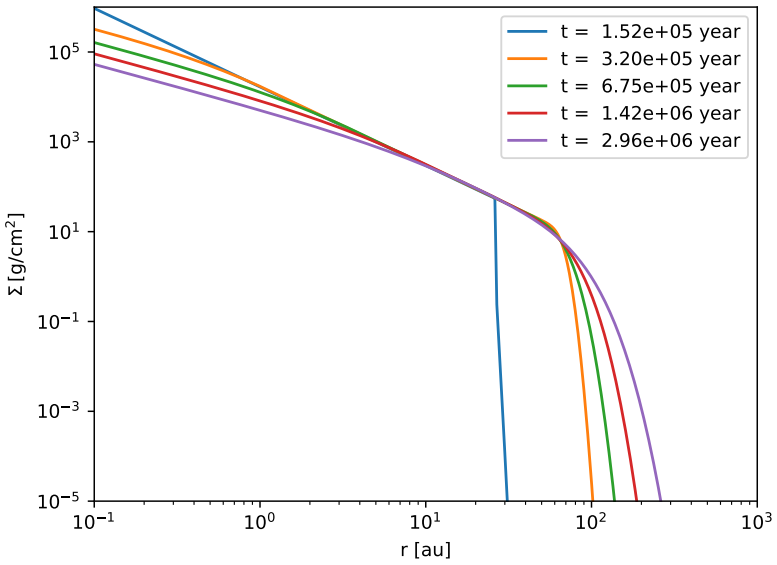
If we have strong infall and yet low α turbulence, the disk tends to become so dense, that the Toomre Q value drops below 2. Physically this means that the disk becomes gravitationally unstable, leading to spiral waves, which

redistribute angular momentum and matter. The normal viscous disk evolution equations do not handle this properly. In Section 5.3 we discuss a simplified method to deal with this. If we apply this at every time step during the infall phase, we can prevent unphysically dense disks from emerging. In other words: we mimic the mass-spreading effect of the gravitational instability to keep $Q \geq 2$ everywhere.

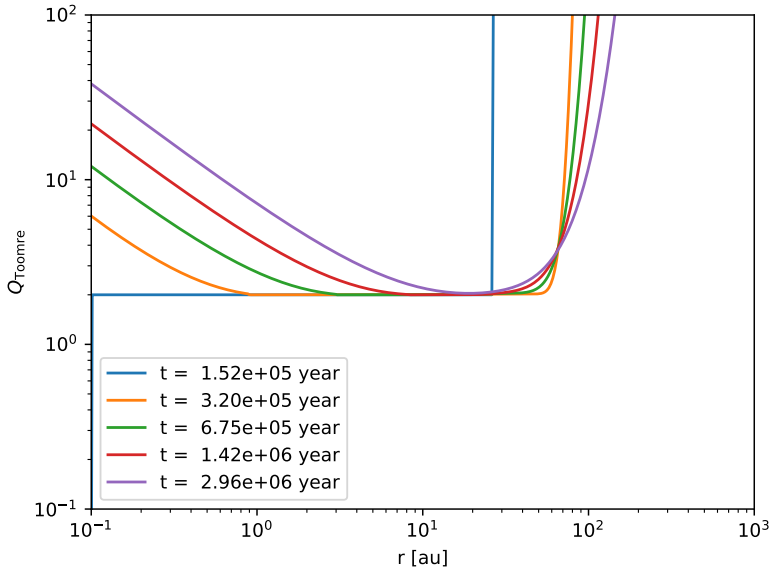
The code snippet is `snippet_infall_3.py`. In Python run it as:

```
%run snippet_infall_3.py
```

The evolution of the surface density is shown here:



and the evolution of the Toomre Q parameter here:



One sees that the gravitational instability treatment of Section 5.3, applied here, keeps the Toomre Q above 2, and the disk (marginally) stable.

Finally, code snippet `snippet_infall_4.py` demonstrates that it can save computational time to start the model only once the centrifugal radius of the infall reaches the inner edge of the radial grid. This is because at earlier times there is anyway no disk yet. The initial stellar mass is then simply the infall rate times the starting time.

5.5 Numerical stability with viscous heating and non-linear opacity model

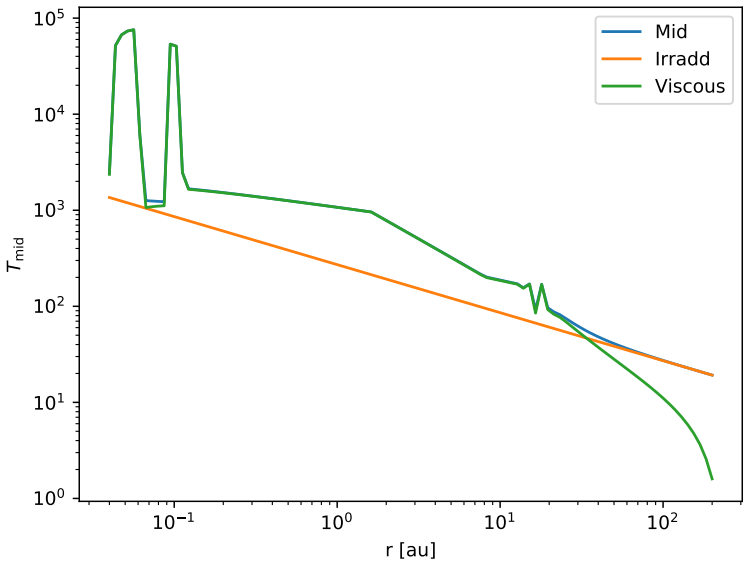
The time-dependent integration of the viscous disk equations in DISKLAB is performed using implicit integration (see Section A). However, the implicit integration method is only applied *partially*: It is applied to $\Sigma(r)$, keeping everything else fixed. Under most circumstances this allows us to make large time steps without risking numerical instability.

However, under special circumstances this approach can fail. It is known to fail, for instance, when we include viscous heating (for the calculation of `tmid`) using a mean opacity model that includes dust evaporation "jumps". The Bell & Lin opacity model (Section 8.5) includes this, for instance. For increasing temperature the opacity makes sudden jumps downward.

An example of how this fails is given in the following snippet: `snippet_viscevol_instability_1.py`. In Python run it as:

```
%run snippet_viscevol_instability_1.py
```

The evolution of the surface density is shown here:

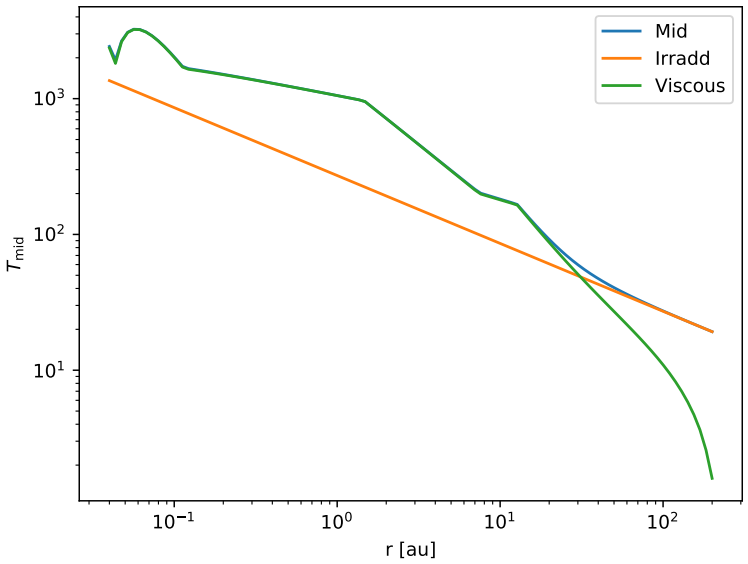


One can clearly see that the solution is wrong: it contains dramatic wiggles. In fact, at each time step, the peaks flip back and forth. In fact, also the surface density Σ flips back and forth between two limits. The product of Σ and $T_{\rm mid}$, however, stays well-behaved, because $\dot{M} \propto \Sigma T_{\rm mid}$, and the implicit viscous disk evolution algorithm tries to keep the accretion well-behaved.

The solution to this flip-flop instability is to vary $\Sigma \propto 1/T_{\rm mid}$ while solving for $T_{\rm mid}$. This can be done by setting `fixmdot=True` in the call to `compute_disktmid()`. The snippet is: `snippet_viscevol_instability_2.py`. In Python run it as:

```
%run snippet_viscevol_instability_2.py
```

The evolution of the surface density is shown here:



As one can see: now everything remains well-behaved.

5.6 For convenience: the `DiskRadialModel.anim()` method

Sometimes it is nice to see how the disk (or some quantity of the disk) evolves time-dependently, in an animation. The `DiskRadialModel.anim()` method provides a simple way to do this.

Here is an example snippet: `snippet_anim_diskmodel_1.py`. In Python run it as:

```
%run snippet_anim_diskmodel_1.py
```

Here is the listing:

```
from snippet_header import np, DiskRadialModel, au, year, finalize

d = DiskRadialModel(rout=1000 * au)
d.make_disk_from_simplified_lbp(1e2, 1 * au, 1.0)
tmax = 1e6 * year
nt = 100
time = np.linspace(0, tmax, nt)
sigma_array = np.zeros((nt, len(d.r)))
sigma_array[0, :] = d.sigma.copy()
for itime in range(1, nt):
    dt = time[itime] - time[itime - 1]
    d.compute_viscous_evolution_next_timestep(dt)
    sigma_array[itime, :] = d.sigma.copy()

d.anim(time, sigma_array, ymin=1e-5, pause=30)

finalize([])
```

This should show a simple viscously spreading disk. The time snapshots are chosen linear in time in this example.

But you can also choose a logarithmic time spacing. And you can animate any other quantity, for instance the dust surface density (see Chapter 6). As a preview of this, you can enjoy the following animation of dust drift in an evolving disk with two planetary gaps: `snippet_anim_diskmodel_2.py`. In Python run it as:

```
%run snippet_anim_diskmodel_2.py
```

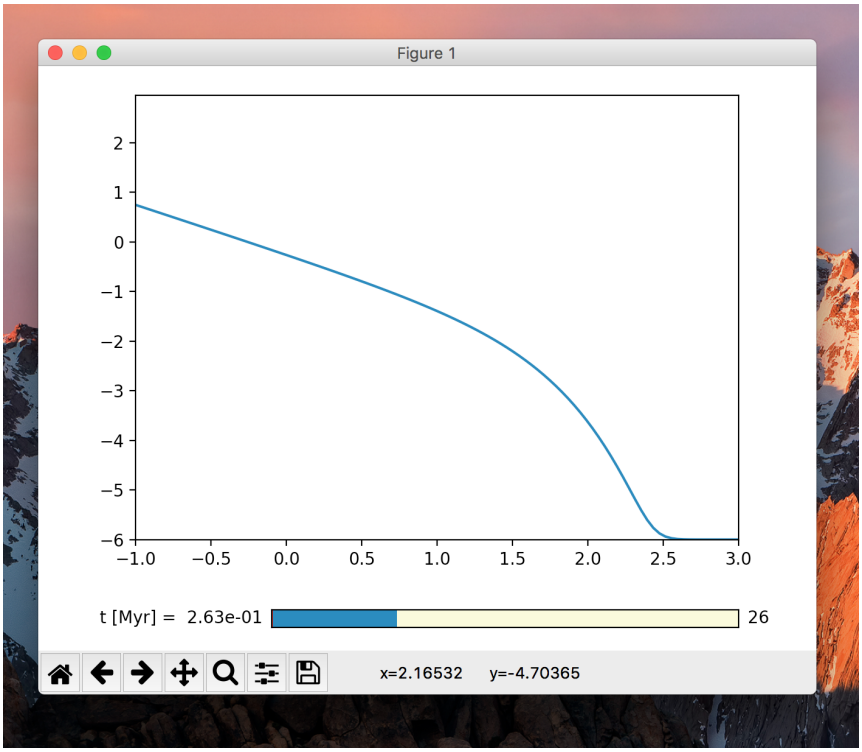
5.7 For convenience: the `viewarr.py` tool

Another way to animate the time-evolution of a model is to use the `viewarr.py` tool, included in DISKLAB. This tool makes use of the `interactive_plot.py` tool, also available in DISKLAB.

`viewarr.py` is a general-purpose tool to plot 1-D cuts from an n-dimensional array. By storing the intermediate time snapshots into a 2-D array, you can view the time-evolution using a slider.

Here is an example snippet: `snippet_anim_diskmodel_3.py`. In Python run it as:

```
%run snippet_anim_diskmodel_3.py
```



Chapter 6

Adding dust components to the 1-D radial disk model

An essential feature of DISKLAB is to include the dynamics of the dust particles. One can include any number of different kinds of dust particles, where each kind is a disk component, represented by an object of the class `DiskRadialComponent`.

A disk component can be regarded as a gas or dust species that is part of the disk model. Mostly we will be concerned with *dust* components, which can have a different velocity and mixing characteristics from the main gas surface density of the disk (dust drift, for example). But a disk component can also be a *gas* component, for instance the vapor of sublimated ice grains. Both dust components and gas components are described as a *disk* component, and are described by objects of the class `DiskRadialComponent`.

For the remainder of this chapter, we will assume that the disk components we add are, in fact, dust components.

6.1 The `DiskRadialComponent` class

A dust component is added to the disk model with the `add_dust()` method. This will create a dust object from the `DiskRadialComponent` class, and it will add this to a list with the name `dust`:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
d.add_dust(agrain=1e-4,xigrain=3.6,dtg=0.1)
d.dust
Out[]: [<disklab.diskradial.DiskRadialComponent at 0x10c02f690>]
```

The `DiskRadialComponent` class is a class like `DiskRadialModel` but containing not the information about the gas, but instead information about the dust (or another type of disk component such as a chemical species).

Note that d.dust is a *list* of dust components. We can add several dust components:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
d.add_dust(agrain=1e-4,xigrain=3.6,dtg=0.1)
d.add_dust(agrain=1e-3,xigrain=3.6,dtg=0.1)
d.add_dust(agrain=1e-2,xigrain=3.6,dtg=0.1)
d.dust
Out[]:
[<disklab.diskradial.DiskRadialComponent at 0x10a82eb90>,
 <disklab.diskradial.DiskRadialComponent at 0x1134d67d0>,
 <disklab.diskradial.DiskRadialComponent at 0x1134d6510>]
```

And like any Python list, we can remove items from the list, e.g.

```
d.dust.pop()
Out[]: <disklab.diskradial.DiskRadialComponent at 0x1134d6510>
d.dust
Out[]:
[<disklab.diskradial.DiskRadialComponent at 0x10a82eb90>,
 <disklab.diskradial.DiskRadialComponent at 0x1134d67d0>]
d.dust.pop(0)
Out[]: <disklab.diskradial.DiskRadialComponent at 0x10a82eb90>
```

The `add_dust()` method is just a shorthand for the following set of commands:

```
dust = DiskRadialComponent(d,agrain=1e-4,xigrain=3.6,sigma=d.sigma*0.1)
d.dust.append(dust)
```

What happens here is that we first create an instance of the class `DiskRadialComponent` and call it `dust`. As you can see, the first argument is `d` (the disk model): any disk component must know about its 'parent', which is the disk model. This is stored in `dust.diskradialmodel`. But the disk component does not necessarily know about all its dust components. Only if you append `dust` to the list `d.dust` will the disk model `d` 'know' about this dust component.

Note that each dust component is just a single dust component, i.e. a single grain size or single Stokes number. If you want to include a *grain size distribution*, then you simply have to add, say, 10 dust components of increasing sizes, which together represent a 10-point sampling of the size distribution (see Section 6.7). For now we will focus on a single species.

For each single dust component it holds that at any given radius the dust grains have only one size a in units of centimeter (given as `agrain`) with material density ξ in units of gram/cm^3 (given as `xigrain`). The grain mass m is then computes as `mgrain` assuming a homogeneous sphere:

$$m = \frac{4\pi}{3}\xi a^3 \tag{6.1}$$

There is, however, an exception to this: if you specify the *Stokes number* instead of the grain size, then the grain size will vary with r (see below).

The computation of the *properties* of the dust grains are handled by a separate class: the `GrainModel` class (see Chapter 7). This class contains all the methods for computing various properties of the dust and how the grains interact with the gas. However, the surface density of the dust `sigma` is an attribute of the `DiskRadialComponent` class (Section 6.1), not of the `GrainModel` class (Chapter 7). The `GrainModel` class is therefore meant only for computing dust grain properties, and at any given time contains only one single grain with one set of properties. In other words: The `GrainModel` class is meant only for properies of an *individual* dust grain, while the `DiskRadialComponent` class describes the *spatial and temporal distribution* of the dust.

When adding a dust component with the `add_dust()` method, one can *either* specify the Stokes number (and the grain size will then be calculated at each radius) *or* one specifies the grain size (and the Stokes number is then calculated at each radius). One can also specify the material density ξ and the dust-to-gas ratio.

Example:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
d.add_dust(agrain=1e-4,xigrain=3.6,dtg=0.1)
```

This adds a dust component to the disk consisting of $a = 1\mu m$ radius grains with a material density of $\xi = 3.6\,\mathrm{g/cm^3}$. The dust is added at a dust-to-gas ratio of 0.1. The Stokes number at each radius is given by `d.St`. One can also specify `St`:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
d.add_dust(St=1e-2,xigrain=3.6,dtg=0.1)
```

You can inspect the results:

```
d.dust[0].agrain    # Grain radius in cm
d.dust[0].mgrain    # Grain mass in gram
d.dust[0].tstop     # Stopping time in seconds (at the midplane of the disk)
d.dust[0].St        # Stokes number (at the midplane of the disk)
```

Note: The `[0]` means: the first dust component. In the above example we added just one component, hence only `[0]`. If you add another dust component, you can also inspect `d.dust[1]`.

6.2 Frictional stopping time and the Stokes number

The dynamics of dust grains in a protoplanetary disk is entirely goverened by the *stopping time* of the grains. In `DiskRadialModel` the stopping time is automatically computed when the `add_dust()` method is called. It is, for each disk component `dc`, stored in `dc.tstop`.

But if the disk midplane temperature `d.tmid` or the disk gas surface density `d.sigma` change, one has to recompute the stopping time in order to obtain the correct drift and mixing behavior. This can be done with

```
for dust in d.dust:
    dust.compute_stokes_from_agrain()
```

The method `compute_stokes_from_agrain()`, in fact, uses the methods of the `GrainModel` class to compute the stopping time. For more details on the *stopping time* and the *Stokes number*, and how they are computed, please see Section 7.2.

Given the Stokes number St, the dynamical behavior of the dust particle in the disk can be calculated. In `DiskRadialModel` we always compute St for the radial drift and mixing from the *midplane* gas density and temperature.

In principle the Stokes number is automatically computed by `add_dust()` if `agrain` is specified, and vice versa. But sometimes it can happen that you do not want to re-call `add_dust()` but you want to re-compute the Stokes number and stopping time (because, e.g., the disk temperature has changed). You can do that with `compute_stokes_from_agrain()`. The reverse method is `compute_agrain_from_stokes()`.

Warning: For large particles and high gas density, the gas drag formulae will be in the *Stokes regime*. In that case, the computation of the stopping time will depend on the relative velocity between the dust and the gas $|v_{\rm dust} - v_{\rm gas}|$. You must then specify this relative velocity in your call to `dust.compute_stokes_from_agrain()`. There is a numerical subtlety here: in `DiskRadialModel` and `DiskRadialComponent` the radial velocity is specified at the cell interfaces, not at the cell centers. To not complicate matters too much, we can simply use those (shifting half a cell):

```
for dust in d.dust:
    dust.compute_stokes_from_agrain(dv=np.hstack((np.abs(d.vr-dust.vr),0.)))
```

And you may need to iterate this with the computation of the radial dust drift velocity (see Section 6.3). So indeed, for large particles (larger than the molecular mean-free path of the gas), things become a bit tricky. **Designing a convenient automatic iteration procedure for this is on our to-do-list (2018.04.14).**

6.3 Computation of radial dust velocity $v_{\rm dust}$

The method `compute_dustvr_at_interfaces()` computes the dust radial drift velocity $v_{\rm dust}$. The method first computes the radial velocity of the gas v_r (see Subsection 5.2), which is important, because the gas flow can drag the dust along with it. It relies on the already-computed Stokes number of the dust (`self.St`, computed as described in Section 6.1). We follow here again Birnstiel, Dullemond & Brauer (2010) A&A 513, 79. Accordingly:

$$v_{\rm dust} = \frac{1}{1+{\rm St}^2}\left(v_r + {\rm St}\frac{c_s^2}{\Omega_K r}\frac{d\ln p}{d\ln r}\right) \tag{6.2}$$

where p is the gas pressure at the midplane $p = \rho c_s^2$, and $d\ln p/d\ln r$ is the double-logarithmic derivative of the gas pressure, which is computed in the method `compute_omega()` (see Section 2.5). Note that for $\mathrm{St} \to 0$ we obtain $v_{\rm dust} \to v_r$, i.e. the dust is then perfectly coupled to the gas, moving along with it. Conversely, for $\mathrm{St} \to \infty$ we obtain $v_{\rm dust} \to 0$.

6.4 Time-dependent radial dust drift and mixing

The method `get_dust_radial_drift_next_timestep()` lets the dust radially drift and mix one time step. Also this method uses implicit integration and is thus stable even for large time steps. But as with the disk evolution: large time steps may lead to unreliable results.

We always use the midplane gas density, temperature and pressure to compute the behavior of the dust. See 6.1. We follow Birnstiel, Dullemond & Brauer (2010) A&A 513, 79. The radial drift/mixing equation for the dust is:

$$\frac{\partial \Sigma_d}{\partial t} + \frac{1}{r}\frac{\partial (r\Sigma_d v_d)}{\partial r} - \frac{1}{r}\frac{\partial}{\partial r}\left(r D_d \Sigma_g \frac{\partial}{\partial r}\left(\frac{\Sigma_d}{\Sigma_g}\right)\right) = \dot{\Sigma}_d \tag{6.3}$$

where v_d is the radial dust velocity given by Eq. (6.2), and the diffusion coefficient D_d is

$$D_d = \frac{1}{\mathrm{Sc}}\frac{1}{1+\mathrm{St}^2}\nu \tag{6.4}$$

where Sc is the Schmidt number of the gas defined as the ratio of gas turbulent viscosity ν over gas turbulent diffusivity D_g:

$$\mathrm{Sc} = \frac{\nu}{D_g} \tag{6.5}$$

The St is the Stokes number of the particles given by Eq. (7.7) and ν is the turbulent viscosity given by Eq. (3.7). If the source term $\dot{\Sigma}_d$ is positive, then this means that dust is added to the disk. Typically this would simply accompany infall of gas. One would then expect $\dot{\Sigma}_d/\dot{\Sigma}_g$ to represent the dust-to-gas ratio of the infalling matter.

IMPORTANT NOTE: If you use this at the same time as the time-dependent evolution of the gas surface density (see the `get_viscous_evolution_next_timestep` method of `DiskRadialModel` described in Subsection 5.1), then you have to keep in mind that small enough time steps have to be made to keep the radial dust drift velocity consistent with the changing gas. Also note that for a given grain size the Stokes number changes as the disk evolves.

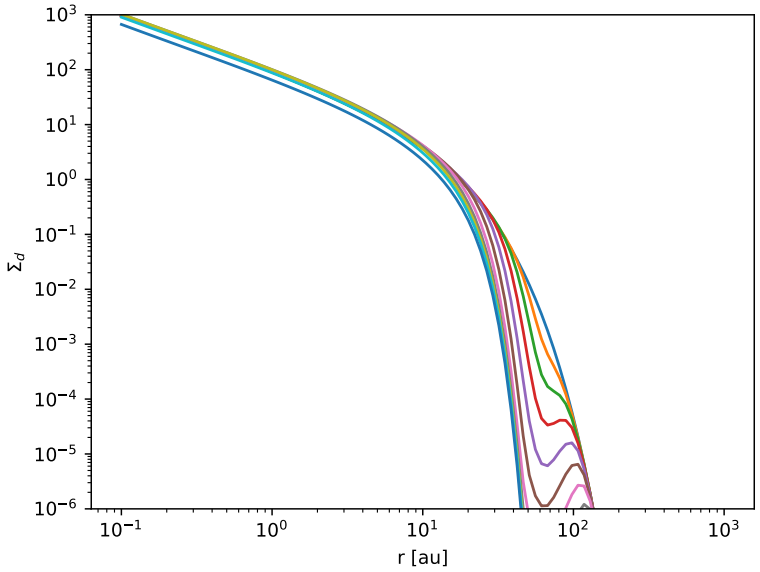
Example: a fixed gas disk with drifting dust. The code snippet is `snippet_dustdrift_1.py`. In Python run it as:

```
%run snippet_dustdrift_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, year, au, finalize
tstart = 1e3 * year
tend   = 3e6 * year
ntime  = 10
nr     = 100
time   = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))

d = DiskRadialModel(rout=1000 * au, nr=nr)
d.make_disk_from_simplified_lbp(1e3, 10 * au, 1)
d.add_dust(agrain=1e-1)
# d.Sc = 1e10    # Switch off mixing by putting Schmidt number to 'infinity'

# plotting

plt.plot(d.r / au, d.dust[0].sigma)
for itime in range(1, ntime + 1):
    d.dust[0].sigma = d.dust[0].get_dust_radial_drift_next_timestep(time[itime] - time[itime - 1], fi
    plt.plot(d.r / au, d.dust[0].sigma)
    d.dust[0].compute_mass()
```

```
    print('{}'.format(d.dust[0].mass / MS))
plt.xscale('log')
plt.yscale('log')
plt.ylim(1e-6, 1e3)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_d$')

finalize(results=(d.dust[0].sigma))
```



Here we gave the argument `fixgas=True`, which means that the radial gas velocity is not included in the dust drift: the gas is assumed to stand still. This is useful for this example, because we do not viscously evolve the gas disk here. If we would not include `fixgas=True`, then we would get an inconsistent result: the gas is not moving, but the dust "thinks" that the gas is moving. By setting `fixgas=True` the gas is kept fixed ($v_r = 0$) (even if `d.vr`$\neq 0$). The print statement gives the dust mass at each of the time steps. One sees that the dust mass decreases. That is because radial turbulent mixing transports dust inward where it is lost (while the gas is kept fixed). If you uncomment the line with `d.Sc=1e10` you switch off the radial mixing. You will then see no decrease of the dust mass.

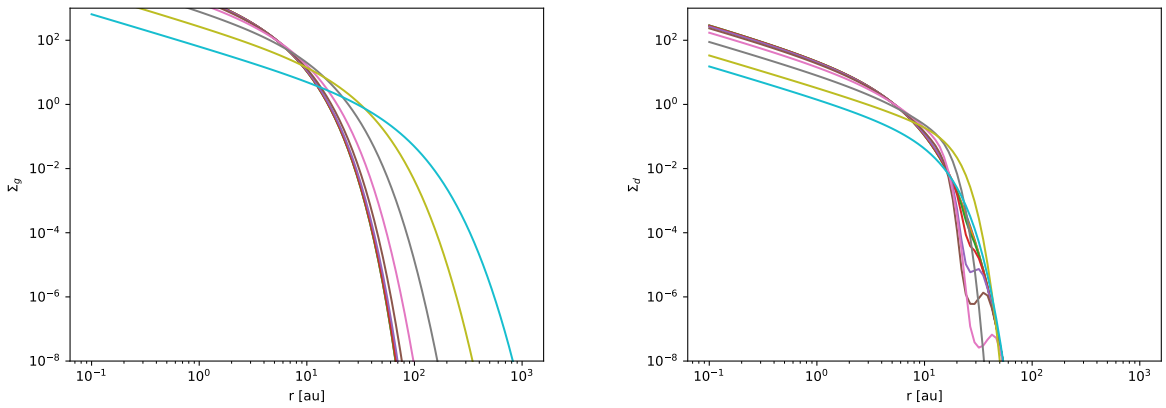
Now let us do a simultaneous viscous disk evolution and dust drift/mixing model. The code snippet is `snippet_dustdrift_2.py`. In Python run it as:

```
%run snippet_dustdrift_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, year, au, finalize
import copy
agrain = 1e-2  # Middle-large grains
tstart = 1e1 * year
tend   = 1e6 * year
ntime  = 100
nr     = 100
rdisk0 = 3 * au
sig0   = 1e3
time   = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))

# setup

d = DiskRadialModel(rout=1000 * au, nr=nr)
```

```
d.make_disk_from_simplified_lbp(sig0, rdisk0, 1)
d.add_dust(agrain=agrain)

# iteration

dlist = [copy.deepcopy(d)]
for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.sigma = d.get_viscous_evolution_next_timestep(dt)
    d.compute_rhomid_from_sigma()
    d.dust[0].compute_stokes_from_agrain()
    d.dust[0].sigma = d.dust[0].get_dust_radial_drift_next_timestep(dt)
    # d.compute_disktmid()  # Uncomment if Tmid depends on d.sigdust
    dlist.append(copy.deepcopy(d))

# plotting

plt.figure()
for itime in range(0, ntime, 10):
    plt.plot(dlist[itime].r / au, dlist[itime].sigma)
plt.xscale('log')
plt.yscale('log')
plt.ylim(1e-8, 1e3)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_g$')


plt.figure()
for itime in range(0, ntime, 10):
    plt.plot(dlist[itime].r / au, dlist[itime].dust[0].sigma)
plt.xscale('log')
plt.yscale('log')
plt.ylim(1e-8, 1e3)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_d$')

finalize(results=(dlist[-1].sigma,dlist[-1].dust[0].sigma))
```



Left is the gas evolution, right the dust evolution. The commented-out command to recompute the disk midplane temperature is here only to keep in mind that if you improve the model to include the viscous energy dissipation (which depends on `d.sigma` and `d.dust.sigma`) or if the irradiation flux depends on the `d.dust.sigma`, you would have to recompute the midplane temperature. In this simple example this is not the case, so it is commented-out.

The `DiskRadialModel` also has shortcuts for the stuff inside the time loop. You can, for instance, replace the time

loop in the above code with:

```
for itime in range(1,ntime+1):
   dt = time[itime]-time[itime-1]
   d.compute_viscous_evolution_next_timestep(dt)
   d.dust[0].compute_dust_radial_drift_next_timestep(dt)
   dlist.append(copy.deepcopy(d))
```

Or even shorter (by combining the viscous and dust steps):

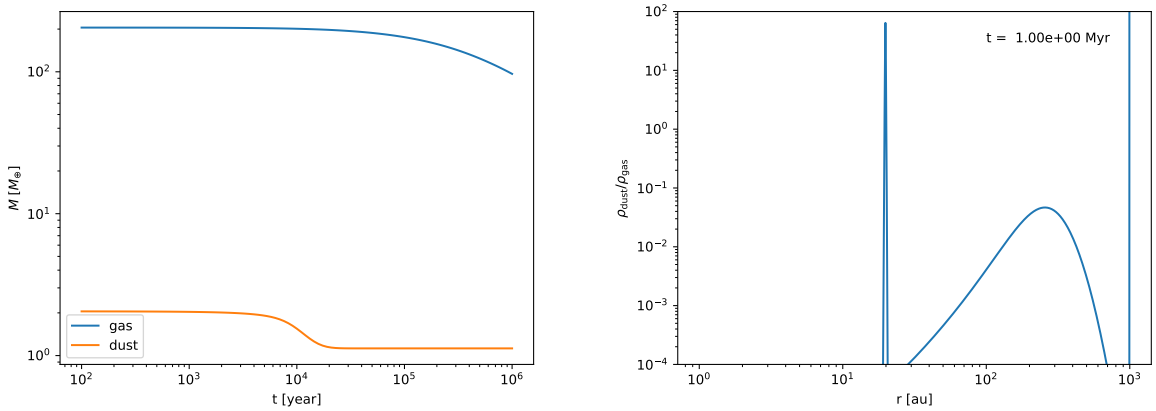
```
for itime in range(1,ntime+1):
   dt = time[itime]-time[itime-1]
   d.compute_viscous_evolution_and_dust_drift_next_timestep(dt)
   dlist.append(copy.deepcopy(d))
```

Here follows an example where the total dust and gas mass as a function of time is shown in a plot, and where the dust surface density is animated as it evolves with time.

The code snippet is `snippet_dustdrift_3.py`. In Python run it as:

```
%run snippet_dustdrift_3.py
```

In the animation you can see in pseudo-real-time how the dust drifts and gets stuck in a pressure bump (the pressure bump is here "installed" by hand in an ad-hoc way). The total dust mass declines but levels off at some non-zero value. This is the effect of the dust trapping in the pressure bump. Note that the trapping is so efficient that at the end of the simulation the dust-to-gas ratio at the disk midplane is much larger than unity. In a real situation this very likely would lead to the onset of the streaming instability.



6.5 Steady-state radial dust drift and mixing solution

Radial drift of dust is often rather rapid. We may therefore assume that if there is dust trapping in pressure maxima, the dust is in steady state. The main parameter is then the dust accretion rate. The `get_drift_diffusion_solution()` method computes this steady-state for a given dust accretion rate.

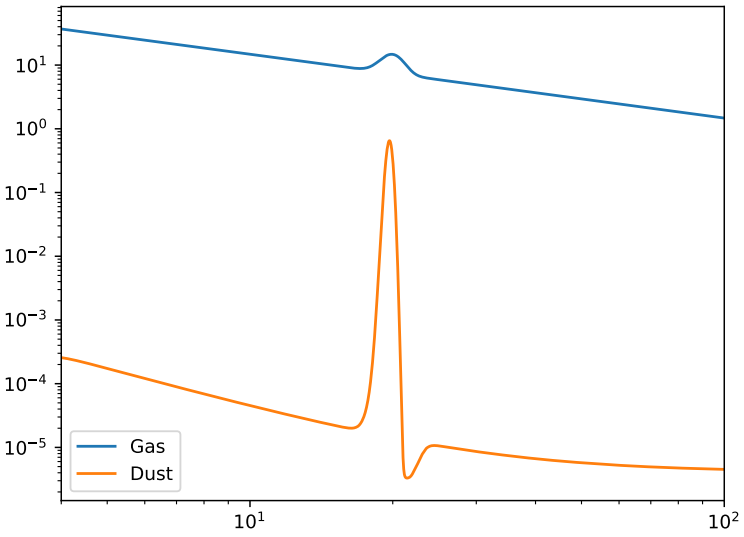
Example of a disk with a bump. The code snippet is `snippet_dustdrift_stationary_1.py`. In Python run it as:

```
%run snippet_dustdrift_stationary_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, year, au, finalize

d = DiskRadialModel(rin=4 * au, rout=100 * au, nr=1000, alpha=2e-2)
```

```
d.make_disk_from_m_pl(1e-2 * MS, plsig=-1)
d.add_dust(agrain=1e0)

rbump     = 20. * au                      # Radial location of bump
abump     = 1.0                           # Amplitude of bump (relative)
hpbump    = np.interp(rbump, d.r, d.hp)   # Pressure scale height
wbump     = hpbump                        # Width (stand dev) of Gaussian bump
fact      = 1 + abump * np.exp(-0.5 * ((d.r - rbump) / wbump)**2)
d.sigma  *= fact
d.rhomid *= fact
mdotdust  = 1e-2 * 1e-10 * MS / year
d.dust[0].sigma = d.dust[0].get_drift_diffusion_solution(mdotdust)

plt.plot(d.r / au, d.sigma, label='Gas')
plt.plot(d.r / au, d.dust[0].sigma, label='Dust')
plt.xscale('log')
plt.yscale('log')
plt.xlim(d.r[0] / au, d.r[-1] / au)
plt.legend(loc='lower left')

finalize(results=(d.dust[0].sigma))
```



Note how sensitive the result is to the viscous alpha parameter. If you plot it with only two times weaker alpha you will find a dramatically stronger peak. This is logical because the dust can only escape the dust trap if there is sufficiently strong mixing. Since the dust accretion rate is fixed, the peak will then change for changing turbulence. Note also the dip before the bump: this is because the dust will get accelerated as it enters the dust trap.

6.6 Testing dust trapping against an analytic solution

For very idealized situations one can compute an analytic steady-state solution for the dust trapping scenario. We consider a narrow gas ring around the star at radius r_0 with a midplane pressure given by

$$p(r) = p_0 \exp\left(-\frac{(r-r_0)^2}{2\sigma^2}\right) \tag{6.6}$$

where $\sigma \ll r_0$ is the parameter setting the width of this gaussian ring. One can derive that the steady state dust distribution is then:

$$\Sigma_{\rm d}(r) = \Sigma_{\rm d0} \exp\left(-\frac{(r-r_0)^2}{2\sigma_{\rm d}^2}\right) \tag{6.7}$$

with

$$\sigma_{\rm d} = \sigma \sqrt{\frac{\Omega_K D_{\rm d}({\rm St}+{\rm St}^{-1})}{c_s^2}} = \sigma \sqrt{\frac{\alpha_{\rm turb}}{{\rm Sc}\,{\rm St}}} \tag{6.8}$$

Note that this is only valid as long as $\alpha_{\rm turb} \ll {\rm Sc}\,{\rm St}$.

A more general solution, which is also valid for $\alpha_{\rm turb} \gtrsim {\rm Sc}\,{\rm St}$, can be found if we specify St at the peak of the bump, but allow it to rise as the gas density ρ drops, when we move away from r_0. The solution is then the radial version of the solution by Fromang & Nelson (2009) A&A 496, 597 (their Eq. 19):

$$\Sigma_{\rm d}(r) = \Sigma_{\rm d0} \exp\left[-\frac{{\rm Sc}\,{\rm St}_0}{\alpha_{\rm turb}}\left(\exp\left(\frac{\Delta r^2}{2\sigma^2}\right)-1\right)-\frac{\Delta r^2}{2\sigma^2}\right] \tag{6.9}$$

where we defined Δr as

$$\Delta r \equiv (r-r_0) \tag{6.10}$$

and ${\rm St}_0$ is the value of the Stokes number at the peak of the pressure bump.

A test of the code against this analytic solution is given in snippet `snippet_dustdrift_traptest_1.py`.

6.7 Some notes on multiple dust species or sizes

6.7.1 How dust species/sizes are associated to a disk model

As shown above, it is very easy to include multiple dust species in the disk model. Simply apply `add_dust()` multiple times, and each time you do this, a new dust component is created and added to the list `dust` (i.e. to `d.dust`, if `d` is an object of class `DiskRadialModel`). To remind you, here is how you can create three independent dust species:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
d.add_dust(agrain=1e-4,xigrain=3.6,dtg=0.1)    # The first dust component
d.add_dust(agrain=1e-3,xigrain=3.6,dtg=0.02)   # The second dust component
d.add_dust(agrain=1e-2,xigrain=3.6,dtg=0.001) # The third dust component
```

You can, alternatively, also create a new component yourself and add them to the list by hand:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
dust1 = DiskRadialComponent(d,agrain=1e-4,xigrain=3.6,sigma=self.sigma*0.1)
dust2 = DiskRadialComponent(d,agrain=1e-3,xigrain=3.6,sigma=self.sigma*0.02)
dust3 = DiskRadialComponent(d,agrain=1e-2,xigrain=3.6,sigma=self.sigma*0.001)
d.dust.append(dust1)
d.dust.append(dust2)
d.dust.append(dust3)
```

This has exactly the same effect.

In fact, strictly speaking, you do not even need to append these components to `d.dust`. If you do not append them to this list, then these components can still be 'live', but the disk model `d` will not 'know' about their existence. That is no problem, as long as you do not use methods of `DiskRadialModel` that require knowledge of the dust components (such as when using the one-zone radiative transfer method, see Section 9.2). So the following code would work fine:

```
from disklab.diskradial import *
from disklab.natconst import *
ntime  = 10
d = DiskRadialModel(mdisk=0.01*MS)
dust1 = DiskRadialComponent(d,agrain=1e-4,xigrain=3.6,sigma=self.sigma*0.1)
dust2 = DiskRadialComponent(d,agrain=1e-3,xigrain=3.6,sigma=self.sigma*0.02)
dust3 = DiskRadialComponent(d,agrain=1e-2,xigrain=3.6,sigma=self.sigma*0.001)
for itime in range(1,ntime+1):
   dust1.sigma=d.dust1.get_dust_radial_drift_next_timestep(time[itime]-time[itime-1],fixgas=True)
   dust2.sigma=d.dust2.get_dust_radial_drift_next_timestep(time[itime]-time[itime-1],fixgas=True)
   dust3.sigma=d.dust3.get_dust_radial_drift_next_timestep(time[itime]-time[itime-1],fixgas=True)
```

Here the dust components are modelled, even though they have not been appended to the disk model `d`.

Nevertheless it is often better to append them to the disk model, because methods such as `d.compute_viscous_evolution_and_dust` or `d.compute_onezone_intensity()` will take their dust information only from the list `d.dust`.

By the way, an object of the `DiskRadialComponent` class always "knows" its master: it knows which object of the `DiskRadialModel` class it belongs to. You can see this from the following example:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS,mstar=2*MS)
dusttogasratio = 0.02
myowndust = DiskRadialComponent(d,sigma=dusttogasratio*d.sigma,agrain=1e-2)
print('Star mass is: {} Msun'.format(myowndust.diskradialmodel.mstar/MS))
```

But conversely: an object of the `DiskRadialModel` class does not necessarily know all the dust components it owns, unless they are all added to the list `d.dust`. This is automatically the case if they were generated using `d.add_dust()`.

6.7.2 Dust components: from Python list to NumPy array, and back

Sometimes it can be unfavorable that the dust components are in a Python list rather than a NumPy array. This is, for instance, the case when you have multiple grain sizes, where each grain size is a dust component. At a given radius r you may want to analyze the grain size distribution, which is a function of grain size a. We thus want to have a function $\Sigma_{\rm d}(a)$, or in other words: an array `sigma[ia]`, where `ia` is the index for grain size. But because the dust components are added to the disk model as a Python list (instead of a NumPy array), this index `ia` is thus a list index, not an array index. That precludes many of the powerful NumPy methods for array manipulation.

To ameliorate this, you can simply create your own NumPy array out of the list:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
agrains      = np.array([1e-4,1e-3,1e-2,1e-1,1.0])
abundances   = 1e-3*(agrains/1e-1)**(1./6.)
for i in range(len(agrains)):
    d.add_dust(agrain=agrains[i],dtg=abundances[i])
nr = len(d.r)
na = len(agrains)
sigmadust = np.zeros((nr,na))
for ia in range(na):
    sigmadust[:,ia] = d.dust[ia].sigma[:]
```

For convenience DISKLAB offers a method for that: You can replace the last 5 lines with:

```
sigmadust = d.dust[0].join_multi_array(d.dust)
```

And once you manipulated `sigmadust` and want to put the results back into the `d.dust` list, you can use:

```
d.dust[0].return_multi_array(d.dust,sigmadust)
```

6.7.3 A note on grain size distributions

If we have a distribution of grain sizes in our disk, then the drift and mixing behavior of each of these grain sizes is different. We thus need to sample the grain size in terms of a discrete set of grain radii a_i with $i = 0, \cdots, n-1$, and equivalently grain masses m_i. Here n is the number of sampling points in grain size.

In DISKLAB this is handled by adding n dust components to the disk model. Each component can drift and mix independently. Whenever you need to study the grain size distribution at a given radius, you can use the methods `join_multi_array()` and `return_multi_array()` described in Section 6.7.2.

The precise definition of a *grain size distribution* sometimes leads to confusion, so let us define it here precisely. Suppose we want to model a powerlaw grain size distribution:

$$N(a) = N_0 (a/a_0)^\gamma \tag{6.11}$$

between the sizes $a_{\rm min}$ and $a_{\rm max}$. The choice of a_0 is arbitrary. For $\gamma = -3.5$ we have the usual Mathis, Rumpl & Nordsieck (MRN) distribution.

We have to normalize this distribution (i.e. choose N_0) such that the total surface density of dust is $\Sigma_{\rm d}$ in units of gram per square centimeter, because this is usually the parameter we wish to set (instead of the more abstract N_0). We thus have

$$\Sigma_{\rm d} = \int_{a_{\rm min}}^{a_{\rm max}} N(a) m(a) da \tag{6.12}$$

where $m(a)$ is the grain mass

$$m(a) = \frac{4\pi}{3} \xi\, a^3 \tag{6.13}$$

where ξ is the material density of the grain in gram per cubic centimeter.

We now want to sample this grain size distribution using n grain sizes. Since it is likely that $a_{\rm max} \gg a_{\rm min}$, it is best to use a logarithmic spacing in a. Here is a recommended way to divide this domain up in uniformly logarithmically spaced domains:

```
import numpy as np
n           = 20
amin        = 1e-5    # 0.1 micron
amax        = 1e-3    # 10 micron
agraini     = amin * (amax/amin)**np.linspace(0.,1.,n+1)
agrain      = 0.5 * ( agraini[1:] + agraini[:-1] )
```

Here `agraini` are the interfaces of the 'grid cells' in a-space, and `agrain` are the cell centers. Mathematically this is often written as

$$a_{i-1/2} = \text{agraini[i]}, \qquad a_i = \text{agrain[i]} \tag{6.14}$$

where $a_{i-1/2}$ are the left cell interfaces belonging to cells i, and a_i are the cell centers. Note that `agrain[0]` is not identical to `amin`, because `amin` is the left cell wall of the first cell, not the cell center (all 'cells' here being in a-space, also often called 'grain size bins'). The corresponding grain masses are:

```
xi          = 3.6     # Material density of the grain in g/cm^3
mgrain      = (4.*np.pi/3.)*xi*agrain**3
```

A grain size distribution in DISKLAB is simply a set of surface densities Σ_i, one for each 'grain size bin'. How does this relate to the grain size distribution of Eq. (6.11)? To see this, we have to integrate Eq. (6.11) over each cell:

$$\Sigma_i = \int_{a_{i-1/2}}^{a_{i+1/2}} N(a) m(a) da \tag{6.15}$$

which is the same as Eq. (6.12), but now over a single size bin only. For the powerlaw distribution of Eq. (6.11) we obtain

$$\Sigma_i = \frac{4\pi}{3}\frac{N_0\xi}{(\gamma+4)a_0^\gamma}\left[a_{i+1/2}^{\gamma+4} - a_{i-1/2}^{\gamma+4}\right] \tag{6.16}$$

However, if the spacing is small enough (i.e. n is large enough), then we can simplify this in the following way:

$$\Sigma_i = \int_{a_{i-1/2}}^{a_{i+1/2}} N(a)m(a)da = \int_{\ln a_{i-1/2}}^{\ln a_{i+1/2}} N(a)m(a)a\,d\ln a \tag{6.17}$$

which leads to

$$\Sigma_i \simeq \frac{4\pi}{3}\frac{N_0\xi}{a_0^\gamma}a_i^{\gamma+4}\Delta\ln a_i \tag{6.18}$$

where

$$\Delta\ln a_i = \ln a_{i+1/2} - \ln a_{i-1/2} \tag{6.19}$$

For a logarithmically equally spaced grid in a we have that $\Delta\ln a_i$ is a constant with i. We therefore conclude that a grain size distribution according to Eq. (6.11) can be represented by a set of surface densities Σ_i such that

$$\Sigma_i = \Sigma_{00}\left(\frac{a_i}{a_0}\right)^{\gamma+4} \tag{6.20}$$

In principle Σ_{00} follows from Eq. (6.18). But given that Eq. (6.18) is only approximate, it is better to determine Σ_{00} directly by the normalization:

$$\sum_{i=0}^{n-1}\Sigma_i = \Sigma_{\rm d} \tag{6.21}$$

where $\Sigma_{\rm d}$ is the total dust surface density (see also Eq. 6.12).

And so we recommend the following implementation of a powerlaw size distribution:

```
gamma   = -3.5    # Here the example of an MRN distribution
abun    = agrain**(gamma+4.)
abun   /= abun.sum()
```

where `abun` is the abundance of grain size i, normalized such that the sum of all abundances is 1.

To summarize, here is a snippet that implements an MRN size distribution into a disk model, and makes a plot of

$$\frac{d\Sigma_{\rm d}}{d\ln a} = \frac{\Sigma_i}{\Delta\ln a_i} \tag{6.22}$$

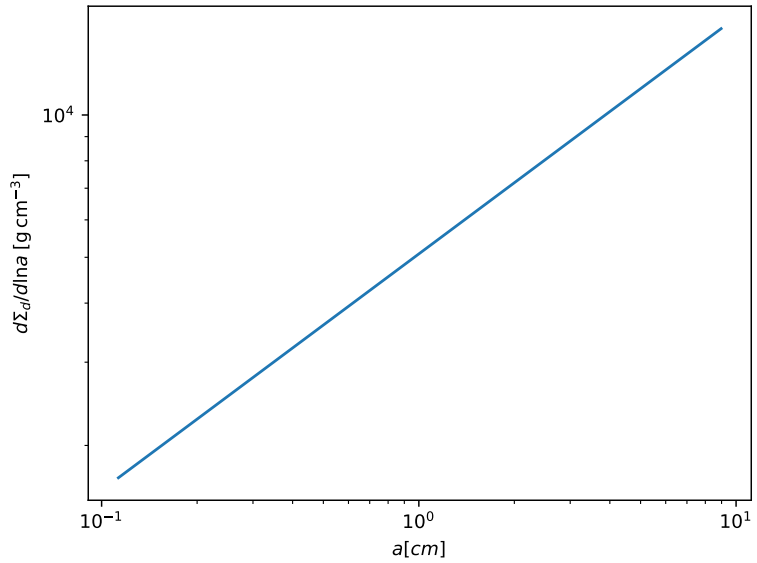
The code snippet is `snippet_grainsize_distribution_1.py`. In Python run it as:

```
%run snippet_grainsize_distribution_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, au, finalize

na      = 20
amin    = 1e-5     # 0.1 micron
amax    = 1e-3     # 10 micron
agraini = amin * (amax / amin)**np.linspace(0., 1., na + 1)
agrain  = 0.5 * (agraini[1:] + agraini[:-1])
xi      = 3.6      # Material density of the grain in g/cm^3
mgrain  = (4. * np.pi / 3.) * xi * agrain**3
gamma   = -3.5     # Here the example of an MRN distribution
abun    = agrain**(gamma + 4.)
abun    /= abun.sum()
rdisk0   = 3 * au  # Radius of the disk
sig0     = 1e3     # Gas surface density at rdisk0
```

```
d = DiskRadialModel()
d.make_disk_from_simplified_lbp(sig0, rdisk0, 1)
for ia in range(na):
    d.add_dust(agrain=agrain[ia], xigrain=xi, dtg=abun[ia])
sigmadust = d.dust[0].join_multi_array(d.dust)  # Make a 2-D array: Sigma(r,a)
dlna = np.log(agraini[1]) - np.log(agraini[0])  # Only valid for log grid in a

# Converted Sigma into dSigma/dln(a) = distr function

dsigdlna = sigmadust / dlna

# plotting

plt.figure()
plt.plot(agrain / 1e-4, dsigdlna[0, :])
plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$a [cm]$')
plt.ylabel(r'$d\Sigma_d/d\ln a\; [\mathrm{g}\,\mathrm{cm}^{-3}]$')

finalize(results=(sigmadust))
```



Note that the MRN distribution is dominated by the large grains, even though the powerlaw is -3.5. That is a well-known property of the MRN distribution.

Chapter 7

Grain properties and wavelength-dependent opacities

7.1 The `GrainModel` class

Dust plays a major role in protoplanetary disks. The properties of the dust grains affect both their dynamics and their appearance (i.e. opacities). These grain properties are managed by the `GrainModel` class. A `GrainModel` object contains only information about the physical properties of the grains in question, but not about their location in the disk (which is managed by the `DiskRadialComponent` class).

A `DiskRadialComponent` object (let's call it `dust`, for example) has – or at least should have – a `GrainModel` object:

```
dust.grain
```

Usually this `grain` is automatically created when you call `d.add_dust(agrain=0.1)`, for instance.

What does this object `grain` contain? Here is a list of its contents:

```
agrain                  # Grain radius in cm (should be same as dust.agrain)
xigrain                 # Material density in g/cm^3 (should be same as dust.xigrain)
mgrain                  # Grain mass in g, if computed via grain.compute_mgrain()
tstop                   # Stopping time in s, if computed using grain.compute_tstop()
species                 # Name of the dust species, if given e.g. when loading
                        # opacity with e.g. grain.load_standard_opacity()
sublimationmodel        # List object specifying if/how this dust species sublimates
                        # (see main text for explaination)
opac_lammic             # (If computed or loaded) Array of wavelength in micron
                        # for opacity table.
opac_kabs               # Likewise: Array of absorption opacity kappa_abs_nu
opac_ksca               # Likewise: Array of scattering opacity kappa_scat_nu
opac_ksca_eff           # As opac_ksca, but now multiplied by (1-g),
                        # with g the forward scattering factor g=<cos(theta)>
meanopactable_tgrid             # (If computed or loaded) Temperature grid for
                                # mean opacitytable
meanopactable_kappa_planck      # Likewise: Planck mean opacity (tabulated)
meanopactable_kappa_rosseland   # Likewise: Rosseland mean opacity (tabulated)
```

In addition to these, the object also contains a set of methods, which we will learn about in the coming sections.

The `opac_***` attributes represent the tabulated wavelength-dependent dust opacity for this grain. This table is not present automatically. It has to be computed or read-in from some file (see Sections 7.3 and 7.4).

The `meanopactable_***` attributes represent the tabulated Planck- and Rosseland-mean opacities. Also this table is not present automatically. First one has to load the wavelength-dependent opacity (see the `opac_***` attributes),

and from these the mean opacity tables are computed (see Section 7.8). Most methods that read or compute the `opac_***` opacity table, also automaticlly compute the mean opacities on a standard temperature grid and store them in the `meanopactable_***` arrays.

7.2 Computing the stopping time

One of the tasks of the `GrainModel` class is to compute the stopping time of the grain, for a given surrounding gas density and temperature. Here we describe how this is done. To compute the frictional stopping time of the grain, given a gas volume density ρ_g, temperature T and relative velocity Δv, one uses the method `compute_tstop()` from the `GrainModel` class. This subroutine follows Birnstiel, Dullemond & Brauer (2010) A&A 513, 79 and Perets & Murray-Clay (2011) ApJ 733, 56 to compute the stopping time $\tau_{\rm stop}$. We define the thermal velocity of the gas particles $v_{\rm th}$ as

$$v_{\rm th} = \sqrt{\frac{8k_BT}{\pi\mu m_p}} = \sqrt{8/\pi}\,c_s \tag{7.1}$$

with $\mu = 2.3$ and m_p the proton mass and $k_{\rm B}$ the Boltzmann constant[1]. For a grain with radius a and material density ξ the Reynolds number is

$$\mathrm{Re} = \frac{2a\Delta v}{\nu_{\rm mol}} \tag{7.2}$$

The molecular viscosity $\nu_{\rm mol}$ is

$$\nu_{\rm mol} = \frac{1}{2} v_{\rm th} \lambda_{\rm mfp} \tag{7.3}$$

where $\lambda_{\rm mfp}$ is the mean free path of the H_2 molecules given by

$$\lambda_{\rm mfp} = \frac{1}{n_{\rm gas}\sigma_{H_2}} \tag{7.4}$$

with $\sigma_{H_2} = 2\times10^{-15}\,\mathrm{cm}^2$ and n the number density of the gas given by

$$n_{\rm gas} = \frac{\rho_{\rm gas}}{\mu m_p} \tag{7.5}$$

Note that we, for simplicity, assumed here that all particles are H_2, i.e. we ignored the He atoms.

Now the stopping time is given by Eq. (10) of Birnstiel et al. (2010). For small particles ($\lambda_{\rm mfp}/a > 4/9$) the Epstein regime holds:

$$\tau_{\rm stop} = \frac{\xi a}{\rho_{\rm gas} v_{\rm th}} \qquad (\text{iff } \lambda_{\rm mfp}/a > 4/9) \tag{7.6}$$

For the Stokes regime (i.e. when $\lambda_{\rm mfp}/a \le 4/9$) we refer to Eq. (10) of Birnstiel et al. (2010). But for the intermediate Stokes regime ($1 < \mathrm{Re} < 800$) we take instead the Perets & Murray-Clay formulae (their Eqs. 6 and 7), which behave better. In all other regimes the two papers are mutually consistent.

Given the stopping time, we can compute the Stokes number:

$$\mathrm{St} = \Omega_K \tau_{\rm stop} \tag{7.7}$$

Note that in accretion disks the largest turbulent eddies have a turnover time scale which is the same as the orbital time scale. This means that particles with $\mathrm{St} = 1$ have the stopping time equal to the largest eddie time scale.

Here is an example how one can use the `graimmodel` class to compute all these quantities:

```
from disklab.grainmodel import *
g = GrainModel(agrain=0.1*1e-4,xigrain=2.0)  # Grain a=0.1 mum and xi=2 gram/cm^3
rhogas = 1e-10          # Gas density in g/cm^3
tgas   = 40.            # Gas temperature in K
dv     = 10.            # Velocity between dust and gas in cm/s
g.compute_tstop(rhogas,tgas,dv)
print(g.tstop)          # This gives the stopping time in seconds
```

[1]Note that in Birnstiel, Dullemond & Brauer (2010) the formula for $\bar{u}$ (which is their symbol for $v_{\rm th}$) uses $\bar{u} = \sqrt{\pi/8}c_s$, which is erroneous; it should be $\bar{u} = \sqrt{8/\pi}c_s$. It was correctly implemented in the code belonging to that paper.

which gives a number of about 3.3 seconds for this example.

The `DiskRadialComponent` class calls the `compute_tstop()` method to compute the stopping time and, from that, the Stokes number.

7.3 Reading opacity table from file

Another main task of the `GrainModel` class is to take care of the wavelength-dependent dust opacities. One way to load such an opacity table into DISKLAB is to use the `grain.read_opacity()` method. This method simply reads an opacity table directly from a file.

There are *two possible opacity file formats*:

- The standard RADMC-3D opacity files `dustkappa_***.inp`, valid only for a single grain size. The user must be sure that the grain size for which this opacity file was created matches that of the grain model (i.e. matches `grain.agrain`). This is not automatically verified.
- The DISKLAB opacity files `dustkappa_***.npz`, which are much more flexible than the RADMC-3D format. It generally contains the opacity table for a series of different grain sizes, so that the `grain.read_opacity()` method can automatically find (and linearly interpolate in `agrain`) the opacity wavelength-dependent opacity table.

where `***` can be any name of a dust compositional species (e.g. `dustkappa_mydust.inp`).

The simplest of the two formats is the standard RADMC-3D opacity file format (`dustkappa_***.inp`), which is an ascii (plain text) file. The most common form is:

```
iformat                      <== This example is for iformat==3
nlam
lambda[1]         kappa_abs[1]       kappa_scat[1]      g[1]
   .                   .                  .              .
   .                   .                  .              .
lambda[nlam]     kappa_abs[nlam]    kappa_scat[nlam]    g[nlam]
```

That is: the first line should contain a 3, the second line the number of wavelength points, and then a table with four columns: wavelength in micrometer, absorption opacity in cm^2/gram-of-dust, scattering opacity in same units, and the forward-scattering factor g defined as the expectation value of the cosine of the scattering angle θ. See Chapter B for information how to create these opacity files from optical constants. Note that the RADMC-3D opacity file format also allows a three-column version, in which the g column is not present (first line contains 2, i.e. `iformat=2`), or even a two-column version, in which also the scattering opacity is omitted (first line contains 1, i.e. `iformat=1`).

The other format, i.e. the `dustkappa_***.npz` format employs the standard zip-compressed `numpy` file format. The `grain.read_opacity()` reader, when it reads an opacity file with `.npz` extension, will read this file in using the standard `numpy.load()` method. If you are curious as to the content of this file, you can have a look inside:

```
import numpy as np
data = np.load('dustkappa_mydust.npz')
```

(assuming you have created, or copied, a file of that kind, see Chapter B). You can then see that the file contains a 1-D array of wavelengths in cm called `lam` and another 1-D array of grain radii in cm called `a`, and the corresponding 2-D arrays `k_abs[:,:]`, `k_scat[:,:]` and optionally `g[:,:]`, all three of size `[len(a),len(lam)]`. The material density is given by `rho_s`. When `grain.read_opacity()` reads such a file, it automatically linearly interpolates `grain.agrain` in the array `a` of the opacity file, and correspondingly extracts the linearly interpolated opacity values. See Chapter B for information how to create these opacity files from optical constants.

Here is an example how to read and plot a DISKLAB dust opacity file. The standard opacity files can be found in the directory `disklab/opacity/precalculated/`.

[FINISH THIS PART]

This will not only read the opacity table, it will also automatically calculate the Planck- and Rosseland-mean opacities (without accounting for sublimation) for this opacity table (see Section 7.8 as well as Chapter 8). The wavelength-dependent opacity tables are stored in the `d.dust[0].grain.opac_***` arrays. The precalculated mean opacities are stored in `d.dust[0].grain.meanopactable_***` (see Section 7.1 and Section 7.8).

Note that `dustkappa_astrosilicate.npz` is just an example dust opacity (the famous 'astronomical silicate' model of Bruce Draine). We discuss in Appendix B how to create your own opacity tables. For now we will stick with `dustkappa_astrosilicate.inp`.

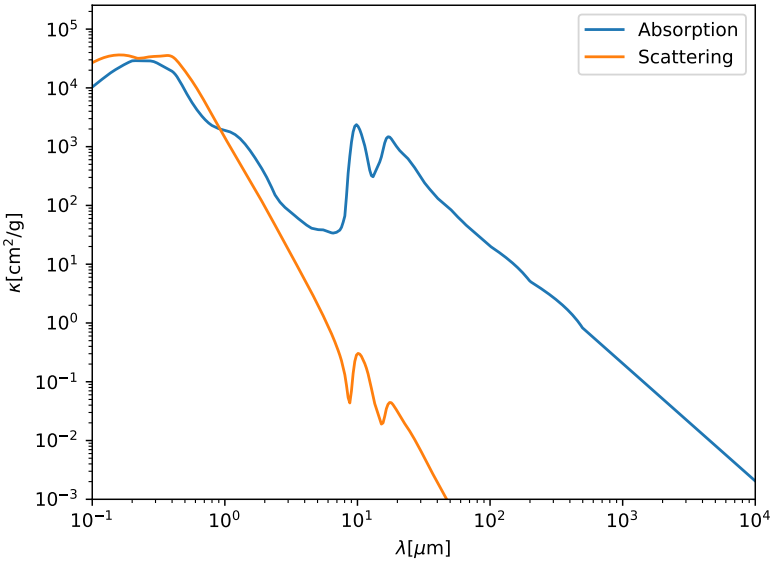
It is useful to inspect what the opacity looks like. So let us plot the opacity table.

The code snippet is `snippet_plot_dustopac_1.py`. In Python run it as:

```
%run snippet_plot_dustopac_1.py
```

Here is the listing:

```
from disklab.grainmodel import GrainModel
from snippet_header import plt, finalize
grain = GrainModel()
grain.read_opacity('dustkappa_silicate.inp')

plt.figure()
plt.plot(grain.opac_lammic, grain.opac_kabs, label='Absorption')
plt.plot(grain.opac_lammic, grain.opac_ksca, label='Scattering')
plt.xlabel(r'$\lambda [\mu\mathrm{m}]$')
plt.ylabel(r'$\kappa [\mathrm{cm}^2/\mathrm{g}]$')
plt.xscale('log')
plt.yscale('log')
plt.xlim(1e-1, 1e4)
plt.ylim(bottom=1e-3)
plt.legend(loc='upper right')

finalize()
```



One simple way to get the opacity at the wavelength you need is to use numpy's `interp` routine:

```
lammic = 1300.    # Wavelength of observation in micron
kabs   = np.interp(lammic,o.opac_lammic,o.opac_kabs)
kabs
Out: 0.12125583143387224
```

[MODIFY SNIPPET AND OUTPUT VALUE]

7.4 Standard precomputed opacities

You are encouraged to create your own `dustkappa_***.inp` opacity files using a Mie code and to use these for the DISKLAB modeling. But very often you want to simple make some *quick-n-dirty* models with some standard opacities, without having to think hard about which opacities you really want to compute (and how).

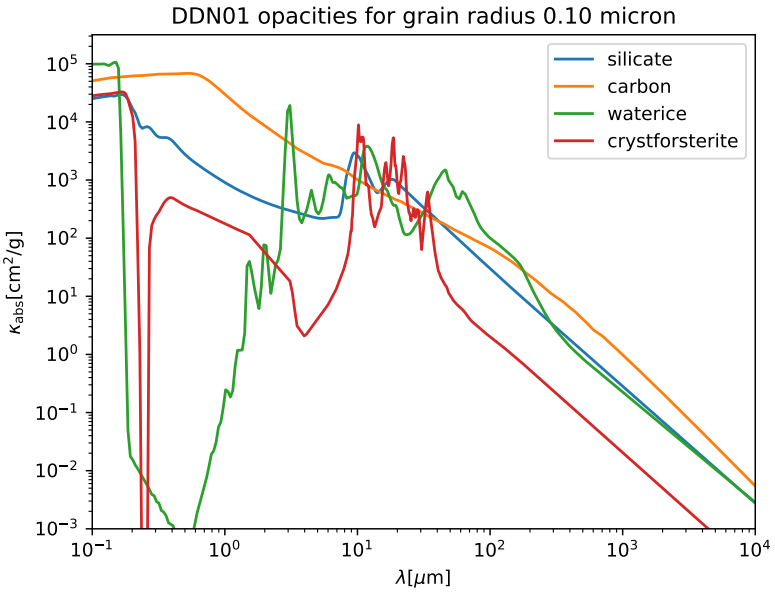
DISKLAB offers a set of standard opacities for this purpose, and a special method to read these standard opacities in. This method is `load_standard_opacity()`. This method is nothing fancy: it in fact uses `read_opacity()` to read in the standard opacities. The only 'special' thing about `load_standard_opacity()` is that you do not have to worry about *where* these standard opacity files reside (FYI: they reside in the directory `disklab/disklab/opacities/precalcula` but you do not need to know). Note also that these precomputed standard opacities each have their `.info` file containing information about the grain radius and material density, which will automatically be read by the method `load_standard_opacity()` as well.

Example: The code snippet is `snippet_plot_dustopac_3.py`. In Python run it as:

```
%run snippet_plot_dustopac_3.py
```

Here is the listing:

```
from disklab.grainmodel import GrainModel
from snippet_header import plt, finalize

grain = GrainModel()

plt.figure()
grain.load_standard_opacity('ddn01', 'silicate')
plt.plot(grain.opac_lammic, grain.opac_kabs, label='silicate')
grain.load_standard_opacity('ddn01', 'carbon')
plt.plot(grain.opac_lammic, grain.opac_kabs, label='carbon')
grain.load_standard_opacity('ddn01', 'waterice')
plt.plot(grain.opac_lammic, grain.opac_kabs, label='waterice')
grain.load_standard_opacity('ddn01', 'crystforsterite')

plt.plot(grain.opac_lammic, grain.opac_kabs, label='crystforsterite')
plt.title('DDN01 opacities for grain radius {0:4.2f} micron'.format(grain.agrain * 1e4))
plt.xlabel(r'$\lambda [\mu\mathrm{m}]$')
plt.ylabel(r'$\kappa_{\mathrm{abs}} [\mathrm{cm}^2/\mathrm{g}]$')
plt.xscale('log')
plt.yscale('log')
plt.xlim(1e-1, 1e4)
plt.ylim(bottom=1e-3)
plt.legend(loc='upper right')

finalize()
```



Currently the following standard dust opacities are built in:

Series	Name	Chem	Cryst	ξ[g/cm^3]	$a[\mu m]$	Reference
ddn01	silicate	$MgFeSiO_4$	amorph	3.6	0.1	Laor & Draine (1993) ApJ 402, 441
ddn01	carbon	C	amorph	1.8	0.1	Preibisch et al. (1993) A&A 279, 577
ddn01	waterice	H_2O	amorph	1.0	0.1	Warren S.G. (1984), Appl.Opt. 23, 1206
ddn01	crystforsterite	Mg_2SiO_4	cryst	3.275	0.1	Servoin & Piriou (1973) Phys.Stat.Sol. 55

where ξ is the material density (specific weight) in gram/cm^2, and a is the grain radius in μm.

[REWRITE THIS SECTION]

7.5 A simple opacity model with grain-size dependence

Sometimes it can be useful to use a much simpler dust opacity, in particular for testing purposes. The method `grain.compute_simple_opacity()` computes the simple dust opacity model of the Ivezic et al. (1997) MNRAS 291, 121. This simple model is:

$$\kappa_\nu^{\rm abs} = \frac{\pi a^2}{m} \begin{cases} 1 & \text{for } \lambda < 2\pi a \\ 2\pi a/\lambda & \text{for } \lambda > 2\pi a \end{cases} \tag{7.8}$$

We do not include the scattering part of the opacity of Ivezic et al., simply because scattering is not really included in the physics of DISKLAB.

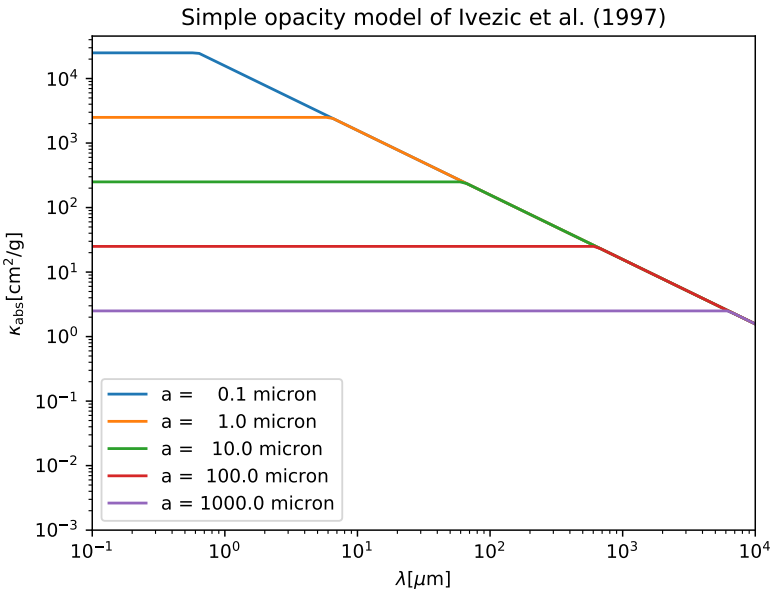
One main advantage of this simple opacity model over the standard pre-computed opacity tables of Section 7.4 is that you can specify any grain size and material density. The disadvantage is, of course, that it is *highly* approximative. It does, however, obey two of the most basic properties of dust grain opacities: that for wavelengths $\lambda \gg a$ the absorption opacity is 'by volume' (and therefore, for a given amount of dust, does not change with grain size a), while for $\lambda \ll a$ the opacity is 'by surface area' (and therefore, for a given amount of dust, drops proportional to $1/a$).

The code snippet is `snippet_plot_dustopac_4.py`. In Python run it as:

```
%run snippet_plot_dustopac_4.py
```

Here is the listing:

```
from disklab.grainmodel import GrainModel
from snippet_header import plt, np, finalize

grain = GrainModel()
grain.xigrain = 3.0
agr = 1e-5 * 1e4**np.linspace(0., 1., 5)

plt.figure()
for agrain in agr:
    grain.agrain = agrain
    grain.compute_simple_opacity()
    plt.plot(grain.opac_lammic, grain.opac_kabs,
             label='a = {0:6.1f} micron'.format(agrain * 1e4))
plt.xlabel(r'$\lambda [\mu\mathrm{m}]$')
plt.ylabel(r'$\kappa_{\mathrm{abs}} [\mathrm{cm}^2/\mathrm{g}]$')
plt.xscale('log')
plt.yscale('log')
plt.xlim(1e-1, 1e4)
plt.ylim(bottom=1e-3)
plt.legend(loc='lower left')
plt.title('Simple opacity model of Ivezic et al. (1997)')

finalize()
```



[REWRITE THIS SECTION: IN READ STANDARD OPACITY]

7.6 Sublimation and freeze-out ('ice lines')

Dust grains (in particular volatile ones such as those made of ice) can only survive up to some temperature. Above that they sublimate and disappear. Including dust opacities in a disk model without accounting for sublimation means that the disk model will only be valid as long as the temperatures remain below this sublimation temperature.

However, the `GrainModel` class provides methods to automatically reduce (or put to zero) the abundance of a given grain species under a given set of pressure and temperature conditions. The main method is `GrainModel.abundance_after_sublimation()`. We will discuss this method in Subsection 7.6.2.

But first we have to tell DISKLAB *how* to sublimate the dust.

7.6.1 The sublimation models

In the literature there are various ways and formulae to compute the temperature of sublimation and/or the degree of sublimation for a given temperature. The physics is complex and involves partial pressures of vapor (or vapor components), the Hertz-Knudsen equation, and equilibrium vapor pressures calculated from Gibbs free energy considerations. Often much of this complexity is simplified using an effective formula for the equilibrium vapor pressure. Different papers use different formula-types. Some papers only compute the sublimation temperature and assume all solids of that type to be gone for higher temperature. DISKLAB implements sublimation in a flexible way, allowing various recipes to be used.

Sublimation of dust can also cause *numerical* complications. Consider the case of a disk with internal viscous heating. The higher the optical depth, the hotter the midplane temperature becomes. This may cause part of the dust to sublimate and vanish, thus reducing the opacity. A reduced opacity lowers the midplane temperature, meaning that the dust can recondense. A flip-flop can occur that does not converge. Physically the problem is solved because there is a temperature domain in which not all dust instantly sublimates. But numerically this temperature range may be too narrow to cause convergence. A simple "dirty trick" that is often applied is to "soften" the transition from full-dust to no-dust with a powerlaw. This method is, for instance, used in the famous Bell & Lin opacity model to handle the sublimation of water ice and of silicate dust. DISKLAB allows you to include such a smoothed sublimation.

Which of the many sublimation models you wish to use is set by a list called `sublimationmodel`. The `GrainModel` class has this as a standard attribute. For instance for a standard disk component:

```
d.dust[0].grain.sublimationmodel
```

We will learn in the next paragraphs how to set `sublimationmodel` to specify a particular sublimation model.

The simplest implementation of sublimation is `'tsub'` where you simply set the sublimation temperature to a given value:

```
grain.sublimationmodel = ['tsub',{'tsub':1500.}]
```

where this dust species will simply sublimate entirely for temperatures above 1500 Kelvin. If you also specify `'plaw'`, then you can smooth the sublimation with a powerlaw, as discussed above, to avoid numerical problems:

```
grain.sublimationmodel = ['tsub',{'tsub':1500.,'plaw':-10}]
```

A more sophisticated method is to provide an analytical formula for the equilibrium vapor pressure. This model is called `'peq'`. The typical formula is:

$$p_{\rm vap}^{\rm eq} = \exp\left(-\frac{a}{T}+b\right) \tag{7.9}$$

where T is in Kelvin and $p_{\rm vap}^{\rm eq}$ is in units of dyne/cm^2. You can specify this like this:

```
grain.sublimationmodel = ['peq',{'peq_a':6070.,'peq_b':30.86,'mu':18}]
```

where `'peq_a'` stands for a and `'peq_b'` stands for b. The `'mu'` is the molecular weight of the vapor particle in units of proton mass. This example is, by the way, the sublimation curve for water ice (Bauer et al. 1997).

Some papers specify, instead, the equilibrium vapor *density*:

$$\rho_{\rm vap}^{\rm eq} = \frac{1}{T}\exp\left(-\frac{a}{T}+b\right) \tag{7.10}$$

You can specify this like this:

```
grain.sublimationmodel = ['peq',{'rhoeq_a':28030.,'rhoeq_b':12.471,'mu':24}]
```

where `'rhoeq_a'` stands for a and `'rhoeq_b'` stands for b. This example is, by the way, the sublimation curve for amorphous olivine. The `'mu'` molecular weight is here just an estimate of the average molecular weight, because the sublimation of olivine involves not just a single vapor species, but multiple. But this does not matter, because the `'mu'` drops out of the equations anyway.

In DISKLAB we have collected a set of standard vapor curves. You can specify this as:

```
grain.sublimationmodel = ['peq',{'species':'H2O'}]
```

for water ice. Included species are `'H2O'`, `'NH3'`, `'CO2'`, `'H2S'`, `'C2H6'`, `'CH4'`, `'CO'`, `'MgFeSiO4'`, `'Mg2SiO4'`, `'MgSiO3'`, `'Fe'`, `'Al2O3'`, `'SiO2'`, `'FeS'`. More may follow.

7.6.2 Using the sublimation model to compute the degree of sublimation

Now let us see how we can actually use the sublimation model to sublimate dust. Suppose we have a `GrainModel` object called `grain`, and the abundance of this dust at some point in the disk *would* be `abun0` if the grains would not sublimate (where `abun0` is by mass, i.e. it is $\rho_{\rm dust}/\rho_{\rm gas}$). The method `abundance_after_sublimation()` can now be invoked to compute how much of this grain material is still left after letting it sublimate:

```
abun = grain.abundance_after_sublimation(abun0,rhogas,temp)
```

You can experiment with different values of `rhogas` (in gram/cm^3) and `temp` (in K). Note that if the temperature is low, you will see that `adun` will be equal to the original `abun0`, because nothing has sublimated. For very high temperatures, you will see that `abun` becomes 0. Only close to the sublimation temperature you will get a reduced-but-not-zero abundance.

How you now use this reduced abundance in your model remains up to you. In the mean opacity models of Chapter 8 the method `abundance_after_sublimation()` is used to automatically implement the sublimation, assuming that the disk component surface density is the *non-sublimated* dust surface density.

7.7 Computing the dust opacity for a given grain size

[THIS SECTION HAS TO BE RECONSIDERED]

7.8 Planck- and Rosseland mean opacities without sublimation

If you have read in a dust opacity table into a `GrainModel` object, then you may be also interested in the *mean* opacities resulting from this table.

The *Planck mean opacity* is the wavelength-averaged opacity weighted by the Planck function:

$$\kappa_P^{\rm abs}(T) \equiv \frac{\int_0^\infty \kappa_\nu^{\rm abs} B_\nu(T) d\nu}{\int_0^\infty B_\nu(T) d\nu} \tag{7.11}$$

It is a function of temperature T even though the dust opacity itself is not a function of temperature (apart from sublimation, but let us ignore this for now). The Planck mean opacity is particularly useful for computing the energy balance of dust grains irradiated by a source of radiation.

The *Rosseland mean opacity* is anothe form of wavelength-averaged opacity, but with a different weighting and averaging:

$$\kappa_{\rm Ross}(T) = \frac{\int_0^\infty (\partial B_\nu(T)/\partial T) d\nu}{\int_0^\infty (1/\kappa_\nu)(\partial B_\nu(T)/\partial T) d\nu} \tag{7.12}$$

The Rosseland mean opacities are particularly useful for computing the diffusion of thermal radiation through an optically thick medium.

Note: The Rosseland mean opacity has, formally, only true meaning for the *sum* of all opacities involved (when we have a mixture of materials). So if our disk has 3 disk components, each representing a different dust species, then the Rosseland mean opacity of each individual species is (formally speaking) not meaningful. Only the Rosseland mean of the total opacity (of all three components combined) has formal meaning. Nevertheless, it turns out that often you can still approximate the true Rosseland mean opacity of the mixture of components from a suitable average of the individual Rosseland means. See Section 8.3 for more information.

You can compute the Planck- and Rosseland-mean opacities for a given temperature T using the following methods:

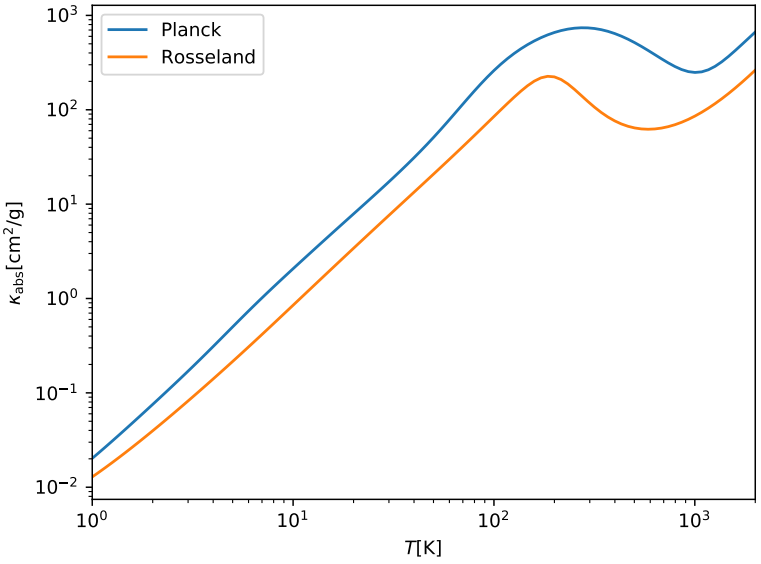
```
temp = 30.      # Temperature at which you want to compute the Rosseland mean opacity
kapross = grain.rosselandmean(temp)
kapross
Out: 7.6048813273354812
```

Likewise you can compute the Planck mean opacity:

```
temp = 30.      # Temperature at which you want to compute the Planck mean opacity
kappl = grain.planckmean(temp)
kappl
Out: 17.255704615914848
```

Let's plot the Rosseland and Planck mean opacities as a function of temperature.
The code snippet is `snippet_plot_dustopac_2.py`. In Python run it as:

```
%run snippet_plot_dustopac_2.py
```



NOTE: The dust opacity is the opacity *per gram of dust*, not per gram of gas+dust. So you have to beware of the proper dust-to-gas ratio factor when using it in combination with the gas.

The computation of these mean opacities on run-time can be making the code very slow. Therefore they are pre-computed and tabulated. This is done by the method `grain.tabulate_mean_opacities_without_sublimation()`. The opacity reading methods `grain.read_opacity()` and `grain.load_standard_opacity()` both automatically call this pre-computation method, so you do not have to call it yourself. These pre-computed mean opacities are stored in the `grain.meanopactable_***` arrays.

The pre-computation of these tables is time-costly at startup. That is why the first time you read in an opacity table, it will take a bit of time. However, DISKLAB will write the table to file. The next time you read the opacity file, the mean opacities will also be read from the previously computed file. DISKLAB will check if the tabulated mean opacities in this file are still up-to-date with the wavelength-dependent opacity file from which it was created. This is done using a hashing function SHA1 on the opacity file. Hence, the second line of the file `dustkappa_silicate.meanopac` is a hash, which is the hash of the file `dustkappa_silicate.inp`. If the hashes do not match, the mean opacity table is recomputed.

Note that the mean opacities computed here are *without* sublimation. Sublimation should be included by the appropriate reduction of the abundance of this dust species. Some methods (such as the mean opacity methods of Chapter 8) can do this automatically, but not all.

Chapter 8

Mean opacities for the disk model

For computing the midplane temperature of the disk it is necessary to know the Rosseland mean opacity and Planck mean opacity of the material of the disk.

Important: In DISKLAB the mean opacity and the frequency-dependent opacity are (algorithmically speaking) independent from each other. The mean opacities are only required for computing the disk interior (midplane) temperature, and all the quantities that depend on that. The frequency-dependent opacity is only used for computing the appearance of the disk at a given frequency or wavelength. One *can* (and perhaps *should*) compute the mean opacities from the wavelength-dependent ones, but from the perspective of the DISKLAB architecture, they are in principle independent, and serve different purposes.

This chapter discusses the *mean* opacities.

The Rosseland- and Planck mean opacities are functions of gas density and temperature, but not of wavelength or frequency (hence they are called *mean* opacities). For dust opacities they are only a function of temperature, except when dust sublimation comes into play. Details about how one can compute the mean opacities from the wavelength-dependent dust opacities follow in Section 8.3.

The mean opacities can vary from location to location. Not only because the composition of the disk may vary from place to place (for instance, when different dust grain components drift at different speeds), but also because of the temperature and density dependence of the mean opacities themselves. At radii r close to the star, i.e. in the hot regions of the disk, the mean opacity of silicate dust will be very different from the mean opacity of the same silicate dust in the disk cold outer regions (see Section 8.3).

A simple, yet flexible, implementation of the mean opacities is therefore not trivial. The solution we have chosen in DISKLAB is described in Section 8.1. Various ways to compute these opacities are described in the following sections.

8.1 Basic architecture of mean opacity implementation

At any moment in time, the Rosseland mean opacity as a function of radius r is (or better: should be) stored in the array

```
d.mean_opacity_rosseland[:]
```

where `d` is the `DiskRadialModel` object. This is an array with the same size as `d.r`, and it represents κ_R at each of these points. Likewise the Planck mean opacity κ_P is stored in the array

```
d.mean_opacity_planck[:]
```

As you can see: these values are given only as a function of r. The density and temperature dependency is not explicitly stored here.

When `DiskRadialModel` needs κ_P or κ_R at a given radius r (for instance, to compute the midplane temperature, see Section 4), it will obtain these values from `d.mean_opacity_planck[:]` and `d.mean_opacity_rosseland[:]`.

Important: *You* (the user) will have to give `DiskRadialModel` the command `compute_mean_opacity()` to compute the values of `d.mean_opacity_planck[:]` and `d.mean_opacity_rosseland[:]` according to some recipe. And you have to *re*-do this whenever the local conditions in the disk change, which is, for instance, the case when you viscously evolve the disk.

There are several ways by which `DiskRadialModel.compute_mean_opacity()` can compute these values. To be more precise: DISKLAB offers several *mean opacity models*:

- **'supersimple'** The simplest opacity model available: you simply specify a value to be used. See Section 8.2.
- **'dustcomponents'** Self-consistently compute the mean opacities from the available dust components in the `d.dust` list. See Section 8.3.
- **'tabulated'** A user-provided 2-D table of $\kappa_R(\rho_g, T)$ and $\kappa_P(\rho_g, T)$ values, for a discrete set of values of the gas density ρ_g and temperature T.
- **'belllin'** The famous Bell & Lin (1994) opacity model.

The way to do this is to first set the `meanopacitymodel` variable of your `DiskRadialModel` object. Let us take the `'supersimple'` model as an example:

```
from disklab.diskradial import *
from disklab.natconst import *
d=DiskRadialModel(mdisk=0.01*MS)
d.meanopacitymodel=['supersimple',{'dusttogas':0.01,'kappadust':1e3}]
d.compute_mean_opacity()
```

As you can see, the `d.meanopacitymodel` is a list. Its first element is the name of the mean opacity model. The second element is a Python dictionary with parameters for that mean opacity model. All mean opacity models follow this scheme (first element is name, second element is dictionary with parameters).

For this 'supersimple' mean opacity model the values will be constant, independent of the local conditions in the disk. However, for the other mean opacity models this is not the case: whenever the disk changes (for instance, because it is viscously evolving or photoevaporating or you name it), the mean opacities have to be recalculated. This is done simply with

```
d.compute_mean_opacity()
```

Important: If you evolve the disk but forget to call `d.compute_mean_opacity()`, you are likely to get wrong results, especially if your model relies on proper values of the mean opacities.

Note: In contrast to the dust opacities, the mean opacities for a disk model are *cross-section per gram of gas*. For a gas+dust mixture, the opacity is caused primarily by the dust, but the weighting is normalized to the gas surface density. The reason for this is simple: if we are interested in computing the midplane temperature, we are only interested in the vertical optical depth, no matter if this is caused by dust or gas. Given that the disk has only one gas component (`DiskRadialModel.sigma[:]`) but possibly multiple dust components (`DiskRadialModel.dust[:].sigma[:]`), it makes life easier to simply normalize the opacity to the gas surface density.

8.2 The super-simple mean opacity model

The super-simple mean opacity is simply a way to set the value of the mean opacity to a number given by you. The simplest way is:

```
d.meanopacitymodel = ['supersimple',{'kappagas':1e0}]
d.compute_mean_opacity()
```

You can also specify a hypothetical dust opacity, and a dust-to-gas ratio:

```
d.meanopacitymodel = ['supersimple',{'dusttogas':0.01,'kappadust':1e2}]
d.compute_mean_opacity()
```

(note that both examples give the same result).

You can also use this model to specify a r-dependent opacity, simply by giving an array instead of a value:

```
kappaarray          = np.ones_like(d.sigma)
d.meanopacitymodel = ['supersimple',{'kappagas':kappaarray}]
d.compute_mean_opacity()
```

where of course you set `kappaarray` to some more useful array than just `np.ones_like(d.rhogas)`, as long as it has the same number of elements as `d.rhogas`.

The super-simple opacity model is, in fact, so simple, that you could also do it by hand, without calling `d.compute_mean_opacity()`:

```
kappa = 0.01*1e3    # Dust-to-gas ratio 0.01 and dust kappa 1000
mean_opacity_planck[:] = kappa
mean_opacity_rosseland[:] = kappa
```

8.3 Computing the mean opacities self-consistently from the dust opacities

8.3.1 Basic method

You can construct the mean opacity arrays from the available dust components in the disk (or, instead, from the dust components you explicitly give as arguments to this method). As usual, the `d.meanopacitymodel` consists of a list of two elements, the first being `'dustcomponents'`, the second being a dictionary with settings. The most important dictionary element is 'method'. It can have the following values:

- **'fullcalculation':** Compute the mean opacities from the full frequency-dependent opacities of the dust components. This model is very time-consuming, because it will recalculate the integrals over the full frequency grids. For fast calculations of disk models, this is not ideal. But there can be circumstances by which this is the better method: if different dust species mix/drift in different ways, so that the composition of the dust mixture is different at different times.
- **'simplemixing':** Use mean opacities from each of the dust species individually, and average them weighted by abundance. For a single dust species this is of course exact. For multiple dust species this is not quite correct, but it is fast. The reason why it is not quite correct is that the Rosseland mean of the mixed frequency dependent opacity is not equal to the average of the Rosseland mean of the individual dust opacities. It is only correct if the frequency- dependent opacities are 'correlated'. In atmospheric radiative transfer physics they call this the 'correlated k assumption'. It is the method of choice if one wants to allow time- and space-varying dust abundances, while avoiding the time- consuming 'fullcalculation' method.

Example:

```
from disklab.diskradial import *
from disklab.natconst import *
d=DiskRadialModel(mdisk=0.01*MS)
d.add_dust(agrain=0.1,xigrain=2.0,dtg=0.005)
d.add_dust(agrain=0.01,xigrain=2.0,dtg=0.005)
for dust in d.dust:
   dust.grain.compute_simple_opacity()
d.meanopacitymodel = ['dustcomponents',{'method':'simplemixing'}]
d.compute_mean_opacity()
```

The call to `dust.grain.compute_simple_opacity()` computes the simple dust opacity model of the Ivezic et al. (1997) MNRAS 291, 121 paper, and also (as a kind of 'bonus') computes a table of Planck and Rosseland mean opacities for this single dust grain. The `'simplemixing'` mean opacity method simply uses these tables to compute the mean opacities for the full opacity model (i.e. that of the dust mixture).

See Chapter 7 for more information about dust opacities.

Let us do a more complex (and realistic) example: that of two standard grain models: the 'astronomical silicate' model of Laor & Draine (1993) ApJ 402, 441, and 'amorphous carbon' measurements of Preibisch et al. (1993) A&A 279, 577. These are two of the opacity models from the disk modeling paper of Dullemond, Dominik & Natta (2001) ApJ 560, 957, hence the name `ddn01`.

```
from disklab.diskradial import *
from disklab.natconst import *
d=DiskRadialModel(mdisk=0.01*MS)
d.add_dust(agrain=0.1e-4,xigrain=3.6,dtg=0.005)
d.add_dust(agrain=0.1e-4,xigrain=1.8,dtg=0.005)
d.dust[0].grain.load_standard_opacity('ddn01','silicate')
d.dust[1].grain.load_standard_opacity('ddn01','carbon')
for dust in d.dust:
   assert dust.grain.agrain==dust.agrain.max()
   assert dust.grain.xigrain==dust.xigrain.max()
d.meanopacitymodel = ['dustcomponents',{'method':'simplemixing'}]
d.compute_mean_opacity()
```

The `assert` statements are meant to make sure that the `agrain` and `xigrain` we specify for the disk components are the same as those of the precalculated standard dust opacities. This is because the precalculated standard dust opacities have a fixed grain size and material density. Using these opacities for a dust component with different grain size or material density would be physically inconsistent. In the above example these `assert` statements are, of course, passed succesfully.

The `'fullcalculation'` method can be compared to the `'simplemixing'`

```
kapros_simple = d.mean_opacity_rosseland.copy()
d.meanopacitymodel = ['dustcomponents',{'method':'fullcalculation'}]
d.compute_mean_opacity()
kapros_full = d.mean_opacity_rosseland.copy()
```

These two are usually not much different, showing that in most cases the `'simplemixing'` method is accurate enough.

8.3.2 Including sublimation physics

When including dust species that can easily sublimate (e.g. ices), it is imperative to include the sublimation physics. The location where this phase transition occurs is often called the 'ice line'. In fact, even silicate or carbon dust will sublimate at high enough temperatures. Simply computing the mean opacities without taking into account this sublimation would yield unphysical mean opacities.

The sublimation and freeze-out physics is included in the `GrainModel` class and is described in Section 7.6. By default this is not included in the computation of the mean opacities. If you wish to include the sublimation and freeze-out in the computation of the mean opacities, you have to explicitly activate this. Either you reduce the dust component surface densities youself according to the methods described in Section 7.6 (this requires a bit of tinkering on your side), or you can let `compute_mean_opacity()` do it automatically for you, which is done with the `'autosublim'`, as in:

```
d.meanopacitymodel = ['dustcomponents',{'method':'simplemixing','autosublim':True}]
```

Note: If you use the `'autosublim':True`, then `d.compute_mean_opacity()` interprets the dust component surface densities `d.dust[:].sigma` as the *combined dust+vapor surface density*. So it depends on your modeling goals whether you use the easy-to-use automatic `'autosublim':True` method, or whether you instead handle the dust and vapor as separate dust components and implement the sublimation physics directly as a source/sink exchange term between these components using the sublimation physics routines of Section 7.6.

8.3.3 When all the dust is gone: Gas opacity model of Bell & Lin

For very hot regions of the disk, all dust will be sublimated. When we compute the mean opacities from the dust components in the disk, this would mean that the mean opacity will drop to zero. This can cause problems for the model, and is also not very realistic. To ameliorate this, you can ask the mean opacity model to add the *gas-opacity part* of the Bell & Lin (1997) opacities (see Section 8.5) to be added. When the dust is gone, the gas opacities remain. This can be done by adding `'gasbelllin':True` like this:

```
d.meanopacitymodel = ['dustcomponents',{'method':'simplemixing', \
                      'autosublim':True,'gasbelllin':True}]
```

8.3.4 Let us plot some of these mean opacity models

Let us give a few examples of how these mean opacities look. Let us make a disk model around a Herbig star with $M_* = 2 * M_\odot$, $T_* = 10^4$ K, and $L_* = 30 * L_\odot$, including olivine and water ice dust, and plot the various approximations of the mean opacities. We also overplot the full Bell & Lin opacity of Section 8.5.

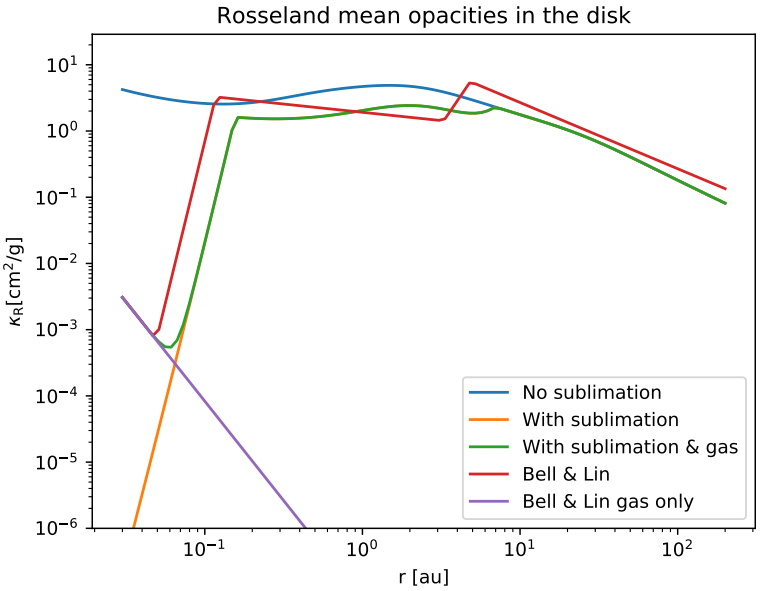
The code snippet is `snippet_plot_dustopac_5.py`. In Python run it as:

```
%run snippet_plot_dustopac_5.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, plt, MS, au, LS, finalize

dtg           = 0.01
abun_silicate = 0.5
abun_waterice = 0.5

# setup

d = DiskRadialModel(mdisk=0.01 * MS, mstar=2 * MS, tstar=1e4, lstar=30 * LS, rin=0.03 * au)
d.add_dust(agrain=0.1e-4, xigrain=3.6, dtg=dtg * abun_silicate)
d.add_dust(agrain=0.1e-4, xigrain=1.0, dtg=dtg * abun_waterice)
d.dust[0].grain.load_standard_opacity('ddn01', 'silicate')
d.dust[1].grain.load_standard_opacity('ddn01', 'waterice')
d.dust[0].grain.sublimationmodel = ['tsubfrompeq', {'species': 'MgFeSiO4', 'plaw': -24}]
d.dust[1].grain.sublimationmodel = ['tsubfrompeq', {'species': 'H2O', 'plaw': -10}]
for dust in d.dust:
    assert dust.grain.agrain == dust.agrain.max()
    assert dust.grain.xigrain == dust.xigrain.max()

# plotting

plt.figure()
d.meanopacitymodel = ['dustcomponents', {'method': 'simplemixing'}]
d.compute_mean_opacity()
plt.plot(d.r / au, d.mean_opacity_rosseland, label='No sublimation')

d.meanopacitymodel = ['dustcomponents', {'method': 'simplemixing', 'autosublim': True}]
d.compute_mean_opacity()
plt.plot(d.r / au, d.mean_opacity_rosseland, label='With sublimation')

d.meanopacitymodel = ['dustcomponents', {'method': 'simplemixing', 'autosublim': True, 'gasbelllin':
d.compute_mean_opacity()
plt.plot(d.r / au, d.mean_opacity_rosseland, label='With sublimation & gas')

d.meanopacitymodel = ['belllin']
d.compute_mean_opacity()
plt.plot(d.r / au, d.mean_opacity_rosseland, label='Bell & Lin')
```

```
d.meanopacitymodel = ['belllin', {'onlygas': True}]
d.compute_mean_opacity()
plt.plot(d.r / au, d.mean_opacity_rosseland, label='Bell & Lin gas only')

plt.xscale('log')
plt.yscale('log')
plt.xlabel('r [au]')
plt.ylabel(r'$\kappa_{\mathrm{R}} [\mathrm{cm}^2/\mathrm{g}]$')
plt.legend(loc='lower right')
plt.ylim(bottom=1e-6)
plt.title('Rosseland mean opacities in the disk')
plt.savefig('fig_snippet_plot_dustopac_5_1.pdf')

finalize()
```



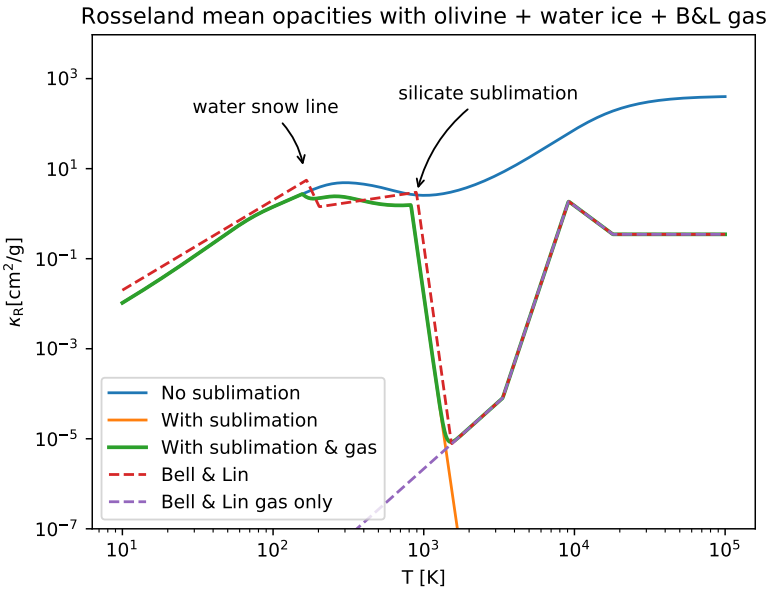
Here we have chosen a powerlaw smoothing of the olivine sublimation front of -24, consistent with the Bell & Lin opacity. For the water snow line (the H_2O sublimation) we have chosen the powerlaw less steep (more smooth): -10, which accounts for the gentle opacity transition near the snow line around 7 au. Remember that more gentle snow line phase transitions are easier for a numerical code to handle.

Note: This disk model does *not* include the "dust inner rim" (see Dullemond, Dominik & Natta 2001 ApJ 560, 957). See Section 12.2 for some notes about the dust inner rim. In the above model the dust sublimation radius is around 0.1 au. If we would include the inner rim physics, then the sublimation of dust would probably move to a larger radius (see Dullemond, Dominik & Natta 2001 ApJ 560, 957).

Perhaps it would be nicer to plot the mean opacities as a function of temperature. Let us do this here, for a gas density of $\rho_{\rm gas} = 10^{-10}\,{\rm g/cm^3}$. We can make direct use of the routines in `meanopacity.py` (which are used by `diskradial.py`).

The code snippet is `snippet_plot_dustopac_6.py`. In Python run it as:

```
%run snippet_plot_dustopac_6.py
```



You can see that the opacity jump at the water snow line is much weaker than for the Bell & Lin opacity model. This is because Bell & Lin chose a water ice abundance that is much higher than the 50% by mass that we take in our calculations.

8.4 Tabulated mean opacities

DISKLAB allows the user not only to include mean opacities from prescribed mean opacity models, but also allows the user to provide a completely user-defined table of mean opacities. The user should provide a table of $\kappa_P(\rho_g, T)$ and $\kappa_R(\rho_g, T)$ for a rectangular grid in the variables ρ_g and T.

Here is an example. Suppose we wish to implement Kramer's free-free opacity law:

$$\kappa_{\rm ff} = 3.68 \times 10^{22} g_{\rm ff}(1-Z)(1+X)\rho_g T^{-7/2} \tag{8.1}$$

which is the Rosseland mean. We take it, for convenience, also for the Planck mean. Here is how we implement it:

```
from disklab.diskradial import *
from disklab.natconst import *
d       = DiskRadialModel(mdisk=0.01*MS)
rhogas0 = 1e-20
rhogas1 = 1e-5
nrhogas = 5
rhogas  = rhogas0 * (rhogas1/rhogas0)**np.linspace(0.,1.,nrhogas)
temp0   = 1.e3
temp1   = 1.e6
ntemp   = 5
temp    = temp0 * (temp1/temp0)**np.linspace(0.,1.,ntemp)
rhog2d, temp2d = np.meshgrid(rhogas, temp, indexing='ij')
Z       = 0.
X       = 1.
gff     = 1.
kappa   = 3.68e22*gff*(1.-Z)*(1.+X)*rhog2d*temp2d**(-7./2.)
ln_kappa= np.log(kappa)
method  = 'linear'
d.meanopacitymodel = ['tabulated',{'rhogrid':rhogas,'tempgrid':temp, \
                                   'ln_kappa_planck':ln_kappa,       \
                                   'ln_kappa_rosseland':ln_kappa,    \
                                   'method':method}]
```

```
d.compute_mean_opacity()
```

The code until and including `kappa` is creating the 2-D table of mean opacity according to the Kramer's formula. We use here only a very coarse grid: 5 points in density and 5 points in temperature. If we would use linear interpolation in such a table, given that Kramer's opacity is very strongly dependent on temperature, we would get very bad interpolated values. However, by specifying `'ln_kappa_planck'` and `'ln_kappa_rosseland'` instead of `'kappa_planck'` and `'kappa_rosseland'` (which we could have done as well), we tell `compute_mean_opacity()` to do the interpolation with the logarithm of the values instead of with the actual values. Such log-interpolation is much more robust for functions that vary over many orders of magnitude. Hence we use this here.

Do not forget to call `d.compute_mean_opacity()` again, every time that `d.tmid[:]` or `d.sigma` changes. And if `d.sigma` changes, you must recompute `d.rhomid` using `d.compute_rhomid_from_sigma()`, because it is `d.rhomid` (together with `d.tmid`) that enter into the opacity calculation.

Note: The phrasing "tabulated mean opacity" is used in different contexts. In Chapter 7, where the wavelength-dependent opacities of dust grains are discussed, the mean opacities are computed self-consistently (see also Section 8.3), and – for speed – are also precalculated, tabulated and even written to a file. In contrast, the "tabulated mean opacity" used in this section are meant mostly for user-specified tables of mean opacities.

8.5 Bell & Lin mean opacities

Among disk radiation hydrodynamics modellers the opacity model of Bell & Lin (1994) ApJ 427, 987 is very popular. It is a simple analytic opacity model consisting of several powerlaw segments stitched together. It is valid for high temperatures, when the dust has vanished through sublimation, as well as for lower temperatures, where dust is present.

Using this opacity model as Planck- and Rosseland-mean opacity in DISKLAB is easy:

```
from disklab.diskradial import *
from disklab.natconst import *
d=DiskRadialModel(mdisk=0.01*MS)
d.meanopacitymodel = ['belllin']
d.compute_mean_opacity()
```

Do not forget to call `d.compute_mean_opacity()` again, every time that `d.tmid[:]` or `d.sigma` changes. And if `d.sigma` changes, you must recompute `d.rhomid` using `d.compute_rhomid_from_sigma()`, because it is `d.rhomid` (together with `d.tmid`) that enter into the opacity calculation.

Note that the Bell & Lin opacity table includes a gas-part and a dust-part. The dust part is valid only for the dust and ice abundances assumed by the Bell & Lin paper. It may not be consistent with the dust abundances you have in your model! You can choose the gas-only part by setting

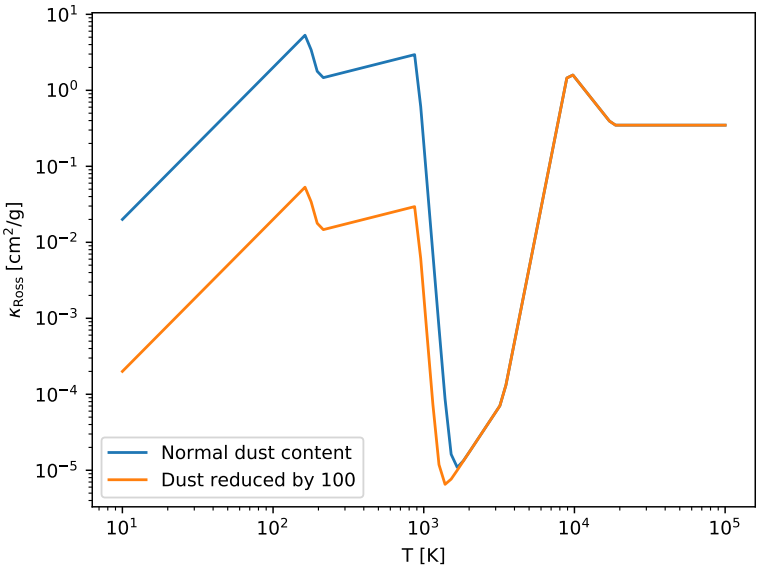
```
d.meanopacitymodel = ['belllin',{'onlygas':True}]
```

Or if you wish to simply reduce the dust content, you can do this by setting

```
d.meanopacitymodel = ['belllin',{'dustfactor':0.02}]
```

which reduces the dust content by a factor of 50 in this example. An example of Bell & Lin opacities with reduced dust content is given by snippet `snippet_plot_meanopac_belllin_1.py`. In Python run it as:

```
%run snippet_plot_meanopac_belllin_1.py
```



Note: The Bell & Lin dust opacity in this model is, however, very simplified. And there is no wavelength-dependent version of these opacities. That means that any post-processing of these models with wavelength-dependent radiative transfer (for instance, the RADMC-3D code) to obtain synthesized observations, will be difficult with the Bell & Lin model. Some modellers take radiation-hydrodynamics models made with the Bell & Lin opacities, and then post-process these with entirely different dust opacities for comparison to observations. This can be problematic. It is therefore perhaps more self-consistent to use the actual dust opacities (see Section 8.3 and Chapter 7). It is even possible to add the gas-only part of the Bell & Lin opacities to those dust opacities, to get more self-consistency.

8.6 Application of the mean opacities

This chapter dealt with the *mean* opacities of the disk. When are they necessary? They are necessary primarily for the computation of the disk thermal structure. The thermal structure, in turn, affects the disk vertical structure as well as the viscous evolution, though in both cases the effects are only moderate. The vertical scale height of the disk is "only" dependent on the square-root of the temperature, and the viscous evolution depends "only" linearly on temperature.

But let us experiment with the effect of such "realistic" mean opacities on the disk flaring and midplane temperature.

TO BE DONE: Flaring 1D, viscous heating.

Chapter 9

Simple radiative transfer model for computing (sub-)millimeter appearance

To compute how a protoplanetary disk looks at different wavelengths is a rather complex problem. It requires, in general, a full-fledged radiative transfer code such as the `RADMC-3D` code or similar codes (see Chapter 13).

However, for some cases one can make simple estimates without having to resort to the full-scale radiative transfer computation. This is, for instance, the case for thermal emission calculations in the millimeter wavelength regime. As long as the wavelength is long enough to be in the Rayleigh-Jeans regime of the Planck spectrum for the given disk midplane temperature, the resulting emission is not too sensitive to inaccuracies in the radiative transfer algorithm.

Let us make a simple estimate: Let us assume that the lowest temperature in the disk midplane is 10 K. The peak of the Planck curve is at about $\nu \simeq 3k_BT/h$, which is is about 625 GHz, or 0.5 millimeter wavelength. That means that for ALMA band 6 ($\lambda \simeq 1.3\,$mm) we are moderately in the Rayleigh-Jeans regime. The difference between $B_\nu(T)$ at $T = 10\,$K and $T = 20\,$K at this wavelength is a factor of 2.7, meaning that an error of a factor of 2 in our estimate of the temperature leads "only" to an error of a factor or 2.7 in the thermal emission. However, for ALMA band 9 ($\lambda \simeq 0.4\,$mm), the difference between $B_\nu(T)$ at $T = 10\,$K and $T = 20\,$K is a factor of 6.4, meaning that an error of a factor of 2 in our estimate of the temperature now leads to an error of a factor or 6.4 in the thermal emission.

What should we conclude from this? It means that when we study protoplanetary disks with longer wavelengths than about 1 mm (e.g. with the VLA), we are safe with the simple one-zone radiative transfer models that we will describe in this section. Also for ALMA bands it is ok, although for ALMA's higher-frequency bands we are on the border of the reliability of such simple radiative transfer models. For shorter wavelengths (far-infrared all the way down to the optical), however, there is no reliable simple radiative transfer recipe, and we are forced to use full radiative transfer tools such as `RADMC-3D`.

In the present section we will therefore describe a simple one-zone radiative transfer model that can be used for ALMA and VLA (and similar radio telescopes), but not for shorter wavelengths.

9.1 Specifying the wavelength-dependent dust opacity

For computing thermal dust emission from a disk we first need to have a model of the wavelength-dependent dust opacity. Such dust opacities are handled by the `GrainModel` class (see Chapter 7). Each dust component of the disk (`d.dust[0]`, `d.dust[1]`, ..., depending how many dust components you have added to the disk model `d`) should have its own grain model. You can check this:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
d.add_dust(agrain=1e-4,xigrain=3.6,dtg=0.1)
d.dust[0].grain
Out[]: <disklab.grainmodel.GrainModel at 0x10a82eb10>
```

In this example, since we specified `agrain` when calling `d.add_dust()`, such a grain model was automatically added to the dust component. If you specify, instead of `agrain`, the Stokes number `St`, then such a grain model will not be automatically added:

```
from disklab.diskradial import *
from disklab.natconst import *
d = DiskRadialModel(mdisk=0.01*MS)
d.add_dust(St=0.1)
d.dust[0].grain
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-6-8227c413dfd4> in <module>()
      3 d = DiskRadialModel(mdisk=0.01*MS)
      4 d.add_dust(St=0.1)
----> 5 d.dust[0].grain

AttributeError: 'DiskRadialComponent' object has no attribute 'grain'
```

The reason is simply that in this case there is no single grain size associated to this dust component, so no single grain model can be associated with it.

But let us go back to the first example, where we set `agrain=1e-4` (i.e. we specified the grain radius to be 1 μm), and a grain model is automatically associated to the dust component.

We can now ask the grain model object `d.dust[0].grain` to produce a simple analytic dust opacity for us. We do this by calling the method `grain.compute_simple_opacity()` and specify a wavelength:

```
lam = 0.13     # Wavelength 1.3 mm, which lies within band 6 of ALMA
d.dust[0].grain.compute_simple_opacity(lam)
```

This produces the following data:

```
d.dust[0].grain.opac_lammic
Out[]: array([ 1300.])
d.dust[0].grain.opac_kabs
Out[]: array([ 10.06920722])
```

which means that the dust opacity has been calculated to be $\kappa_{\mathrm{abs},\nu} = 10.069\,\mathrm{cm}^2/\mathrm{g}$, which is *per gram of dust*. The optical depth of the disk is defined as

$$\tau_\nu(r) = \Sigma_{\mathrm{dust}}(r)\,\kappa_\nu^{\mathrm{abs}} \tag{9.1}$$

So for our model (assuming that `d.dust[0]` is the only dust component) this would then be:

```
taudisk = d.dust[0].sigma * d.dust[0].grain.opac_kabs[0]
```

If you have multiple dust components:

```
lam = 0.13     # Wavelength 1.3 mm, which lies within band 6 of ALMA
tau = np.zeros_like(d.sigma)
for dust in d.dust:
   dust.grain.compute_simple_opacity(lam)
   tau += dust.sigma * dust.grain.opac_kabs[0]
```

9.2 One-zone radiative transfer model for dust emission

Once a dust opacity is set, you can compute the emerging intensity from the disk, assuming face-on inclination, and assuming a one-zone model (i.e. no "warm surface layer"). This is computed using the method

`compute_onezone_intensity()`:

$$I_\nu(r) = (1 - \exp(-\tau_\nu(r)))\, B_\nu(T) \tag{9.2}$$

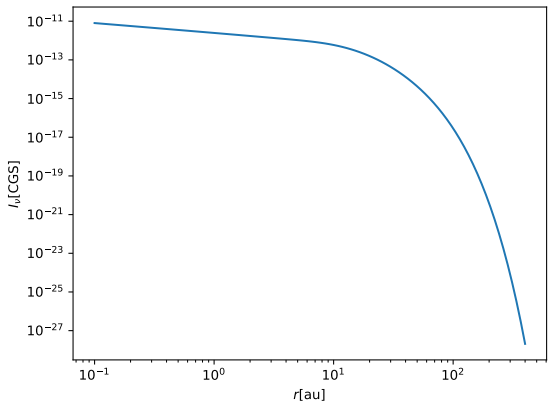
where $B_\nu(T)$ is the Planck function in the usual CGS units of erg/cm^2/s/Hz/ster. This is stored in `intensity`. One can also express the intensity as a brightness temperature using `compute_tbright_from_intensity()`.

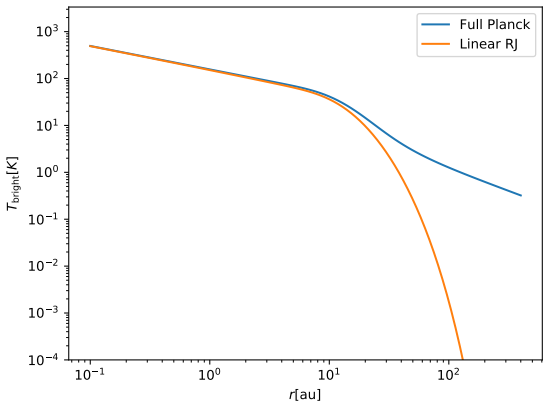
Example. The code snippet is `snippet_simple_rt_2.py`. In Python run it as:

```
%run snippet_simple_rt_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, plt, year, MS, au, finalize

d = DiskRadialModel(rout=400 * au)
d.make_disk_from_lbp_alpha(1e-2 * MS, 1 * au, 1e-2, 1e5 * year)
d.add_dust(agrain=1e-4)
lam = 0.13   # At 1.3 mm
d.dust[0].grain.compute_simple_opacity(lam, tabulatemean=False)
d.compute_onezone_intensity(lam)

plt.figure()
plt.plot(d.r / au, d.intensity[0, :])
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$I_\nu [\mathrm{CGS}]$')
plt.xscale('log')
plt.yscale('log')

print('Flux at one pc distance = {} Jy'.format(d.flux_at_oneparsec / 1e-23))
d.compute_tbright_from_intensity()
tbright_fullplanck = d.tbright.copy()
d.compute_tbright_from_intensity(linear=True)

plt.figure()
plt.plot(d.r / au, tbright_fullplanck[0, :], label='Full Planck')
plt.plot(d.r / au, d.tbright[0, :], label='Linear RJ')
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$T_{\mathrm{bright}} [K]$')
plt.xscale('log')
plt.yscale('log')
plt.ylim(bottom=1e-4)
plt.legend()

finalize(results=(tbright_fullplanck[0, :],d.tbright[0, :]))
```





9.3 Including "real" dust opacities

[TIL: WILL YOU WORK ON THE REAL OPACITY MODEL?]

So far we simply "guessed" the dust opacity. But the `GrainModel` class has methods for reading dust opacity data. In the `opacity/` directory there is a file called `dustkappa_silicate.inp` (a copy of which is also in the `snippets/` directory).

MOVE THIS STUFF

INCLUDE HOW TO INCLUDE SUBLIMATION

Chapter 10

Advanced applications of the DiskRadialModel class

Here we give a few more sophisticated examples of how the `DiskRadialModel` class can be used to solve problems in astrophysics. All these models are stand-alone code snippet files that can be run from within Python using the `%run` command with the snippet file name. Or you can directly call Python from the command line with the code snippet file name added.

10.1 Radial mixing of crystalline silicates

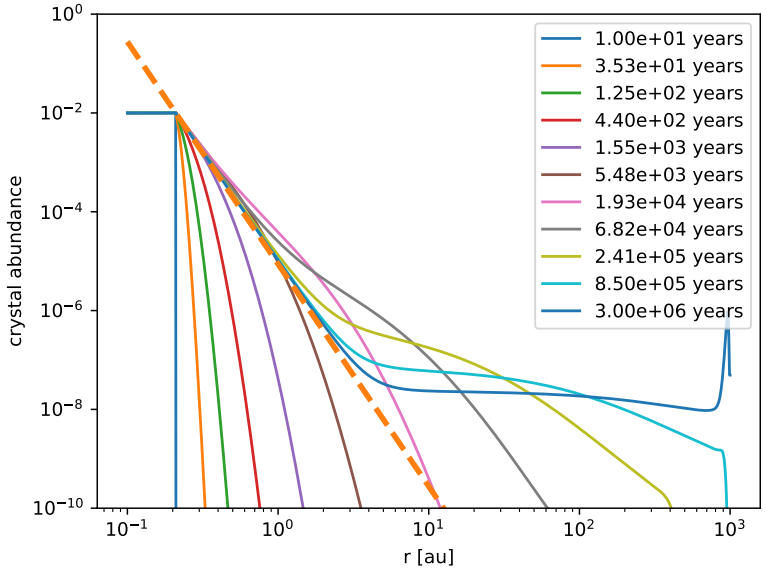
A model of full disk evolution with a massless dust component passively moving along, but continously set to the dust-to-gas ratio inward of the point where the temperature is higher than the crystallization temperature. The code snippet is `snippet_radialmix_crystals_1.py`. In Python run it as:

```
%run snippet_radialmix_crystals_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, au, LS, year, finalize
import copy
St     = 1e-10       # Let's take infinitely small dust
tstart = 1e1 * year
tend   = 3e6 * year
ntime  = 100
nr     = 1000
dtg    = 0.01
tcryst = 800.
time   = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))

d = DiskRadialModel(mstar=2.4 * MS, lstar=30 * LS, tstar=1e4, rout=1000 * au, nr=nr)
d.make_disk_from_simplified_lbp(1e3, 3 * au, 1)
d.add_dust(St=St)
d.dust[0].sigma[:] = 1e-20 * d.sigma[:]   # Reset: Make everything 'amorphous'
d.dust[0].sigma[d.tmid > tcryst] = dtg * d.sigma[d.tmid > tcryst]  # 'Crystallize' hot part
d.Sc = 3.0                        # Set the Schmidt number
dlist = [copy.deepcopy(d)]
ircryst = np.where(d.tmid > tcryst)[0][-1]
for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    extracond = [(0., 1., dtg, 1, ircryst)]
    d.compute_viscous_evolution_and_dust_drift_next_timestep(
        dt, extracond=extracond)
```

```
    dlist.append(copy.deepcopy(d))
plt.figure()
for itime in range(0, ntime + 1, 10):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].dust[0].sigma / dlist[itime].sigma, label=s)
plt.plot(d.r / au, dtg * (d.r[ircryst] / d.r)**(1.5 * d.Sc), linewidth=3, linestyle='--')
plt.xscale('log')
plt.yscale('log')
plt.ylim(1e-10, 1e0)
plt.xlabel('r [au]')
plt.ylabel('crystal abundance')
plt.legend()

finalize(results=(dlist[-1].dust[0].sigma))
```



The different lines are different time snapshots, with logarithmic time intervals.

The crystallization is implemented directly inside the implicit differencing code. We make use of the "internal condition" method, where we replace the differential equation with a condition similar to a boundary condition. We do this at the largest radius where the $T > T_{\rm cryst}$. That is sufficient, because inside of that point everything has passed through that point (except at the very beginning, which is why at the start we "crystallize" by hand).

Overplotted as a dashed line is the analytic powerlaw from Clarke & Pringle (1988) MNRAS 235, 365 (see also Pavlyuchenkov & Dullemond 2007, A&A 471, 833). Note that to obtain the correct powerlaw (as shown in this example), the model requires indeed about 1000 radial gridpoints (i.e. 250 gridpoints per decade in radius). With 100 gridpoints, as you can verify yourself, you will find that the mixing is too strong.

Another thing that can be seen is that there is a period where the mixing outward appears to be more efficient that the analytic slope. That is because the whole disk is still expanding in this early phase.

Note that the example of crystalline sillicates mixing here is done with *one* dust species. But this can be also done with the multi-dust-species methods (see Section 6.7). So let's do *exactly the same* but now with the multi-species method. The code snippet is `snippet_radialmix_crystals_2.py`. In Python run it as:

```
%run snippet_radialmix_crystals_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, au, year, LS, finalize, DiskRadialComponent
import copy
```

```
St     = 1e-10       # Let's take infinitely small dust
tstart = 1e1 * year
tend   = 3e6 * year
ntime  = 100
nr     = 1000
dtg    = 0.01
tcryst = 800.
nspec  = 2
time   = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))

d = DiskRadialModel(mstar=2.4 * MS, lstar=30 * LS, tstar=1e4, rout=1000 * au, nr=nr)
d.make_disk_from_simplified_lbp(1e3, 3 * au, 1)
d.dust = []
d.dust.append(DiskRadialComponent(d, St=St, sigma=dtg * d.sigma))
d.dust.append(DiskRadialComponent(d, St=St, sigma=1e-20 * d.sigma))
d.dust[0].sigma[d.tmid > tcryst] = 1e-20 * d.sigma[d.tmid > tcryst]  # 'Crystallize' hot part
d.dust[1].sigma[d.tmid > tcryst] = dtg * d.sigma[d.tmid > tcryst]    # 'Crystallize' hot part
d.Sc = 3.0  # Set the Schmidt number

# iteration

dlist = [copy.deepcopy(d)]
ircryst = np.where(d.tmid > tcryst)[0][-1]
for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.compute_viscous_evolution_next_timestep(dt)
    extracond = [(0., 1., 0., 1, ircryst)]
    d.dust[0].compute_dust_radial_drift_next_timestep(dt, extracond=extracond)
    extracond = [(0., 1., dtg, 1, ircryst)]
    d.dust[1].compute_dust_radial_drift_next_timestep(dt, extracond=extracond)
    dlist.append(copy.deepcopy(d))

# plotting

plt.figure()
for itime in range(0, ntime + 1, 10):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].dust[1].sigma / dlist[itime].sigma, label=s)

plt.plot(d.r / au, dtg * (d.r[ircryst] / d.r)**(1.5 * d.Sc), linewidth=3, linestyle='--')
plt.xscale('log')
plt.yscale('log')
plt.ylim(1e-10, 1e0)
plt.xlabel('r [au]')
plt.ylabel('crystal abundance')
plt.legend()

finalize(results=(dlist[-1].dust[1].sigma))
```

We do not show the plot, because it is exactly the same as above. The advantage of the multi-species version of this code snippet is that this one is easier to generalize to many-species models. For instance, if you want to do more complex chemistry of the dust in the disk.

10.1.1 Crystallization: Why not use operator-splitting?

We implemented the crystallization condition inside the implicit differencing code for the transport (=mixing and drift). Why? Wouldn't it be much more straightforward to use the method of 'operator splitting'? Operator splitting is the method in which different processes happening at the same time in reality are handled one-after-the-other in the numerical scheme. Here we could consider doing the dust mixing step first, and then make all the dust

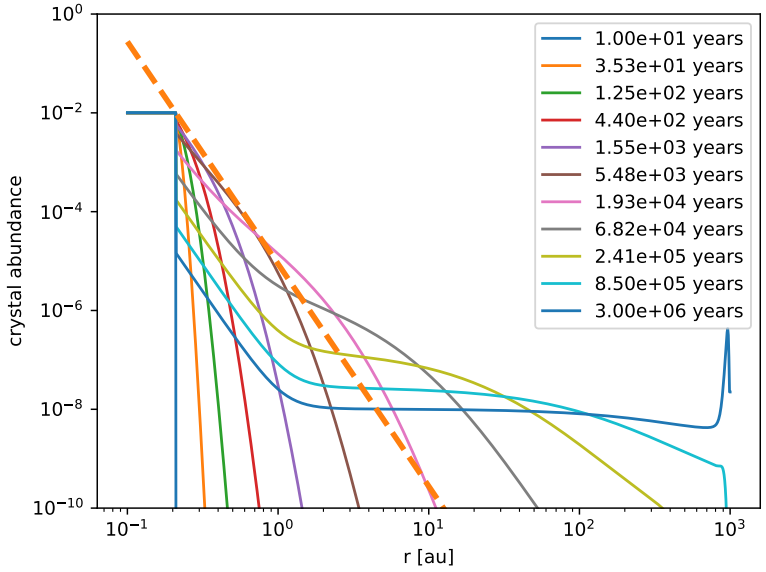
crystalline where the temperature is higher than 800 K, and then continue to the next time step, where we again first to transport, then crystallization. Let us try this out and see what happens. Let us replace in the above examples the internal condition with simply setting the dust to crystalline in the inner (hot) regions. The code snippet is `snippet_radialmix_crystals_3.py`. In Python run it as:

```
%run snippet_radialmix_crystals_3.py
```

Here is the listing:

```
# WARNING: THIS IS AN EXAMPLE OF WHAT CAN GO WRONG WITH RADIAL MIXING
from snippet_header import DiskRadialModel, np, plt, MS, au, year, LS, finalize, DiskRadialComponent
import copy

St     = 1e-10       # Let's take infinitely small dust
tstart = 1e1 * year
tend   = 3e6 * year
ntime  = 100
nr     = 1000
dtg    = 0.01
tcryst = 800.
nspec  = 2
time   = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))

# setup

d = DiskRadialModel(mstar=2.4 * MS, lstar=30 * LS, tstar=1e4, rout=1000 * au, nr=nr)
d.make_disk_from_simplified_lbp(1e3, 3 * au, 1)
d.dust = []
d.dust.append(DiskRadialComponent(d, St=St, sigma=dtg * d.sigma))
d.dust.append(DiskRadialComponent(d, St=St, sigma=1e-20 * d.sigma))
d.dust[0].sigma[d.tmid > tcryst] = 1e-20 * d.sigma[d.tmid > tcryst]  # 'Crystallize' hot part
d.dust[1].sigma[d.tmid > tcryst] = dtg * d.sigma[d.tmid > tcryst]    # 'Crystallize' hot part
d.Sc = 3.0                       # Set the Schmidt number
dlist = [copy.deepcopy(d)]
ircryst = np.where(d.tmid > tcryst)[0][-1]

# iteration

for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.compute_viscous_evolution_next_timestep(dt)
    d.dust[0].compute_dust_radial_drift_next_timestep(dt)
    d.dust[1].compute_dust_radial_drift_next_timestep(dt)
    sigdust = d.dust[0].sigma + d.dust[1].sigma
    # 'Crystallize' hot part
    d.dust[0].sigma[d.tmid > tcryst] = 1e-20
    d.dust[1].sigma[d.tmid > tcryst] = sigdust[d.tmid > tcryst]  # 'Crystallize' hot part
    dlist.append(copy.deepcopy(d))

# plotting

plt.figure()
for itime in range(0, ntime + 1, 10):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].dust[1].sigma /
             dlist[itime].sigma, label=s)

plt.plot(d.r / au, dtg * (d.r[ircryst] / d.r) **
         (1.5 * d.Sc), linewidth=3, linestyle='--')
plt.xscale('log')
plt.yscale('log')
plt.ylim(1e-10, 1e0)
```

```
plt.xlabel('r [au]')
plt.ylabel('crystal abundance')
plt.legend()

finalize(results=(dlist[-1].dust[1].sigma))
```



You can see that the results are strange: the crystal-abundance makes a jump right after the crystallized part. This is clearly not correct. Let us analyze why.

The operator splitting method would work fine *if* we use small enough time steps. The time step must be small enough that the mixing step does not change the dust surface density by more than a few tens of percent at most. Or to put it more scientifically: The time step must obey the Courant-Friedrichs-Lewy (CFL) condition. Then the transport and the crystallization both modify the dust only by modest amounts and the results will be reasonably accurate.

However, because we use a super-stable transport scheme (implicit differencing) we usually take super-large time steps. At the end of the model the time steps are about a 1000× larger than the CFL condition at the inner edge of the disk. If the transport scheme were an explicit differencing scheme, the model would have already blown up long ago. By virtue of the implicit differencing the radial drift and mixing method still works stably for these super-large time steps.

However, if we add new physics (here: crystallization) using the operator-splitting method (and thus not included in the implicit integration scheme), then the implicit drift-and-mixing method will not treat this physics. For the entire time step it thinks that this physics is not included. Only at the end of the time step, when the crystallization is carried, the inner disk is re-crystallized. However, by then the implicit differencing time step has already shifted almost all the crystals into the inner boundary. This means that the way we calculate the abundance of crystals is no longer accurate. *This is therefore an example of how the radial mixing and drift calculations can go wrong if not done carefully*.

One solution (other than using the internal condition) could be to use very small time steps. But that is of course numerically very costly.

The problem we demonstrate here is a general problem when using an implicitly differenced transport scheme in combination with another method in an operator-splitted way (one step transport, one step other method, then one step transport again and one step other method again etc). One can only take large time steps if one does *all* the processes within a single implicit scheme.

This becomes tricky if you couple radial drift and mixing of a multi-dust-species problem to chemistry. The chemistry may couple the various dust species to each other at each location, while the transport will move them around. How can this be done? Typically it would simply mean that you would have to reduce the time step enormously (and

thus increase the computational cost enormously). If the chemistry rates are small enough (i.e. the chemical time steps are long enough) then you only have to ensure that the time steps are small enough to handle the chemistry explicitly. But at any rate: the time steps may become substantially smaller than we usually use.

10.2 Ring-shaped dust traps

In the paper by Pinilla, Birnstiel, Ricci, Dullemond, Uribe, Testi, & Natta (2012) A&A 538, 114, a simple bumpy disk model was implemented to see how the dust could be trapped by these bumps. That model also included the dust coagulation. Let us here do something similar, but without the dust growth. First, we want to make a "sine wave" with varying wavelength, such that the local wavelength is always a constant factor ζ times the local pressure scale height:

$$\lambda_{\rm wave}(r) = \zeta H_p(r) \tag{10.1}$$

We can numerically compute the phase $\Phi(r)$ by numerical integration of the following equation:

$$\frac{d\Phi(r)}{dr} = \frac{2\pi}{\lambda_{\rm wave}(r)} \tag{10.2}$$

which gives:

$$\Phi(r) = \Phi_0 + \int_{r_0}^{r} \frac{2\pi}{\lambda_{\rm wave}(r')} dr' \tag{10.3}$$

The wave is then the following dimensionless function:

$$w(r) = A \sin(\Phi(r)) \tag{10.4}$$

We could use this directly for making the surface density profile:

$$\Sigma_g(r) = \Sigma_{g1}(r)\,(1 + w(r)) \tag{10.5}$$

where $\Sigma_{g1}(r)$ is the unperturbed surface density from one of the models of Chapter 3. However, if we then let the disk viscously evolve, the bumps will quickly disappear again.

Instead we apply the wave to the viscosity. To allow experimentation with large amplitudes, we apply it in log-space, i.e. we add it to $\ln \nu(r)$, or equivalently to $\ln \alpha(r)$:

$$\ln \alpha(r) = \ln \alpha_1(r) + A \sin(\Phi(r)) \tag{10.6}$$

where $\alpha_1(r)$ is the unperturbed value of α, which is usually taken to be constant with r but could equally well be r-dependent.

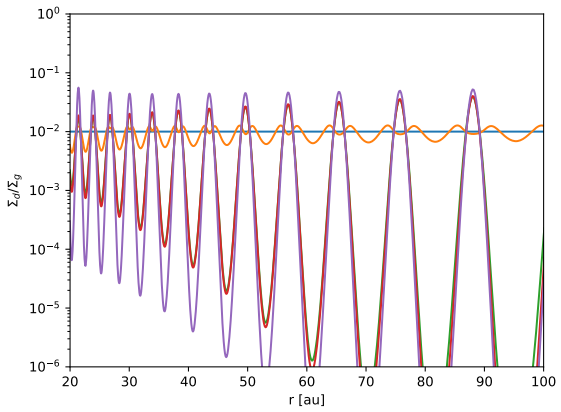
Here is a setup example. We use the combined gas and dust evolution method of `DiskRadialModel`. The code snippet is `snippet_rings_dusttrap_1.py`. In Python run it as:

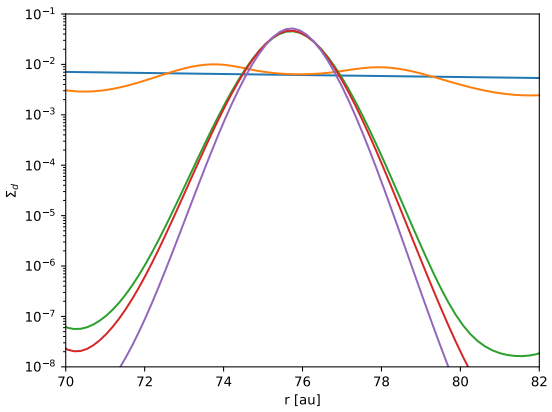
```
%run snippet_rings_dusttrap_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, year, au, finalize
import copy

tstart    = 1e1 * year
tend      = 3e6 * year
ntime     = 100
nr        = 2000
alpha     = 1e-2
agrain    = 1e-2
zeta      = 2.0
ampl      = 1.0
time      = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))

# setup

d = DiskRadialModel(rin=10 * au, rout=1000 * au, nr=nr, alpha=alpha)
d.make_disk_from_simplified_lbp(1e0, 100 * au, 1)
d.add_dust(agrain=agrain)
```

```
lamwav = zeta * d.hp
integrand = 2 * np.pi / lamwav
integrand = 0.5 * (integrand[1:] + integrand[:-1])
phase = np.zeros(nr)
for ir in range(1, nr):
    phase[ir] = phase[ir - 1] + integrand[ir - 1] * (d.r[ir] - d.r[ir - 1])
wave = ampl * np.sin(phase)
d.alpha *= np.exp(wave)
dlist = [copy.deepcopy(d)]

# iteration

for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.compute_viscous_evolution_and_dust_drift_next_timestep(dt)
    dlist.append(copy.deepcopy(d))

# plotting

plt.figure()
for itime in range(0, ntime, 20):
    plt.plot(dlist[itime].r / au,
             dlist[itime].dust[0].sigma / dlist[itime].sigma)
plt.yscale('log')
plt.xlim(20, 100)
plt.ylim(1e-6, 1e0)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_d/\Sigma_g$')

plt.figure()
for itime in range(0, ntime, 20):
    plt.plot(dlist[itime].r / au, dlist[itime].dust[0].sigma)
plt.yscale('log')
plt.xlim(70, 82)
plt.ylim(1e-8, 1e-1)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_d$')

finalize(results=(dlist[-1].dust[0].sigma))
```





In the first figure the dust-to-gas ratio is shown. This model shows how the dust gets trapped and concentrated in to thin rings. In the second figure a zoom-in on one of these rings is shown, this time the actual dust surface density

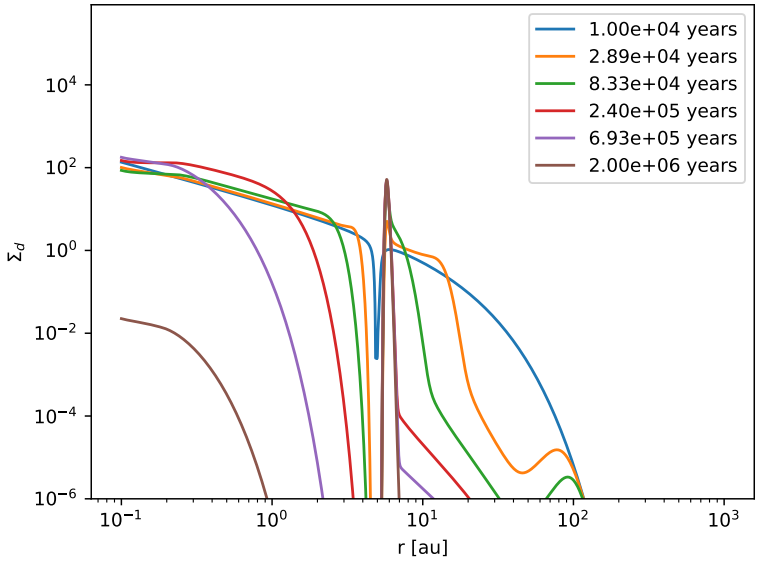
10.3 Simple planetary gap model with dust drift

A simple model of a gap produced by a planet is presented by Duffell (2015) ApJL 807, 11. Let us see how the dust drift reacts to that. The code snippet is in `snippet_planetgap_dustdrift_1.py`. In Python run it as:

```
%run snippet_planetgap_dustdrift_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, au, finalize, year, Mea
import copy
tstart    = 1e4 * year
tend      = 2e6 * year
ntime     = 100
nr        = 2000
alpha     = 1e-4
agrain    = 1e-1
mplanet   = 100 * Mea
aplanet   = 5 * au
mdisk0    = 1e-2 * MS
rdisk0    = 10 * au

d = DiskRadialModel(rin=0.1 * au, rout=1000 * au, nr=nr)
d.make_disk_from_lbp_alpha(mdisk0, rdisk0, alpha, tstart)
d.add_dust(agrain=agrain)
d.alphamix = d.alpha      # Make separate alpha for mixing
d.add_planet_gap(aplanet, 'duffell', mpl=mplanet, smooth=2., log=True, innu=True)

# iteration

time = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))
dlist = [copy.deepcopy(d)]
for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.compute_viscous_evolution_and_dust_drift_next_timestep(dt)
    dlist.append(copy.deepcopy(d))

# plotting

plt.figure()
for itime in range(0, ntime + 1, 20):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].dust[0].sigma, label=s)
plt.xscale('log')
plt.yscale('log')
plt.ylim(bottom=1e-6)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_d$')
plt.legend()

finalize(results=(dlist[-1].dust[0].sigma))
```



You see here that the dust rapidly drifts toward the planet gap in the outer disk. There it gets trapped, as expected. The dust in the inner disk is drifting much more slowly, because the Stokes number of these grains is much smaller there. The model has a fixed grain size, so that the grains drift slower and slower as they get further inward. They eventually just drift with the gas.

It is important to keep in mind that in this model the planetary gap is included as a change in viscosity parameter α. This means that the gas radial velocity v_r has a strong increase in the gap. This may drag along the dust. Also, it would mean that the dust mixing coefficient is also increased, which is not physical. To prevent this, the above listing creates a `self.alphamix` as a copy of `self.alpha` *before* adding the planetary gap. The dust drift/mixing method automatically checks if `self.alphamix` is present. If yes, then it will use that value of α instead of the viscous α, for the computation of the mixing coefficient.

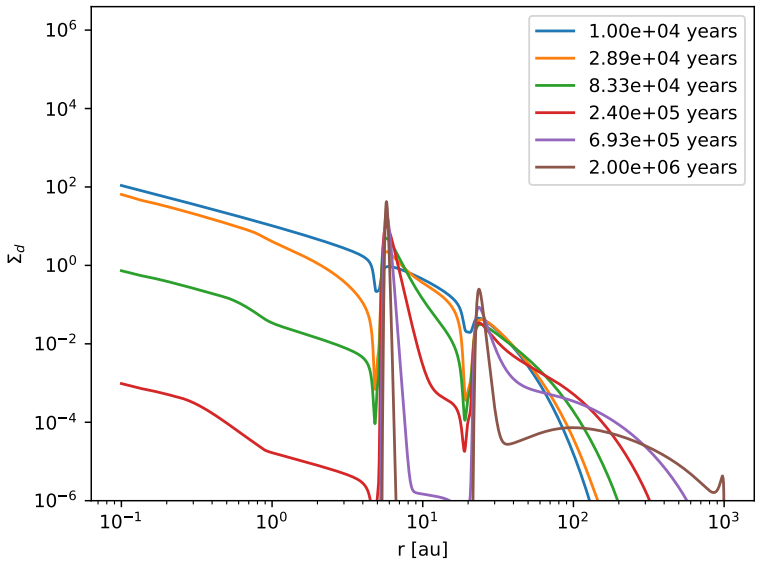
Here is another example. We now add two planets, we keep the Stokes number fixed and tweak a few other parameters. In Python run it as:

```
%run snippet_planetgap_dustdrift_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, np, plt, MS, au, finalize, Mea, year
import copy

tstart    = 1e4 * year
tend      = 2e6 * year
ntime     = 100
nr        = 2000
alpha     = 1e-2
St        = 0.01
mplanet1  = 100 * Mea
aplanet1  = 5 * au
mplanet2  = 200 * Mea
aplanet2  = 20 * au
mdisk0    = 1e-2 * MS
rdisk0    = 10 * au

# setup

d = DiskRadialModel(rin=0.1 * au, rout=1000 * au, nr=nr)
d.make_disk_from_lbp_alpha(mdisk0, rdisk0, alpha, tstart)
d.add_dust(St=St)
```

```
d.alphamix = d.alpha      # Make separate alpha for mixing
d.add_planet_gap(aplanet1, 'duffell', mpl=mplanet1, smooth=2., log=True, innu=True)
d.add_planet_gap(aplanet2, 'duffell', mpl=mplanet2, smooth=2., log=True, innu=True)

# iteration

time = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))
dlist = [copy.deepcopy(d)]
for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.compute_viscous_evolution_and_dust_drift_next_timestep(dt)
    dlist.append(copy.deepcopy(d))

# plotting

plt.figure()
for itime in range(0, ntime + 1, 20):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].dust[0].sigma, label=s)
plt.xscale('log')
plt.yscale('log')
plt.ylim(bottom=1e-6)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_d$')
plt.legend()

finalize(results=(dlist[-1].dust[0].sigma))
```



One can see the dust being trapped at both planetary gaps, and inside of the gaps the dust disappears entirely. You can now use the technique of Section 9.2 to make some radiative transfer maps for ALMA for instance.

10.4 Formation of a dead zone

Let us make a simple example of a dead zone, but one that "suddenly appears": we start from a normal disk, and set the α parameter to a low value within a certain region. This is an unphysical starting condition, because the disk does not suddenly get a dead zone, but it is what is often done in 2-D hydrodynamic disk models, so let us see what the viscous evolution and dust drift say.

We set up a model with normal $\alpha = 10^{-2}$, and deadzone $\alpha_{\rm dead} = 10^{-4}$ (i.e. not so "dead" as is often assumed). The inner edge of the dead zone is taken at 0.4 au and the outer edge at 5 au (both values are rather arbitrary).

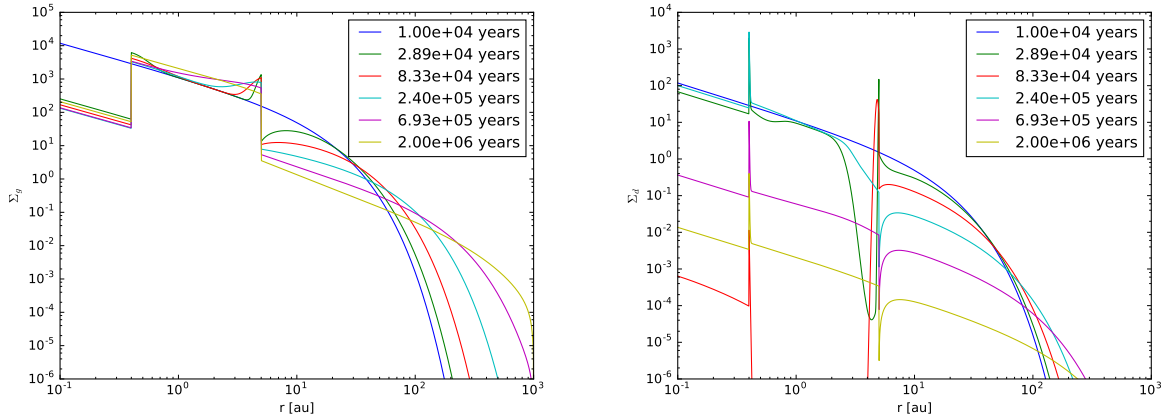
The code snippet is `snippet_deadzone_1.py`. In Python run it as:

```
%run snippet_deadzone_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, year, au, MS, plt, np, finalize
import copy

tstart   = 1e4 * year
tend     = 2e6 * year
ntime    = 1000
nr       = 2000
alpha1   = 1e-2
alpha0   = 1e-4
rdeadin  = 0.4 * au
rdeadout = 5 * au
St       = 0.01
mdisk0   = 1e-2 * MS
rdisk0   = 10 * au

d = DiskRadialModel(rin=0.1 * au, rout=1000 * au, nr=nr)
d.make_disk_from_lbp_alpha(mdisk0, rdisk0, alpha1, tstart)
d.add_dust(St=St)
d.alpha = np.zeros(nr) + d.alpha
d.alpha[np.logical_and((d.r > rdeadin), (d.r < rdeadout))] = alpha0

time = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))
dlist = [copy.deepcopy(d)]
for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.compute_viscous_evolution_and_dust_drift_next_timestep(dt, updatestokes=False)
    dlist.append(copy.deepcopy(d))

plt.figure()
for itime in range(0, ntime + 1, ntime // 5):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].sigma, label=s)
plt.xscale('log')
plt.yscale('log')
plt.ylim(bottom=1e-6)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_g$')
plt.legend()

plt.figure()
for itime in range(0, ntime + 1, ntime // 5):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].dust[0].sigma, label=s)
plt.xscale('log')
plt.yscale('log')
plt.ylim(bottom=1e-6)
```

```
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_d$')
plt.legend()

finalize(results=(dlist[itime].sigma,dlist[-1].dust[0].sigma))
```



Note that we took $10\times$ smaller time steps than in most other examples, to ensure that the dust trapping behavior is properly modelled at the inner dead zone edge. That means that the code is $10\times$ slower in this example, but it should not take more than a minute.

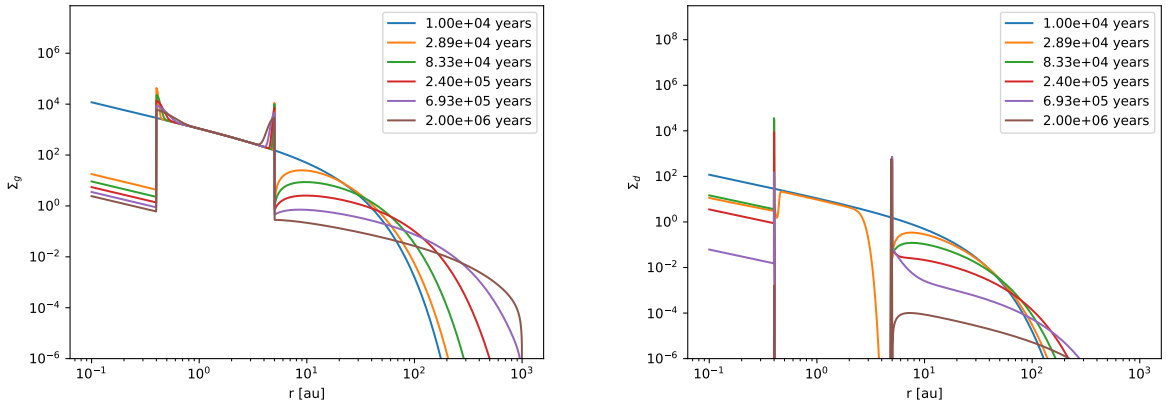
You can see a number of things in this example: At the start of the simulation there is a sudden accumulation of gas mass at the outer edge of the dead zone. This is to be expected, because there is continuous feeding of mass from the outside, but the deadzone cannot transport this mass as quickly as it is fed. Then this mass is slowly but gradually transported through the deadzone into the inner disk. Eventually a (semi-)steady state is reached in which $\Sigma_g \nu$ is (semi-)constant again, meaning that Σ_g is higher in the deadzone than outside by a factor of $\alpha/\alpha_{\rm dead}$.

Another interesting thing happens at the inner edge. There there is, initially, also an accumulation of gas mass. This is because the viscously active inner disk is transporting mass *outward* (as if it were a small 0.4 au size spreading disk). Only later does this bump disappear because the system wants to achieve (semi-)steady state with $\Sigma_g \nu$ constant.

The dust also behaves interestingly. *Note that we took the Stokes number fixed to* $\mathrm{St} = 10^{-2}$. *To ensure that the code does not accidently update* St *from the grain size at each time step and radius, we set* `updatestokes=False`. What happens is the following: Initially the dust gets trapped at the outer deadzone edge because the gas bump is occurring there. Meanwhile the dust in the deadzone itself is drifting inwards (see curves at around 2.9×10^4 and 8.3×10^4 years). But as this gas bump is spreading inward, the dust is released (around 2.4×10^5 years) leading to a sudden pile-up of dust at the inner edge of the dead zone (where there is a long-lived bump).

Note that these results are for a relatively "mild" dead zone. If, however, the $\alpha_{\rm dead} = 10^{-6}$, the semi-steady state in the deadzone is never reached. The code snippet `snippet_deadzone_2.py` is the same as before, but only with $\alpha_{\rm dead} = 10^{-6}$. In Python run it as:

```
%run snippet_deadzone_2.py
```



Note that in both example models the deadzone edges are ultra-sharp. That is unrealistic (if the above model is realistic in the first place). One may want to smooth these edges.

Another, more physical way to implement the deadzone idea is to go back to the original two-layered disk model of Gammie (1996) ApJ 457, 355. The *DISKLAB* package does not explicitly include a two-layered accretion model, but there is a simply way to mimic this: by vertically averaging the $\alpha(r,z)$:

$$\alpha(r) = \begin{cases} \alpha_1 & (r < r_{\rm in}) \\ \alpha_1(\Sigma_{\rm active}/\Sigma_g) & (\Sigma_g > \Sigma_{\rm active}) \\ \alpha_1 & (\Sigma_g \le \Sigma_{\rm active}) \end{cases} \tag{10.7}$$

where α_1 is the normal active alpha value, $r_{\rm in}$ is the inner edge of the dead zone (where we assume that the temperature of the disk is high enough to thermally ionize alkali metals; we simply fix it here to the same value as the inner deadzone edge radius as in the above examples).

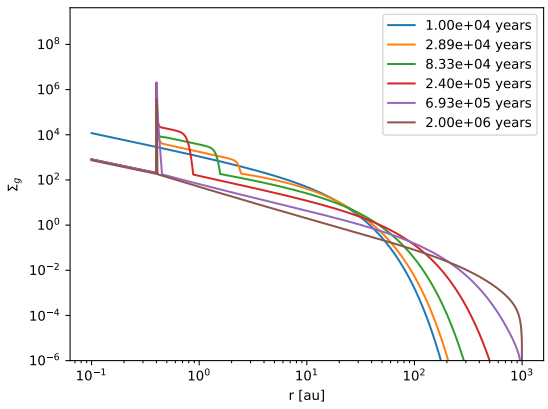
The code snippet is `snippet_deadzone_3.py`. In Python run it as:

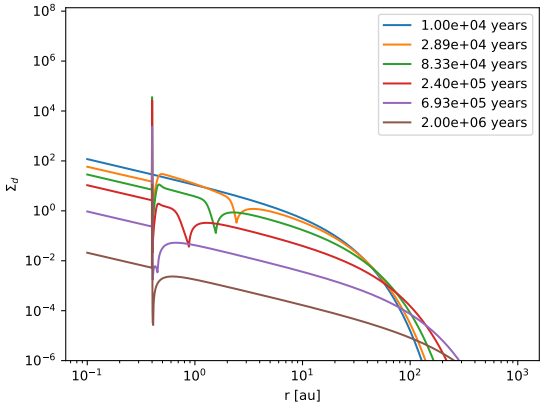
```
%run snippet_deadzone_3.py
```

Here is the listing:

```
from snippet_header import np, plt, DiskRadialModel, MS, au, year, finalize
import copy

tstart    = 1e4 * year
tend      = 2e6 * year
ntime     = 1000
nr        = 2000
alpha1    = 1e-2
alphamin  = 1e-6
rdeadin   = 0.4 * au
siglive   = 2e2    # Twice Gammie's penetration depth of cosmic rays
St        = 0.01
mdisk0    = 1e-2 * MS
rdisk0    = 10 * au

# setup

d = DiskRadialModel(rin=0.1 * au, rout=1000 * au, nr=nr)
d.make_disk_from_lbp_alpha(mdisk0, rdisk0, alpha1, tstart)
d.add_dust(St=St)
d.alpha = np.zeros(nr) + d.alpha

# iteration

time = tstart * (tend / tstart)**(np.linspace(0., 1., ntime + 1))
```

```
dlist = [copy.deepcopy(d)]
for itime in range(1, ntime + 1):
    dt = time[itime] - time[itime - 1]
    d.alpha[:] = alpha1
    d.alpha[d.sigma > siglive] *= siglive / d.sigma[d.sigma > siglive]
    d.alpha[d.r < rdeadin] = alpha1
    d.alpha[d.alpha < alphamin] = alphamin
    d.compute_viscous_evolution_and_dust_drift_next_timestep(
        dt, updatestokes=False)
    dlist.append(copy.deepcopy(d))

# plotting

plt.figure()
for itime in range(0, ntime + 1, ntime // 5):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].sigma, label=s)
plt.xscale('log')
plt.yscale('log')
plt.ylim(bottom=1e-6)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_g$')
plt.legend()

plt.figure()
for itime in range(0, ntime + 1, ntime // 5):
    s = '{0:8.2e} years'.format(time[itime] / year)
    plt.plot(dlist[itime].r / au, dlist[itime].dust[0].sigma, label=s)
plt.xscale('log')
plt.yscale('log')
plt.ylim(bottom=1e-6)
plt.xlabel('r [au]')
plt.ylabel(r'$\Sigma_d$')
plt.legend()

finalize()
```



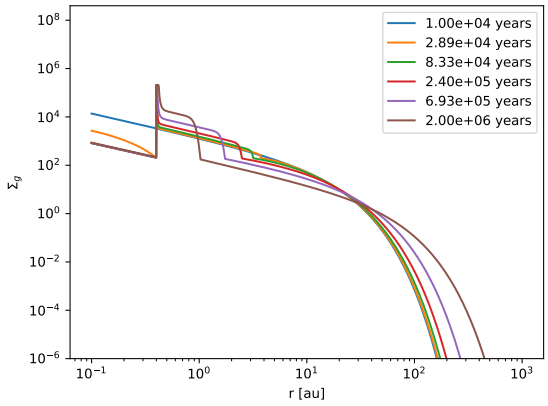


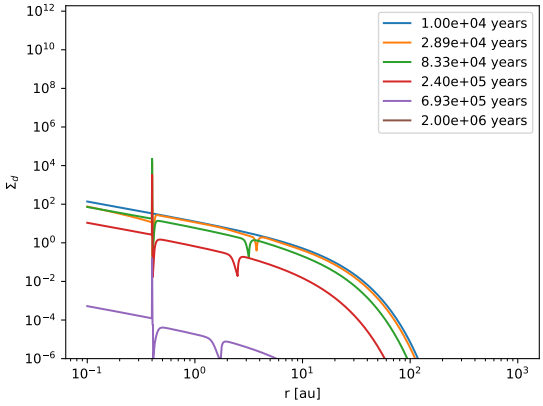
You can see that the deadzone rapidly shrinks as the disk evolves with time, and has vanished entirely by the end of the simulation. This is partly due to the high value of $\alpha_1 = 10^{-2}$.

If we put the $\alpha_1 = 10^{-3}$ then the disk evolves slower and the deadzone does not vanish, but qualitatively the disk behaves much the same.

The code snippet is `snippet_deadzone_4.py`. In Python run it as:

```
%run snippet_deadzone_4.py
```


1.00e+04 years
2.89e+04 years
8.33e+04 years
2.40e+05 years
6.93e+05 years
2.00e+06 years
Σ_g
r [au]


1.00e+04 years
2.89e+04 years
8.33e+04 years
2.40e+05 years
6.93e+05 years
2.00e+06 years
Σ_d
r [au]

Part II

Disk Vertical Structure: 1+1D and 2-D

Chapter 11

Disk vertical structure

The DISKLAB code package also includes a model of the disk vertical structure in a 1-D vertical manner. The `diskvertical` module is meant for this. As usual, this is a class, called `DiskVerticalModel`, and it is entirely *independent* of the `DiskRadialModel` class.

11.1 Basic equations

Let us make a simple 1-D vertical disk model at cylindrical radius r_c distance from the star. The model has z as the vertical coordinate, starting from $z = 0$ (midplane) to some value $z = z_{\max}$, which is typically a few pressure scale heights. We have to make a distinction between *cylindrical* radius r_c and *spherical* radius r_s:

$$r_s^2 = r_c^2 + z^2 \tag{11.1}$$

For $z \ll r_c$ this distinction is usually so small that one can ignore the difference. But if one is interested in the very upper layers of the disk, especially at large radii from the star, the difference can become large enough to be noticable.

11.1.1 Hydrostatic equilibrium

Strictly speaking, the hydrostatic equilibrium condition is a 3-D condition: The force components in all three directions (in the corotating frame) have to add up to zero. However, for axisymmetric disks without self-gravity, it fortunately turns out that the pressure equilibrium in vertical direction (along cylinders) becomes an independent 1-D ordinary differential equation in z.

The gravitational force per unit mass of gas is

$$\mathbf{f}_g = -\frac{GM_*}{r_s^3}\left(r_c\mathbf{e}_{r_c} + z\mathbf{e}_z\right) \tag{11.2}$$

where $\mathbf{e}_{r_c}$ is the unit vector in cylindrical radial direction and $\mathbf{e}_z$ the unit vector in vertical direction. The centrifugal force is

$$\mathbf{f}_c = \frac{v_\phi^2}{r_c}\mathbf{e}_{r_c} \tag{11.3}$$

and the pressure gradient force is

$$\mathbf{f}_p = -\frac{1}{\rho}\frac{\partial p}{\partial r_c}\mathbf{e}_{r_c} - \frac{1}{\rho}\frac{\partial p}{\partial z}\mathbf{e}_z \tag{11.4}$$

where $p = \rho c_s^2$ is the gas pressure, and c_s is the isothermal sound speed (cf. Eq. 2.4). Hydrostatic equilibrium means that all these forces add up to zero:

$$\mathbf{f}_g + \mathbf{f}_c + \mathbf{f}_p = 0 \tag{11.5}$$

For the $\mathbf{e}_z$ component this leads to the equation

$$\frac{\partial \ln p(r_c,z)}{\partial z} = -\frac{GM_*}{(r_c^2+z^2)^{3/2}} \frac{z}{c_s(r_c,z)^2} \tag{11.6}$$

If the temperature structure, and thus $c_s(r_c,z)$, is a *given* function of r_c and z (see Subsection 11.1.2), then Eq. (11.6) can be directly numerically integrated from the midplane upward, assuming we know the midplane pressure $p(r_c,0)$. This yields the vertical pressure profile $p(r_c,z)$. By dividing this by the known $c_s(r_c,z)^2$, we obtain the density profile $\rho(r_c,z)$ and the vertical density structure has been found.

As promised, the 3-D hydrostatic structure equations have reduced to a simple 1-D ordinary differential equation (Eq. 11.6) that can be integrated without knowledge of the density or pressure at other positions in r_c (or in azimuth). And this is not an approximation: it is exact. It works because the right-hand-side of Eq. (11.6) contains no dependence on ρ or p.

There is one small problem still, but that can be easily solved: Typically we do not know $p(r_c,0)$ a-priori, but instead we prescribe the radial surface density profile

$$\Sigma(r_c) = \int_{-\infty}^{+\infty} \rho(r_c,z) dz \tag{11.7}$$

So how do we find the vertical density profile that is consistent with this (prescribed) surface density profile? The way we do this is as follows: Numerically we start the vertical integration of Eq. (11.6) with a trial value of $p(r_c,0)$, integrate Eq. (11.6), and rescale $\rho(r_c,z)$ to match Eq. (11.7). This immediately gives us the 2-D density structure.

Note that all of this does not work if we include self-gravity of the disk.

Let us now see how this works for the simplest example possible: a vertically isothermal geometrically thin disk. Vertically isothermal means that c_s in Eq. (11.6) can be considered a given constant. And in the geometrically thin limit ($z \ll r_c$) we can approximately set $r_s \simeq r_c$, or in Eq. (11.6) set $r_c^2+z^2 \simeq r_c^2$. Under these simplifying conditions the solution to Eq. (11.6) becomes:

$$\rho(r_c,z) = \frac{\Sigma_g(r_c)}{\sqrt{2\pi} H_p(r_c)} \exp\left(-\frac{1}{2}\frac{z^2}{H_p(r_c)^2}\right) \tag{11.8}$$

with $H_p = c_s/\Omega_K$ and $\Omega_K = \sqrt{GM_*/r_c^3}$. This is the Gaussian vertical density structure that implicitly stands at the basis of the radial structure models of Part I of this tutorial (see e.g. Equation 4.3).

11.1.2 Vertical temperature structure

In Subsection 11.1 we assumed that we *know* what the temperature structure $T(r_c,z)$ of the disk is. However, in general this has to be calculated using a radiative transfer model. And here we encounter the issue of the 3-D geometry again, that we managed to circumvent in Subsection 11.1.1: The radiative transfer problem in protoplanetary disks is a 3-D problem (or, with axial symmetry, at least a 2-D problem in the coordinates r_c and z). The simplest way to see this is to imagine how the light from the star (which irradiates and heats the disk) can easily be shadowed by a vertically extended ("fat") inner disk: in such a case the stellar photons get absorbed by the disk regions close to the star (small r_c) and never reach the surface of the disk regions far from the star (large r_c). These outer disk regions will therefore be colder than if this shadowing did not occur.

Strictly speaking, therefore, there is no accurate way to compute a 1-D vertical structure model of the temperature of the disk without including the radial structure as well (and thus making it a 2-D model). This will be dealt with in Section 12.4 and Section 13.1.

However, as we have seen in Section 2.2, there are *approximate* methods to treat the radiative transfer as a local problem in r_c, and thus making it a 1-D problem in z. This is what we will do here.

We will start with a simple estimate of the vertical density structure $\rho(z)$. Let us assume that the stellar irradiation enters the disk under an angle φ called the *flaring angle*, which in the code is called `flang`. The optical depth of the disk to the stellar radiation at a given z is then

$$\tau_*(z) = \int_z^{\infty} \frac{\rho_{\rm d}(z')\kappa_{\rm P*}}{\varphi} dz' \tag{11.9}$$

where the dust density $\rho_{\rm d}$ is the dust density. For now, let us assume that it is a given fraction of the gas density:

$$\rho_{\rm d} = {\rm dtg}\,\rho_{\rm g} \tag{11.10}$$

where dtg is the dust-to-gas ratio, here assumed to be constant with z. The opacity $\kappa_{\rm P*}$ is the *Planck mean opacity* for stellar wavelengths, which for now we will assume to be a given value, but later we will show how this can be self-consistently computed. We now can compute the flux of stellar radiation:

$$F_*(z) = \frac{L_*}{4\pi r^2}\,e^{-\tau_*(z)} \tag{11.11}$$

We do not take into account the stellar spectrum for now, just the frequency-integrated stellar flux. The mean intensity of stellar radiation at each height z in the disk is then

$$J_*(z) = \frac{F_*(z)}{4\pi} \tag{11.12}$$

The flux is extincted by the dust in the disk. We assume, for now, that the albedo of the dust is zero, so that all the radiation of the star is absorbed and re-emitted. The source term of this re-emission in units of erg cm^{-3} s^{-1} is then

$$q_*(z) = \varphi\,\frac{dF_*(z)}{dz} \tag{11.13}$$

One could also write $q_*(z)$ in terms of $J_*(z)$ and the extinction coefficient, but we chose the flux-conservative form for numerical accuracy even at low numerical resolution.

Next we handle the diffuse radiation field given by the mean intensity $J_{\rm diff}(z)$ and first moment $H_{\rm diff}(z)$. They obey the equations:

$$\frac{dH_{\rm diff}(z)}{dz} = \frac{q(z)}{4\pi} \tag{11.14}$$

$$\frac{dJ_{\rm diff}(z)}{dz} = -3\rho_{\rm d}(z)\kappa_{\rm R}H_{\rm diff}(z) \tag{11.15}$$

where $\kappa_{\rm R}$ is the *Rosseland mean opacity* of the dust, and $q(z)$ is the source term, which for an irradiation-only model is $q(z) = q_*(z)$. Any viscous heating or other source terms can be included in $q(z)$ as well by simple addition. The boundary conditions are $H_{\rm diff}(0) = 0$ and $H_{\rm diff}(z_{\rm max}) = J_{\rm diff}(z_{\rm max})/\sqrt{3}$. Given $q_*(z)$ we first integrate $H_{\rm diff}(z)$ from $z = 0$ to $z = z_{\rm max}$, starting with $H_{\rm diff}(0) = 0$. Then we impose the upper boundary condition, giving us $J_{\rm diff}(z_{\rm max})$, and we integrate $J_{\rm diff}(z)$ back to the midplane.

Once we have the radiation field, we can compute the gas and dust temperature (which we assume to be the same) according to:

$$T(z) = \left(\frac{\pi}{\sigma_{\rm SB}}\left(J_{\rm diff}(z) + \frac{\kappa_{\rm P*}}{\kappa_{\rm P}(T)}J_*(z)\right)\right)^{1/4} \tag{11.16}$$

where, as usual, $\sigma_{\rm SB}$ is the Stefan-Boltzmann constant and $\kappa_{\rm P}(T)$ is the Planck-mean opacity of the dust for the temperature T.

Once we have this vertical temperature profile, we can solve for the vertical density profile using the equation of hydrostatic equilibrium, as described in Section 11.1.1. This will give a new density profile, and thus the radiative transfer would have to be recomputed. In practice this requires a few iterations until convergence is reached. We then have the full vertical density and temperature solution.

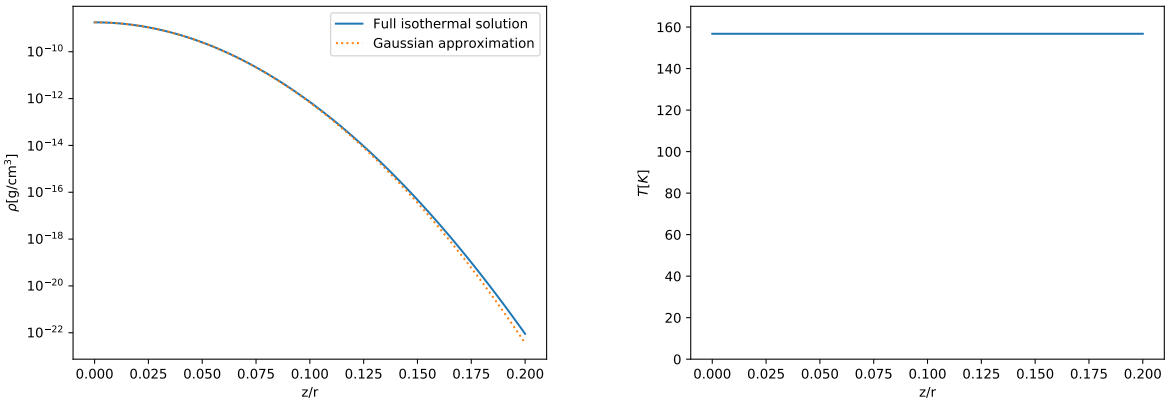
11.2 Basic setup

Let us set up a simple vertically isothermal disk model at $r_c = 1$ au to begin with. We fix the value of φ and estimate the temperature to be $T = (\frac{1}{2}\varphi L_*/4\pi r^2\sigma_{\rm SB})^{0.25}$. The code snippet is `snippet_vertstruct_basic_1.py`. In Python run it as:

```
%run snippet_vertstruct_basic_1.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, np, plt, MS, LS, au, kk, mp, finalize

mstar  = 1 * MS
lstar  = 1 * LS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
zrmax  = 0.2
opac   = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert   = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=zrmax,
                           lstar=lstar, meanopacitymodel=opac)
vert.kappadust = kapdst

# For comparison: The simple Gaussian model:
cs       = np.sqrt(kk*vert.tgas[0]/(vert.mugas*mp))
hp       = cs/vert.omk_midpl
rhogauss = (siggas/(np.sqrt(2*np.pi)*hp))*np.exp(-0.5*(vert.z/hp)**2)

plt.figure()
plt.plot(vert.z / vert.r, vert.rhogas,label='Full isothermal solution')
plt.plot(vert.z / vert.r, rhogauss,':',label='Gaussian approximation')
plt.xlabel('z/r')
plt.yscale('log')
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')
plt.legend()

plt.figure()
plt.plot(vert.z / vert.r, vert.tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(0, 170.)

finalize(results=(vert.rhogas,vert.tgas))
```



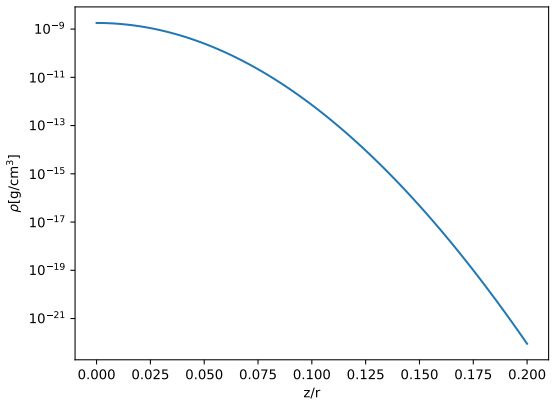
As you can see, the temperature is constant, as we assumed. The density profile is accordingly almost a perfect Gaussian (cf. Eq. 11.8). The slight deviation from Gaussian is due to the difference between the cylindrical radius r_c and spherical radius r_c, as discussed in Section 11.1.1. This difference becomes larger for larger z/r. Since the ratio H_p/r_c (the dimensionless pressure scale height) increases with r_c, this difference becomes more relevant at larger r_c, i.e. in the outer regions of the disk ($\sim 100\,\mathrm{au}$).

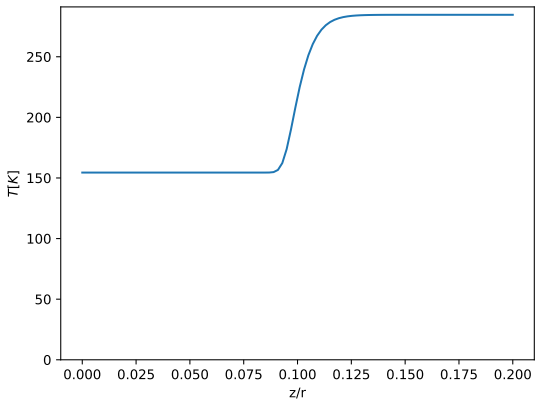
Next let us compute the radiative transfer for this Gaussian vertical density profile. The code snippet is `snippet_vertstruct_basic_2`. In Python run it as:

```
%run snippet_vertstruct_basic_2.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, plt, MS, LS, au, finalize

mstar  = 1 * MS
lstar  = 1 * LS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
zrmax  = 0.2
opac = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=zrmax,
                      lstar=lstar, meanopacitymodel=opac)
vert.kappadust = kapdst
vert.irradiate_with_flaring_index()
vert.solve_vert_rad_diffusion()
vert.compute_temperature_from_radiation()
plt.figure()
plt.plot(vert.z / vert.r, vert.rhogas)
plt.xlabel('z/r')
plt.yscale('log')
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')

plt.figure()
plt.plot(vert.z / vert.r, vert.tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(bottom=0)

finalize(results=(vert.rhogas,vert.tgas))
```





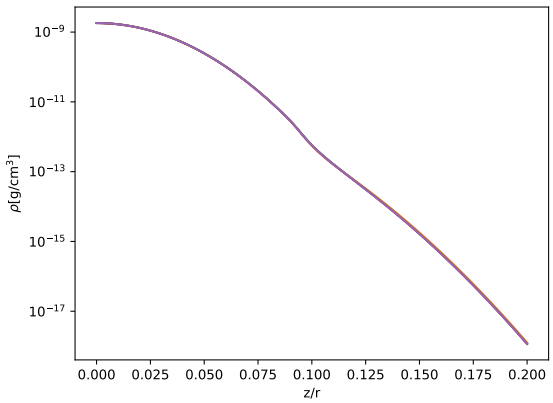
The density profile is still a Gaussian, because we have not yet iterated on the vertical hydrostatic equilibrium. But the temperature profile now clearly shows a transition from midplane tempearture to (warmer) surface layer temperature, as expected.

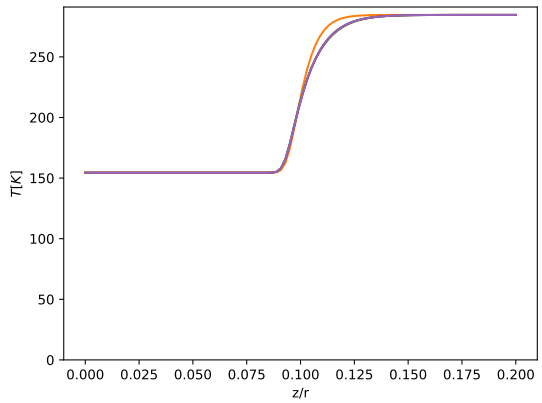
Now let us feed this temperature profile back into the hydrostatic equilibrium equation and do a few iterations. The code snippet is `snippet_vertstruct_basic_3.py`. In Python run it as:

```
%run snippet_vertstruct_basic_3.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, plt, MS, LS, au, finalize
import copy

mstar  = 1 * MS
lstar  = 1 * LS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
zrmax  = 0.2
opac = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=zrmax,
                         lstar=lstar, meanopacitymodel=opac)
verts = [vert]
niter = 5
for iter in range(niter):
    vert.irradiate_with_flaring_index()
    vert.solve_vert_rad_diffusion()
    vert.compute_temperature_from_radiation()
    vert.compute_rhogas_hydrostatic()
    vert.compute_mean_opacity()
    verts.append(copy.deepcopy(vert))

# plotting

plt.figure()
plt.plot(vert.z / vert.r, vert.rhogas)
for iter in range(1, niter):
    plt.plot(vert.z / vert.r, verts[iter].rhogas)
plt.xlabel('z/r')
plt.yscale('log')
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')

plt.figure()
plt.plot(vert.z / vert.r, vert.tgas)
for iter in range(1, niter):
    plt.plot(vert.z / vert.r, verts[iter].tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(bottom=0)

finalize(results=(verts[-1].rhogas,verts[-1].tgas))
```



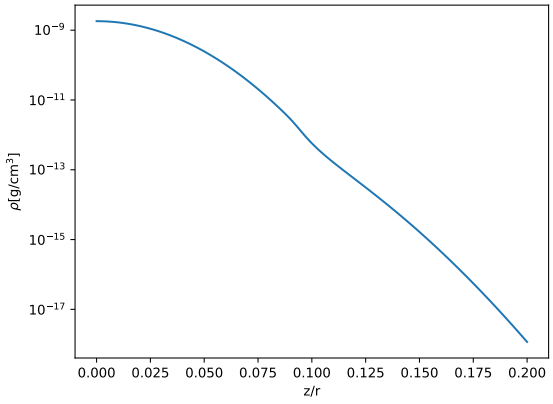


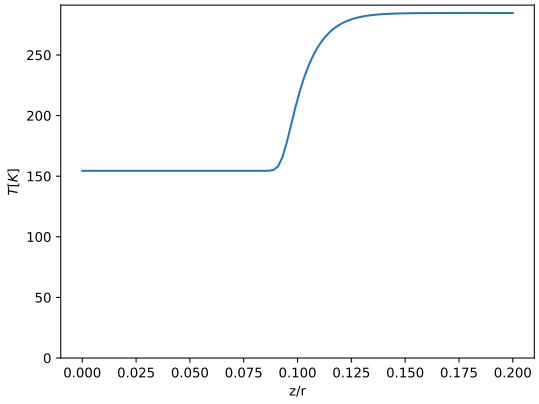
In practice you do not have to do this iteration by hand. There is a method for that. The code snippet is `snippet_vertstruct_basic_4`
In Python run it as:

```
%run snippet_vertstruct_basic_4.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, plt, MS, LS, au, finalize

mstar  = 1 * MS
lstar  = 1 * LS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
zrmax  = 0.2
opac   = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert   = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=zrmax,
                           lstar=lstar, meanopacitymodel=opac)
vert.iterate_vertical_structure()

plt.figure()
plt.plot(vert.z / vert.r, vert.rhogas)
plt.xlabel('z/r')
plt.yscale('log')
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')

plt.figure()
plt.plot(vert.z / vert.r, vert.tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(bottom=0)

finalize(results=(vert.rhogas,vert.tgas))
```





11.3 Including viscous heating

We can include viscous heating using the following formula:

$$q_{\rm visc}(z) = \frac{9}{4}\Omega_K^2 \rho_{\rm gas}(z)\nu(z) \tag{11.17}$$

with viscosity ν given by

$$\nu(z) = \alpha(z)\frac{c_s(z)^2}{\Omega_K} \tag{11.18}$$

This source is simply added to the irradiative source, so that the total source is

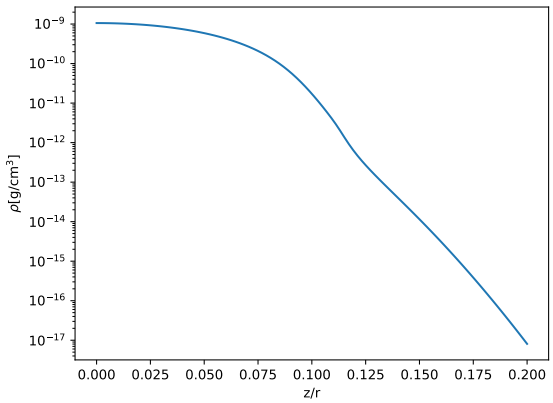
$$q(z) = q_{\rm visc}(z) + q_*(z) \tag{11.19}$$

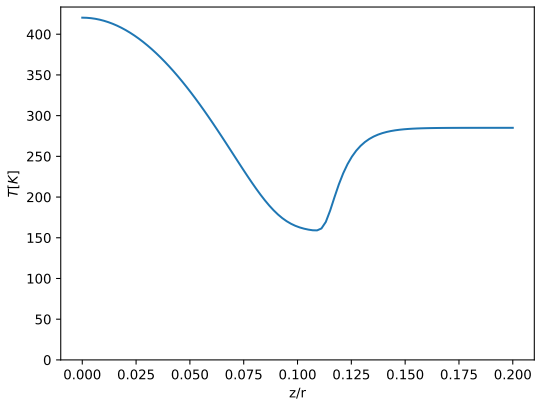
Let us simply add this to our example: we set $\alpha = 10^{-3}$ and see what comes out. The code snippet is `snippet_vertstruct_vischeat_1.py`. In Python run it as:

```
%run snippet_vertstruct_vischeat_1.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, plt, MS, LS, au, finalize
mstar  = 1 * MS
lstar  = 1 * LS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
zrmax  = 0.2
alpha  = 1e-3
opac = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=zrmax,
                     lstar=lstar, meanopacitymodel=opac,
                     alphavisc=alpha)
vert.iterate_vertical_structure()

plt.figure()
plt.plot(vert.z / vert.r, vert.rhogas)
plt.xlabel('z/r')
plt.yscale('log')
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')

plt.figure()
plt.plot(vert.z / vert.r, vert.tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(bottom=0)

finalize(results=(vert.rhogas,vert.tgas))
```



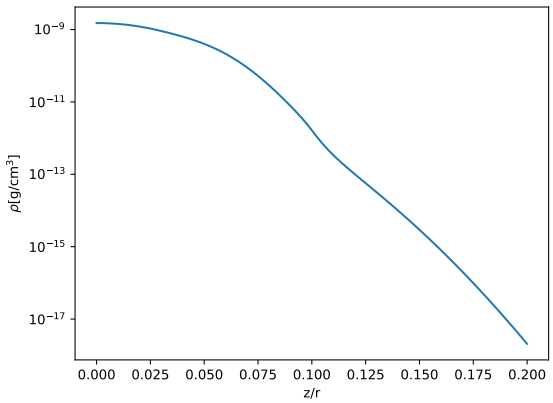


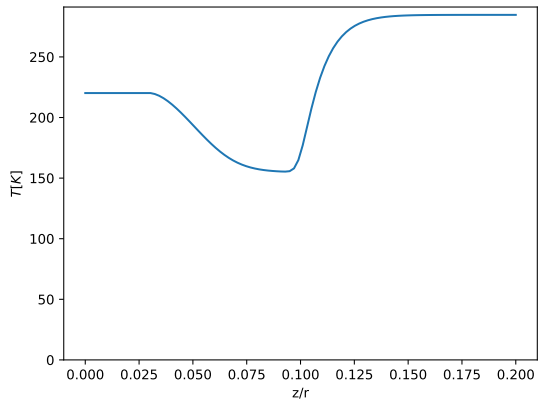
DISKLAB allows α to be $z-$dependent. So if the disk has a dead zone, this will be included. Let us include such a "dead zone" in a simplified way: we simply set $\alpha = 0$ for $z < 0.03\,r$. The code snippet is `snippet_vertstruct_vischeat_2.py`. In Python run it as:

```
%run snippet_vertstruct_vischeat_2.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, np, plt, MS, au, LS, finalize
mstar  = 1 * MS
lstar  = 1 * LS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
zrmax  = 0.2
nz     = 100
alpha  = np.zeros(nz) + 1e-3
opac = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=zrmax,
                     lstar=lstar, meanopacitymodel=opac,
                     alphavisc=alpha, nz=nz)
vert.alphavisc[vert.z < 0.03 * vert.r] = 0.0
vert.iterate_vertical_structure()

plt.figure()
plt.plot(vert.z / vert.r, vert.rhogas)
plt.xlabel('z/r')
plt.yscale('log')
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')

plt.figure()
plt.plot(vert.z / vert.r, vert.tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(bottom=0)

finalize(results=(vert.rhogas,vert.tgas))
```





11.4 Including 1-D vertical self-gravity

Self-gravity is a global force, and it is not really possible to treat it in a localized 1-D vertical manner. But one can approximate the effect of the vertical squeezing of the disk by its own gravity by employing the 1-D plane-parallel approximation of the Poisson equation. If we have an infinite plane surface with mass surface density Σ, then the vertical downward gravitational body force of a particle above that plane is $f_z = -2\pi G\Sigma$. We can now regard our 1-D vertical structure model as a set of 1-D infinite planes. The body force at any given height z is then

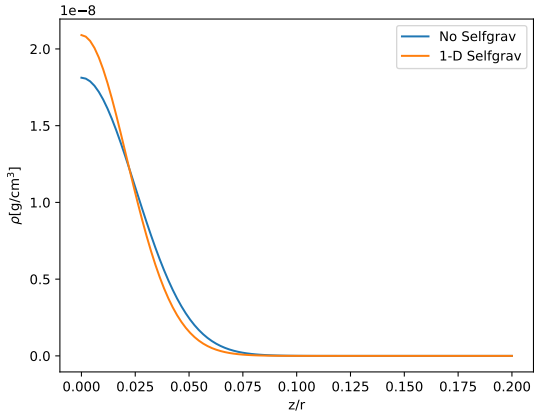
$$f_z(z) = -4\pi G \int_0^z \rho(z) dz \tag{11.20}$$

The factor 2 comes in because every step downward in z-direction means that a plane of gas no longer attracts downward, but in fact pulls upward. At the midplane the up- and down-ward forces cancel exactly, as they should. And for $z\infty$ the force becomes $-2\pi G\Sigma$ as it should.

Now we add this force to the vertical component of the gravitational force by the star, and we can thus compute the self-gravity-corrected vertical structure. This is all automatically handled by the `diskvertical.iterate_vertical_structure()` method, with the keyword `iterate_selfgrav1d=True`.

The code snippet is `snippet_vertstruct_selfgrav1d_1.py`. In Python run it as:

```
%run snippet_vertstruct_selfgrav1d_1.py
```



One sees that the self-gravity squeezes the vertical structure a bit, but not much. Already the example above is not far from gravitational instability (it has $Q \simeq 4$). If the surface density is chosen such that $Q \lesssim 2$, then the disk would become gravitationally unstable, and the solution is not physical.

Warning: Whenever you include self-gravity, you must always check that the Toomre $Q > 2$, otherwise you obtain a solution that, in reality, does not exist because it is gravitationally unstable!

11.5 Using more realistic opacities

IN PROGRESS

11.6 Time-dependent radiative transfer

Although the radiation field itself propagates with the speed of light, and is thus effectively instant, the reaction of the gas temperature to the radiative cooling takes a certain amount of time. To implement this we follow Kuiper, Klahr, Dullemond, Kley & Henning (2010), A&A 511, 81. In this method the local gas and dust temperature is assumed to be always in strict equilibrium with the radiation field according to Eq. (11.16). The time-dependent zeroth moment equation of the radiation field now includes the thermal energy of the gas and the dust. We assume

that the thermal heat capacity of the dust is negligible compared to that of the gas. The energy conservation equation then reads

$$\frac{\partial(E_{\rm gas}(z)+E_{\rm diff}(z))}{\partial t} = -\frac{\partial F_{\rm diff}(z)}{\partial z} + q(z) \tag{11.21}$$

where $E_{\rm diff} = 4\pi J_{\rm diff}/c$, $F_{\rm diff} = 4\pi H_{\rm diff}$ and $E_{\rm gas} = c_{\rm v}\rho_{\rm gas}T$, where $c_{\rm v}$ is the specific heat of the gas, given by

$$c_{\rm v} = \frac{k_B}{(\gamma-1)\mu m_p} \tag{11.22}$$

Since the gas temperature is assumed to be strictly coupled to the radiation field, we can express $\partial_t E_{\rm gas}$ in terms of $\partial_t E_{\rm rad}$ through

$$\frac{\partial E_{\rm gas}}{\partial t} = c_{\rm v}\rho_{\rm g}\frac{\partial T}{\partial t} = \frac{c_{\rm v}\rho_{\rm g}}{4aT^3}\frac{\partial E_{\rm diff}}{\partial t} \tag{11.23}$$

where we used the following version of Eq. (11.16):

$$aT^4 = E_{\rm diff} + \frac{\kappa_{\rm P*}}{\kappa_{\rm P}(T)}E_* \tag{11.24}$$

with $a = 4\sigma_{\rm SB}/c$. *For now we ignore the time-derivative of* $\kappa_{\rm P}(T)$, so that Eq. (11.23) follows directly. Now eleminating $E_{\rm gas}$ in Eq. (11.21) in favor of $E_{\rm diff}$, we obtain

$$\left(1+\frac{c_{\rm v}\rho_{\rm g}}{4aT^3}\right)\frac{\partial E_{\rm diff}(z)}{\partial t} = -\frac{\partial F_{\rm diff}(z)}{\partial z} + q(z) \tag{11.25}$$

This is the 1-D form of the equation 6 from Kuiper et al., who define $f_c = 1/(1+c_{\rm v}\rho_{\rm g}/4aT^3)$ to simplify their equation.

In DISKLAB we use $J_{\rm diff} = cE_{\rm diff}/4\pi$ and $H_{\rm diff} = F_{\rm diff}/4\pi$. So the equation then becomes

$$\frac{1}{c}\left(1+\frac{c_{\rm v}\rho_{\rm g}}{4aT^3}\right)\frac{\partial J_{\rm diff}(z)}{\partial t} = -\frac{\partial H_{\rm diff}(z)}{\partial z} + \frac{q(z)}{4\pi} \tag{11.26}$$

We now integrate Eq. (11.26) using an implicit integration scheme, using the diffusion solver of appendix A. Since that solver does not have an explicit factor before the time-derivative, we simply provide the solver with a space-dependent time step $\Delta t(z) = dt/c(1+c_{\rm v}\rho_{\rm g}(z)/4aT(z)^3)$.

Now let us experiment with this. We start with an irradiated disk with $\varphi = 0.05$ in full equilibrium. Now we change the flaring angle to $\varphi = 0.07$, and let the time-dependent radiative diffusion change the temperature. We take a single time step of 100 years.

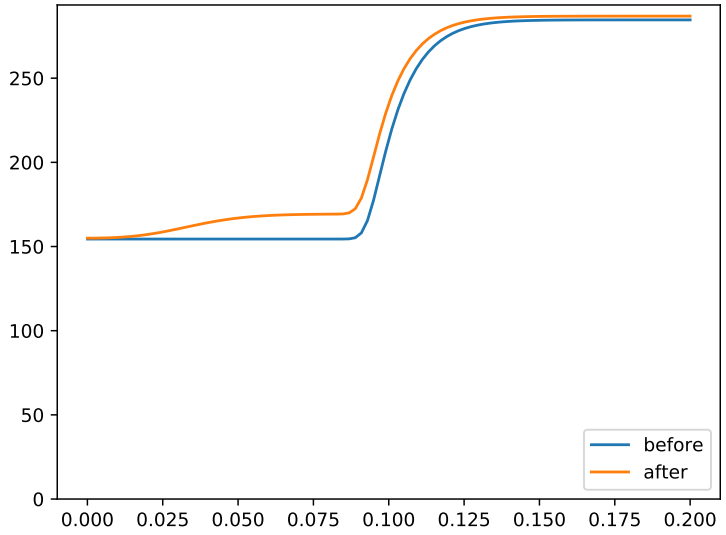
The code snippet is `snippet_vertstruct_timedep_1.py`. In Python run it as:

```
%run snippet_vertstruct_timedep_1.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, plt, MS, year, au, finalize

mstar  = 1 * MS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
flang1 = 0.07
dt     = 1e2 * year   # Time step
opac = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=0.2,
                    meanopacitymodel=opac)
vert.iterate_vertical_structure()

plt.figure()
```

```
plt.plot(vert.z / vert.r, vert.tgas, label='before')
vert.flang = flang1
vert.irradiate_with_flaring_index()
vert.timestep_vert_rad_diffusion(dt)
plt.plot(vert.z / vert.r, vert.tgas, label='after')
plt.ylim(bottom=0)
plt.legend(loc='lower right')

finalize(results=(vert.rhogas,vert.tgas))
```



As you can see the radiative heating takes time to penetrate into the disk. This is the time-dependent heating from the top. To convince yourself that everything is self-consistent, you can (as a test) change `flang1` to the same value as `flang` (i.e. 0.05). In that case nothing should happen, and you can convince yourself that that is indeed the case.

Note that we took *a single time step* here, for a non-linear problem. That is of course not quite correct. We should do multiple smaller time steps, because the factor f_c changes in time.

Note also that we *did not change the density structure* during this time step. So the disk has not reacted to the change in temperature. This is, of course, not self-consistent.

We should let the disk accomodate hydrostatically to the new temperature. But in doing so, we have to do this *adiabatically*. In other words, we have to fix the *specific entropy* everywhere, and then find a new hydrostatic equilibrium. The adiabatic equation of state is

$$p_{\rm gas}(z) \equiv \rho_{\rm gas}(z)\frac{k_B T(z)}{\mu m_p} = K(z)\rho_{\rm gas}(z)^\gamma \tag{11.27}$$

The specific entropy is

$$s = s_0 + \frac{k_B}{\mu(\gamma-1)}\ln K \tag{11.28}$$

where we take $s_0 = 0$. For each gas packet the s should (upon re-computation of the vertical hydrostatic structure $\rho_{\rm gas}(z)$) remain the same *in a comoving Lagrangian manner*. So we first map $s(z)$ onto $s(\sigma)$, where $\sigma(z)$ is the column density:

$$\sigma(z) = \int_z^\infty \rho(z)dz \tag{11.29}$$

with the property $\sigma(0) = \Sigma_{\rm gas}/2$. We then compute the new $\rho_{\rm gas}(z)$ for the given temperature structure $T(z)$ and recompute the new $\sigma_{\rm new}(z)$. We now map $s(\sigma)$ onto this new $\sigma_{\rm new}(z)$ through interpolation (linear interpolation is usually enough). This gives a new function $s(z)$ from which we can compute a new temperature $T(z)$. It is important

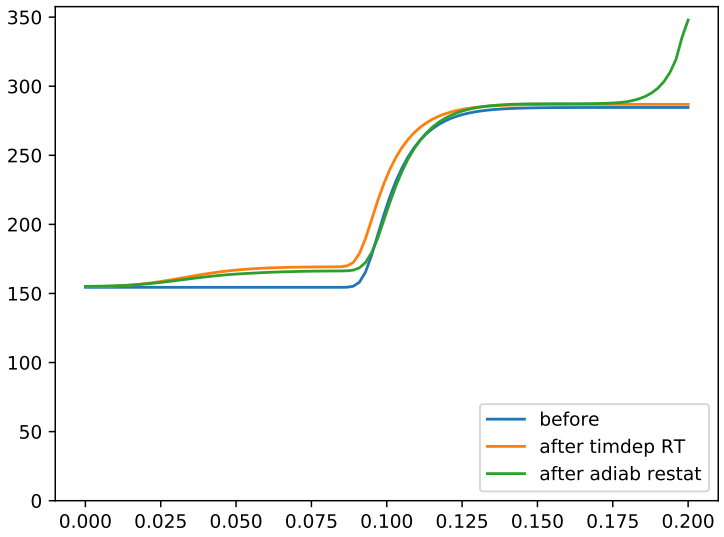
to define the $\sigma(z)$ as the column density toward $z = +\infty$ instead of between $z = 0$ and z, because in the very surface layers the density is so low that one would hit the machine precision limit if $\sigma(z)$ is defined from the bottom. The interpolation would then go wrong.

Let us try this method out. The code snippet is `snippet_vertstruct_timedep_2.py`. In Python run it as:

```
%run snippet_vertstruct_timedep_2.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, plt, MS, year, au, finalize

mstar  = 1 * MS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
flang1 = 0.07
dt     = 1e2 * year   # Time step
opac = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=0.2,
                    meanopacitymodel=opac)
vert.iterate_vertical_structure()

plt.figure()
plt.plot(vert.z / vert.r, vert.tgas, label='before')
vert.flang = flang1
vert.irradiate_with_flaring_index()
vert.timestep_vert_rad_diffusion(dt)
plt.plot(vert.z / vert.r, vert.tgas, label='after timdep RT')
vert.compute_rhogas_hydrostatic_adiabatic()
plt.plot(vert.z / vert.r, vert.tgas, label='after adiab restat')
plt.ylim(bottom=0)
plt.legend(loc='lower right')

finalize(results=(vert.rhogas,vert.tgas))
```



One sees that due to the higher temperature, the surface of the disk has expanded a bit, hence the red curve is moving upward again near the surface layers.

What is peculiar is the sudden upward trend of the temperature at the upper edge. This is because due to the expansion of the disk, while keeping the $z_{\max}$ fixed, the *very* upper layers just below the $z_{\max}$ have, so to speak, left the upper grid boundary. Although we have renormlized the total density (and therefore no mass is really lost), the effect is that the mapping of the entropy onto the new σ-grid is thus slightly wrong near the upper boundary. This is a computational / numerical effect caused by imposing a fixed upper grid boundary at $z = z_{\max}$. When we do another time step with the time-dependent radiative diffusion, this disappears, but it will always re-appear after calling the hydrostatic adiabatic solver. If the time steps are small enough it should be not a major problem, however. It occurs at such a high location in the surface layers, that the denisty there is negligbly low.

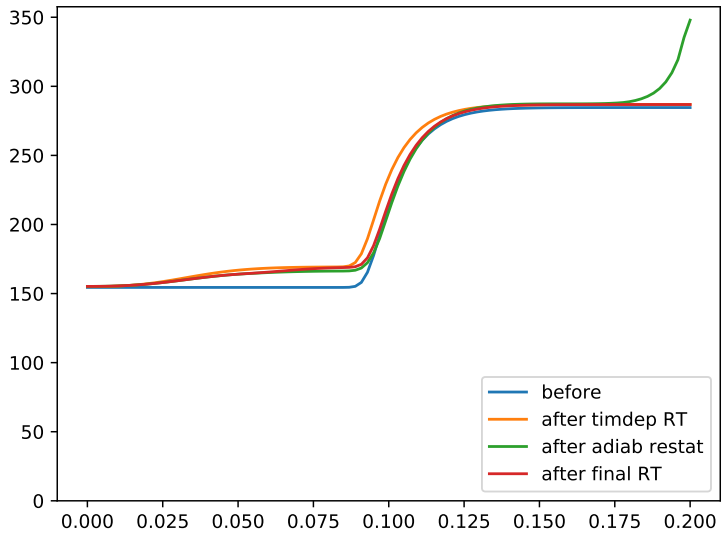
The solution is simple: just do a tiny time step for radiative transfer and things are back in equilibrium.

The code snippet is `snippet_vertstruct_timedep_3.py`. In Python run it as:

```
%run snippet_vertstruct_timedep_3.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, plt, MS, year, au, finalize

mstar  = 1 * MS
r      = 1 * au
siggas = 1700.
dtg    = 0.01
kapdst = 1e2
flang  = 0.05
flang1 = 0.07
eps    = 1e-3
dt     = 1e2 * year   # Time step
opac = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
vert = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=0.2, meanopacitymodel=opac)
vert.iterate_vertical_structure()

plt.figure()
plt.plot(vert.z / vert.r, vert.tgas, label='before')
vert.flang = flang1
vert.irradiate_with_flaring_index()
vert.timestep_vert_rad_diffusion((1 - eps) * dt)
plt.plot(vert.z / vert.r, vert.tgas, label='after timdep RT')
vert.compute_rhogas_hydrostatic_adiabatic()
plt.plot(vert.z / vert.r, vert.tgas, label='after adiab restat')
vert.compute_mean_opacity()
vert.irradiate_with_flaring_index()
vert.timestep_vert_rad_diffusion(eps * dt)
plt.plot(vert.z / vert.r, vert.tgas, label='after final RT')
plt.ylim(bottom=0)
plt.legend(loc='lower right')

finalize(results=(vert.rhogas,vert.tgas))
```



11.7 Dust settling and vertical mixing

The `dustvertstruct.py` module also allows for modeling the vertical settling and vertical mixing of dust species (or of gas chemical species, which are mathematically equivalent to dust with zero stopping time). Each dust or gas species is represented by an object of type `diskvertcomponent`, in much the same way as the radial `DiskRadialModel` object can be supplemented with `DiskRadialComponent` objects (see Section 6.7). A `diskvertcomponent` object should be linked to an existing `DiskVerticalModel` object, again in the same way as a `DiskRadialComponent` is linked to a `DiskRadialModel` object (see Chapter 6).

The modeling of the vertical settling and mixing goes in a very similar way as for the radial drift and mixing of Chapter 6.

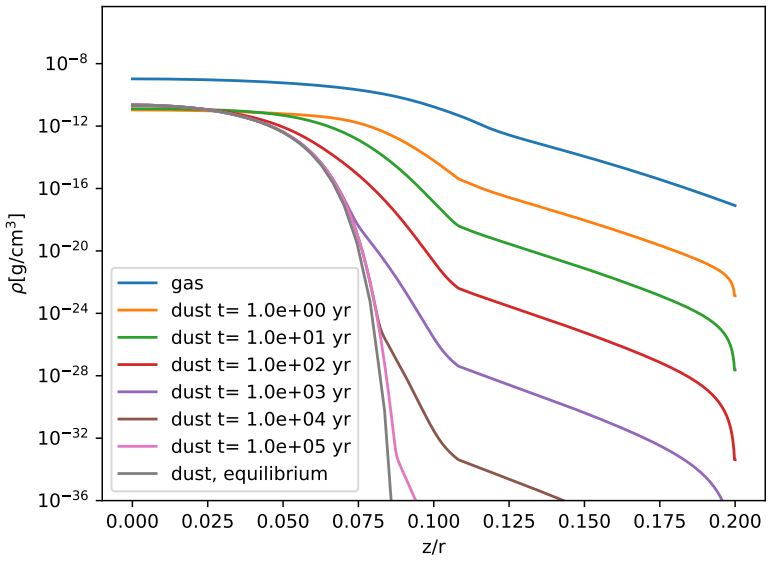
Here is an example. The code snippet is `snippet_vertstruct_dustsett_1.py`. In Python run it as:

```
%run snippet_vertstruct_dustsett_1.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, DiskVerticalComponent, np, plt, MS, LS, \
    au, year, finalize

mstar   = 1 * MS
lstar   = 1 * LS
r       = 1 * au
siggas  = 1700.
dtg     = 0.01
kapdst  = 1e2
flang   = 0.05
zrmax   = 0.2
opac    = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
dtg     = 0.01
agrain  = 1e0
xigrain = 3.0
alpha   = 1e-3
nz      = 1000
time    = np.array([0., 1e0, 1e1, 1e2, 1e3, 1e4, 1e5]) * year
ntime   = time.size
```

```
vert    = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=zrmax,
                        lstar=lstar, meanopacitymodel=opac, alphavisc=alpha, nz=nz)
vert.iterate_vertical_structure()
dstvert = DiskVerticalComponent(vert, dtg, agrain=agrain, xigrain=xigrain)

plt.figure()
plt.plot(vert.z / vert.r, vert.rhogas, label='gas')
for it in range(ntime - 1):
    dtime = time[it + 1] - time[it]
    dstvert.timestep_settling_mixing(dtime)
    plt.plot(vert.z / vert.r, dstvert.rho,
             label='dust t={0:8.1e} yr'.format(time[it + 1] / year))
    print('dust surface density = {0:3.5e}'.format(dstvert.diskverticalmodel.vertically_integrate(dst

rhodustfinal = dstvert.rho.copy()

dstvert.compute_settling_mixing_equilibrium()
plt.plot(vert.z / vert.r, dstvert.rho, label='dust, equilibrium')

plt.xlabel('z/r')
plt.yscale('log')
plt.ylim(bottom=1e-36)
plt.legend(loc='lower left')
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')

finalize(results=(rhodustfinal,dstvert.rho))
```



Note that this snippet demonstrates both the time-dependent modeling of the dust settling (using the function `timestep_settling_mixing()`), as well as the computation of the steady-state solution (using the function `compute_settling_mixing_equilibrium()`).

11.8 Effect of dust weight on vertical structure

In most cases we can compute the vertical pressure balance solution without having to worry about the weight of the dust. The reason is that it is generally the case that the dust-to-gas ratio is much smaller than unity. However, there could be instances where radial trapping of dust leads to such a strong accumultation of dust at a given radius,

that the vertically integrated dust-to-gas ratio becomes close to unity. In that case, the weight of the dust will have a non-negligible effect on the vertical pressure equilibrium of the disk. Essentially the gas pressure will not only have to support the weight of the gas, but also that of the dust. For the same disk temperature, the disk will then become a bit flatter.

The vertical pressure balance equation given by Eq. (11.6) then becomes

$$\frac{\partial \ln p(r_c,z)}{\partial z} = -\left(1+\frac{\rho_{\rm d}(r_c,z)}{\rho_{\rm g}(r_c,z)}\right)\frac{GM_*}{(r_c^2+z^2)^{3/2}}\frac{z}{c_s(r_c,z)^2} \tag{11.30}$$

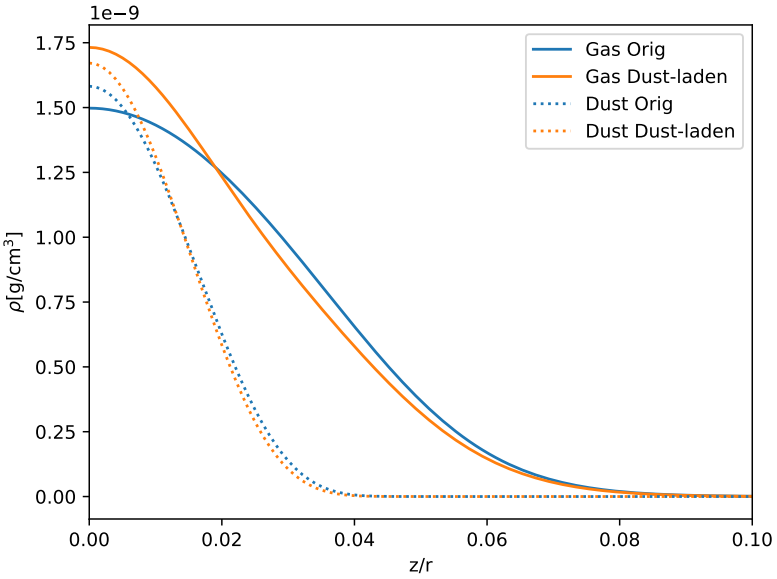
where $\rho_{\rm d}$ is the volume density of all dust species together, while $\rho_{\rm g}$ is the volume density of the gas. Note that once $\rho_{\rm d} > \rho_{\rm g}$ we should expect the streaming instability to operate (Youdin & Goodman, 2005, ApJ 620, 459). And when $\rho_{\rm d} \ll \rho_{\rm g}$ the effect of dust-loading is negligible. So the applicability of Eq. (11.30) is limited in scope. But here is a snippet to model it:

```
%run snippet_vertstruct_dustload_1.py
```

Here is the listing:

```
from snippet_header import DiskVerticalModel, np, plt, MS, LS, au, finalize
import copy

mstar  = 1 * MS
lstar  = 1 * LS
r      = 1 * au
siggas = 1700.
kapdst = 1e2
flang  = 0.05
zrmax  = 0.2
alpha  = 1e-4
agrain = 1e-1
dtg    = 0.5    # Yes, this is a huge dust-to-gas ratio!
opac = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]  # Only for opacity!
vert = DiskVerticalModel(mstar, r, siggas, flang=flang, zrmax=zrmax,
                         lstar=lstar, meanopacitymodel=opac,
                         alphavisc=alpha,nz=200)
vert.iterate_vertical_structure()
vert.add_dust(agrain,dtg=dtg)
vert.dust[0].compute_settling_mixing_equilibrium()
vertorig = copy.deepcopy(vert)

# Now include the dust loading
for iter in range(3):
    vert.iterate_vertical_structure(dtgitermax=20,dtgerrtol=0.02)
    vert.dust[0].compute_settling_mixing_equilibrium()

# Check dust mass conservation
vertorig.dust[0].compute_surfacedensity()
vert.dust[0].compute_surfacedensity()
assert np.abs(vert.dust[0].sigma/vertorig.dust[0].sigma-1.)<1e-3

# plotting
plt.figure()
plt.plot(vert.z / vert.r, vertorig.rhogas,label='Gas Orig')
plt.plot(vert.z / vert.r, vert.rhogas,label='Gas Dust-laden')
plt.gca().set_prop_cycle(None)
plt.plot(vert.z / vert.r, vertorig.dust[0].rho,':',label='Dust Orig')
plt.plot(vert.z / vert.r, vert.dust[0].rho,':',label='Dust Dust-laden')
plt.xlabel('z/r')
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')
plt.xlim(left=0,right=0.1)
plt.legend()
```

```
finalize()
```



One can see that the dust-laden gas is a bit more (though not much) concentrated toward the midplane.

Chapter 12

Disk 2D structure

Now that we dealt with the 1-D radial disk structure and the 1-D vertical disk structure, it is time to combine them into a 2-D disk structure. This is done with the `Disk2D` class. The simplest version of a 2-D model is simply a sequence (i.e. a python list) of 1-D vertical structure models, one for each radial grid point of a 1-D radial disk model. This is called a 1+1D model. The vertical structure models are then independent of each other. They "do not feel" their neigbors. Such a model is therefore not really a true 2-D model, but it often serves its purpose.

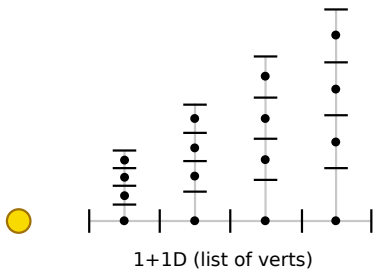
The `Disk2D` class also contains methods for truly 2-D models, where the vertical slices do communicate, e.g. via radiation or via self-gravity. These truly 2-D models require truly 2-D arrays, instead of a list of 1-D vertical models. The `Disk2D` class contains methods for mapping the 1+1D "list of vertical structures" to truy 2-D arrays, and backward.

Another issue is that some methods of the `Disk2D` class require the disk model to be in *spherical* coordinates. The 1-D vertical structure models are truly vertical, i.e. they are along cylindrical coordinate lines. The `Disk2D` class contains coordinate mappings from cylindrical to spherical coordinates, and back.

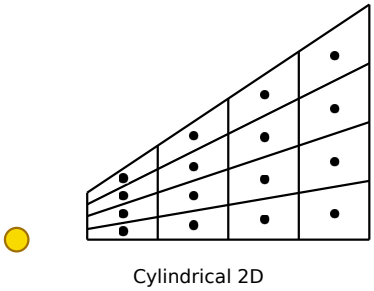
Here is a summary of the three different representations of 2-D models:

- *1+1D List of vertical structures:*

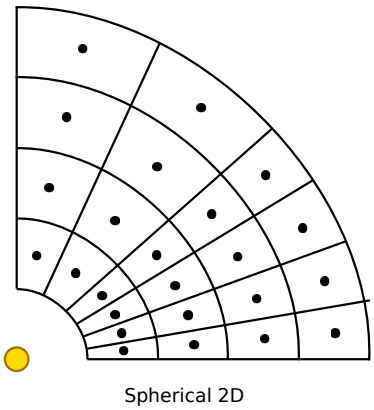
  This is the simplest extension of the vertical structure models of Chapter 11. It is simply a python list of 1-D vertical structure models of the type discussed in Chapter 11. One starts with a 1-D *radial* model, and for each radial grid point one constructs a 1-D vertical model and adds it to the list called `Disk2D.verts`. So `Disk2D.verts[3]` is the 1-D vertical structure model at radius `Disk2D.r[3]`.



- *2-D Cylindrical coordinates:*
  This is the 2-D version of the 1+1D List of vertical structures. You can use `Disk2D.convert_1p1d_to_cyl2d()` to map the 1+1D List of vertical structures onto the 2-D cylindrical coordinates. The z-grid points are r-dependent, such that for radius `Disk2D.r[ir]` (with `ir` being the radial index, e.g. `ir=3` for the third radial grid point), the vertical grid `Disk2D.cyl2d_zz[ir,:]` corresponds to the z-grid of vertical structure model `Disk2D.verts[ir]`. The physical quantities in the 2-D cylindrical model are thus a *direct copy* of the 1+1D model: no interpolation is needed. The only difference with the 1+1D model is that now all the data are available in a single 2-D array. For instance, the gas density is then `Disk2D.cyl2d_rhogas[:,:]` instead of `Disk2D.verts[:].rhogas[:]`. Always keep in mind, though, that if you manipulate e.g. `Disk2D.cyl2d_rhogas[:,:]`, then `Disk2D.verts[:].rhogas[:]` stays unaffected, until you call `Disk2D.convert_cyl2d_to_1p1d()` to copy the data back. Also keep in mind that `Disk2D.convert_cyl2d_to_1p1d()` does not copy everything. Please check out the code to see details.


Cylindrical 2D

- *2-D Spherical coordinates:*
  The 2-D spherical coodinates $(r_{\rm sper}, \theta)$ are fundamentally different from the previous two. Instead of z we have θ as the "vertical" coordinate, and "vertical" is now curved along the sphere. Very close to the equatorial plane the two coordinate systems become tangent, and thus for $z \ll r$ one can *approximately* say that $z/r \simeq \pi/2 - \theta$. It is for this reason that for geometrically thin disks one can make a simplfied approximative mapping from cylindrical to spherical coordinates by simply ignoring the difference between the cylindrical radius r and the spherical radius $r_{\rm spher}$, and by setting $z/r = \pi/2 - \theta$. The advantage is that no interpolation is needed if the grid points in z_{ij} are chosen linearly proportional to cylindrical radius r_i (i.e. $z_{ij}/r_i = z_{0j}/r_0$), and the grid points in θ_j are chosen as $\pi/2 - z_{0j}/r_0$. However, a proper mapping between cylindrical and spherical coordinates does require interpolation, since the 2-D gridpoints of the two coordinate systems do not match. The mapping between the coordinate systems is done with `Disk2D.coord_trafo_cyl2d_to_spher2d()` and `Disk2D.coord_trafo_spher2d_to_cyl2d()`. Note that for reasons related to the multi-D diffusion/Poisson solver, the spherical arrays are 3-D arrays, with the right index always 0.


Spherical 2D

12.1 1+1D disk models

The easiest way to make a 2D (radial-vertical) disk model is to simply make a sequence of 1-D vertical disk models "next to each other". This is called a 1+1D model. This is made easy with the `Disk2D` class. The idea is to first make a 1-D *radial* disk model with the `DiskRadialModel` class (see Chapter 2), and then make a `Disk2D` model with it.

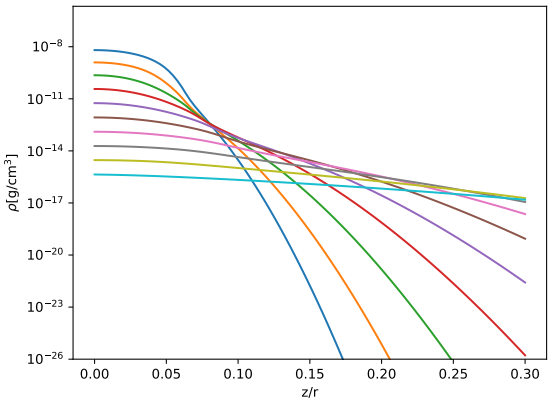
Here is an example. The code snippet is `snippet_vertstruct_1p1d_1.py`. In Python run it as:

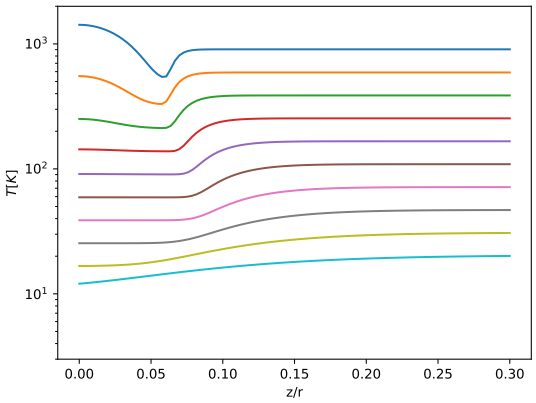
```
%run snippet_vertstruct_1p1d_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, Disk2D, plt, MS, LS, finalize
mstar  = 1 * MS
lstar  = 1 * LS
mdisk  = 0.01 * MS
dtg    = 0.01
```

```
kapdst = 1e2
flang  = 0.05
zrmax  = 0.2
nr     = 10
opac   = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
disk   = DiskRadialModel(mstar=mstar, lstar=lstar, mdisk=mdisk, nr=nr)
disk2d = Disk2D(disk, zrmax=0.3, meanopacitymodel=opac)

for vert in disk2d.verts:
    vert.iterate_vertical_structure()

# plotting

plt.figure()
for vert in disk2d.verts:
    plt.plot(vert.z / vert.r, vert.rhogas)
plt.xlabel('z/r')
plt.yscale('log')
plt.ylim(bottom=1e-26)
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')


plt.figure()
for vert in disk2d.verts:
    plt.plot(vert.z / vert.r, vert.tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(top=2e3, bottom=3e0)
plt.yscale('log')

finalize(results=(disk2d.verts[1].rhogas, disk2d.verts[1].tgas))
```





These figures show the vertical 1-D models at 10 radii logarithmically spaced between 0.1 au and 200 au. One can clearly see that the disk has a flaring geometry and that the disk at small r is affected by viscous heating near the midplane.

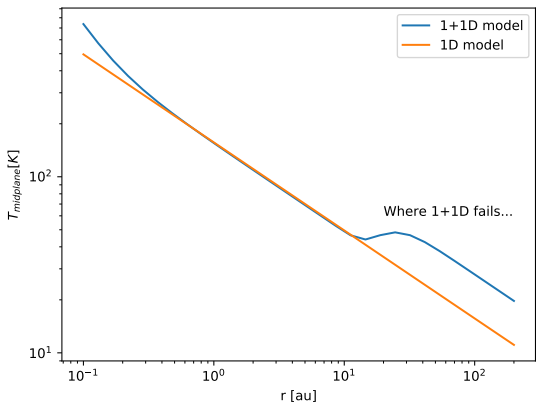
However, when the other regions of the disk become very optically thin, the 1+1D approach breaks down, and will give the wrong midplane temperature. Below is an example of such a case. The code snippet is `snippet_vertstruct_1p1d_2.py`. In Python run it as:

```
%run snippet_vertstruct_1p1d_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, Disk2D, plt, MS, LS, au, finalize
mstar  = 1 * MS
```

```
lstar  = 1 * LS
dtg    = 0.01
kapdst = 1e2
sig0   = 2.0
r0     = 10*au
flang  = 0.05
zrmax  = 0.2
nr     = 30
opac   = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
disk   = DiskRadialModel(mstar=mstar, lstar=lstar, nr=nr)
disk.make_disk_from_simplified_lbp(sig0, r0, 1)
disk2d = Disk2D(disk, zrmax=0.3, meanopacitymodel=opac)

for vert in disk2d.verts:
    vert.iterate_vertical_structure()

disk2d.convert_1p1d_to_cyl2d(rhogas=True,tgas=True)

# plotting

plt.figure()
plt.plot(disk.r/au,disk2d.cyl2d_tgas[:,0],label='1+1D model')
plt.plot(disk.r/au,disk.tmid,label='1D model')
plt.xscale('log')
plt.xlabel('r [au]')
plt.yscale('log')
plt.ylabel(r'$T_{midplane} [K]$')
plt.text(2e1,6e1,'Where 1+1D fails...')
plt.legend()

finalize()
```



The key difference with the previous example is that we now have a Lynden-Bell-Pringle-like exponential cut off of the outer disk. The outer disk thus becomes optically thin. The midplane temperature is then no longer correctly described by the 1+1D model.

12.2 A note on the breakdown of 1+1D models at the inner rim

A key point of 1-D flaring disk models, including their 1+1D extensions, is to use the relatively flat geometry to simplify the radiative transfer of the irradiating photons from the star. This simplification breaks down badly at the "disk inner rim", whether this be the true inner edge of the disk, or merely the location where all the dust evaporates, or simply a place where the dust has been removed by other processes (such as radial drift). The radiative transfer at such an "inner rim" can no longer be approximated with a "flaring angle recipe". It becomes truly 2-D/3-D. As

long as you are not interested in the regions close to such an "inner rim", the 1-D and 1+1D models are a reasonable approximation. But near such "inner rims" full 2-D/3-D radiative transfer becomes necessary. For this, see Section 12.4 and Section 13.1.

12.3 Computing the true azimuthal velocity

In the simplest approximation we can assume that the azimuthal velocity v_ϕ equals the local Kepler velocity $v_K = \sqrt{GM_*/r}$. In Section 2.5 we have, however, seen that in reality the v_ϕ slightly deviates from this Kepler velocity due to the radial pressure gradient. In that Section we have studied this phenomenon only in the disk midplane. Above the midplane, however, things become a bit more difficult.

In Section 11.1.1 we discussed the force balance, decomposed in vertical and cylindrical-radial direction. We used only the vertical component, in order to compute the vertical density structure. To obtain an expression for $v_\phi(r_c,z)$ we will now use the cylindrical-radial component of the force balance. So, adding the $\mathbf{e}_{r_c}$ components of Eqs. (11.2, 11.3, 11.4) together and demanding this sum to be zero, we obtain

$$v_\phi^2 = \frac{GM_*}{r_s^3} r_c^2 + c_s^2 \left.\frac{\partial \ln p}{\partial \ln r_c}\right|_{z=const} \tag{12.1}$$

The double-logarithmic pressure gradient is computed horizontally, keeping z constant. We used $p = \rho c_s^2$ to obtain the above equation. If we define

$$v_K(r_c,z) \equiv \sqrt{\frac{GM_*}{r_s^3} r_c^2} \tag{12.2}$$

then we see that the pressure gradient term in Eq. (12.1) represents, as expected, the deviation from Kepler. But we also see that the Kepler velocity (Eq. 12.2) drops with height z, approximately (for small z) with a factor $1 - 1.5\,(z/r_c)^2$. In practice the pressure gradient will make the disk *sub*-kepler close to the midplane, but *super*-kepler well above the midplane. This is because, as a result of the increase of the pressure scale height with radius ($dH_p/dr > 0$), the radial pressure gradient will become positive at large z, even if it is negative at $z = 0$. In other words: sufficiently above the midplane, the gas pressure gradient pushes the gas *inward*, irrespective of whether the midplane pressure gradient is negative or positive.

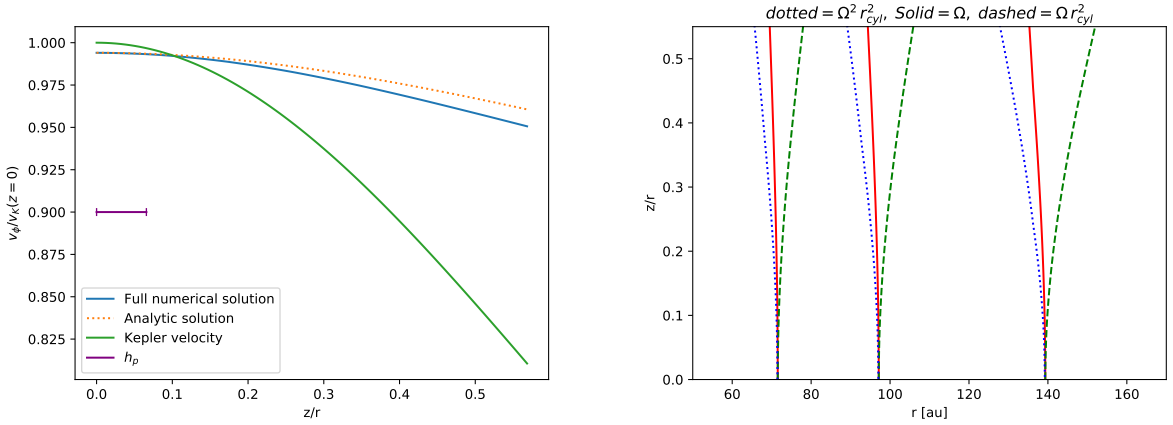
To numerically compute the radial pressure gradient we need a full 2-D or 1+1-D disk model. We give here an example for the case of a vertically isothermal disk. The code snippet is `snippet_vertstruct_vphi_1.py`. In Python run it as:

```
%run snippet_vertstruct_vphi_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, Disk2D, np, plt, MS, LS, au, kk, mp, finalize

dradial = DiskRadialModel(mdisk=0.01*MS)
disk2d  = Disk2D(dradial)

disk2d.compute_vphi()

ir   = 80
cs2  = kk*disk2d.verts[ir].tgas/(2.3*mp)
vk   = disk2d.verts[ir].omk_full*disk2d.verts[ir].r
r    = disk2d.verts[ir].r
omk  = disk2d.verts[ir].omk_midpl
hp   = np.sqrt(cs2[0])/omk
hpr  = hp/r
p    = (np.log(disk2d.verts[ir+1].rhogas[0])-np.log(disk2d.verts[ir].rhogas[0]))/(np.log(disk2d.r[ir-
q    = (np.log(disk2d.verts[ir+1].tgas[0])-np.log(disk2d.verts[ir].tgas[0]))/(np.log(disk2d.r[ir+1])-
zr   = disk2d.cyl2d_zr[ir,:]
#om0  = disk2d.cyl2d_vphi[ir,0]/r
#omana= om0 + (q*omk*zr**2/4.)   # Integral of Eq. 2 Lin & Youdin (2015), ApJ 811, 17
```

```
rsph = r*np.sqrt(1+zr**2)
omana= omk * np.sqrt((p+q)*hpr**2 + (1+q) - (q*r/rsph))   # Eq. 13 Nelson, Gressel & Umurhan (2013),

plt.figure()
plt.plot(disk2d.cyl2d_zr[ir,:],disk2d.cyl2d_vphi[ir,:]/vk[0],label='Full numerical solution')
plt.plot(disk2d.cyl2d_zr[ir,:],omana/omk,':',label='Analytic solution')
plt.plot(disk2d.cyl2d_zr[ir,:],vk/vk[0],label='Kepler velocity')
plt.plot([0,hpr],[0.9,0.9],label=r'$h_p$',color='purple')
plt.plot([0,hpr],[0.9,0.9],'|',color='purple')
plt.xlabel('z/r')
plt.ylabel(r'$v_\phi/v_K(z=0)$')
plt.legend()

plt.figure()
rr       = disk2d.cyl2d_rr
omega    = disk2d.cyl2d_vphi/rr
lphi     = omega*rr**2
ephi     = (omega*rr)**2
vplevels = np.array([2.5,3.,3.5])*1e5 # Levels at velocities in km/s
rvpl     = np.interp(vplevels,disk2d.cyl2d_vphi[::-1,0],disk2d.r[::-1])
omlevels = (vplevels/rvpl)
lplevels = (vplevels*rvpl)[::-1]
eplevels = (vplevels**2)
plt.contour(disk2d.cyl2d_rr/au,disk2d.cyl2d_zr,omega,levels=omlevels,colors='red',linestyles='solid')
plt.contour(disk2d.cyl2d_rr/au,disk2d.cyl2d_zr,lphi,levels=lplevels,colors='green',linestyles='dashed
plt.contour(disk2d.cyl2d_rr/au,disk2d.cyl2d_zr,ephi,levels=eplevels,colors='blue',linestyles='dotted
plt.xlim(50,170)
plt.ylim(0,0.55)
plt.xlabel('r [au]')
plt.ylabel('z/r')
plt.title(r'$dotted=\Omega^2\,r_{cyl}^2,\;Solid=\Omega,\;dashed=\Omega\,r_{cyl}^2$')

finalize(results=(disk2d.cyl2d_vphi[ir,:]))
```



The left plot shows the v_ϕ (Eq. 12.1) as a function of z/r_c at $r_c = 47\,\mathrm{au}$, in units of the Kepler velocity at $z = 0$. Overplotted is the Kepler velocity without the pressure correction (Eq. 12.2). At the midplane ($z/r_c = 0$) one can see that the disk rotates indeed with subkepler velocity, as expected due to the negative radial pressure gradient. But well above the midplane the disk becomes super-Keplerian. Overall the pressure gradient has the effect of making $v_\phi(z)$ dropping less fast with z than $v_K(z)$. In the figure we also overplot the analytic solution of $v_\phi(z)$ according to Eq. (13) of Nelson, Gressel & Umurhan (2013) MNRAS 435, 2610.

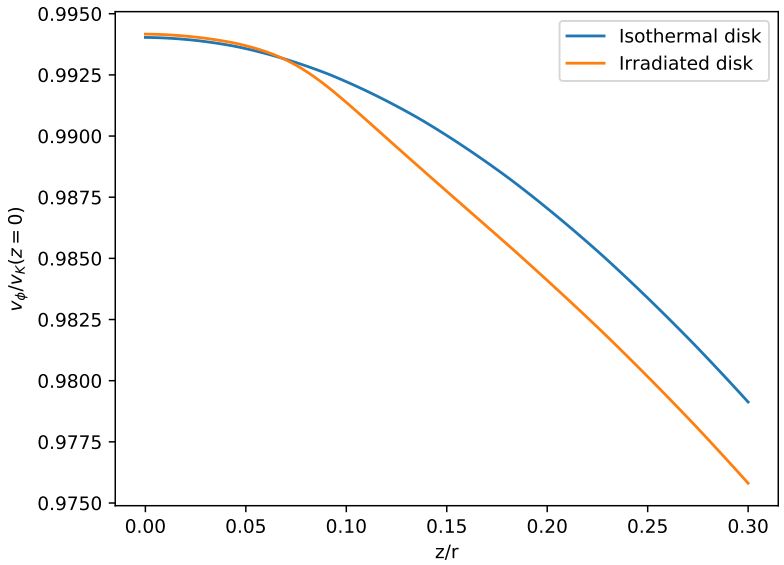
The right plot shows contour lines of the angular frequency $\Omega(r_c,z)$ (solid lines), the specific angular momentum $l(r_c,z) = \Omega r_c^2$ (dashed lines), and the kinetic energy $e_{\rm kin}(r_c,z) = \Omega^2 r_c^2$ (dotted lines). The contour lines are for midplane v_ϕ values of 2.5, 3.0 and 3.5 km/s. This figure shows that above the midplane the contour lines diverge as a result of the vertical shear.

We know, however, that disks are not vertically isothermal. So let us redo the above computation of $v_\phi(z)$ but now for a disk which is irradiated from the top according to a flaring angle of $\varphi = 0.05$ (see Section 11.2). The code snippet is `snippet_vertstruct_vphi_2.py`. In Python run it as:

```
%run snippet_vertstruct_vphi_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, Disk2D, np, plt, MS, LS, au, kk, mp, finalize
import copy

opac    = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
disk    = DiskRadialModel(mdisk=0.01*MS,flang=0.05)
disk2d  = Disk2D(disk, zrmax=0.3, meanopacitymodel=opac)
disk2diso = copy.deepcopy(disk2d)

for vert in disk2d.verts:
    vert.iterate_vertical_structure()

disk2diso.compute_vphi()
disk2d.compute_vphi()

ir = 80
vk = disk.omk[ir]*disk.r[ir]

plt.figure()
plt.plot(disk2d.cyl2d_zr[ir,:],disk2diso.cyl2d_vphi[ir,:]/vk,label='Isothermal disk')
plt.plot(disk2d.cyl2d_zr[ir,:],disk2d.cyl2d_vphi[ir,:]/vk,label='Irradiated disk')
plt.xlabel('z/r')
plt.ylabel(r'$v_\phi/v_K(z=0)$')
plt.legend()

finalize(results=(disk2d.cyl2d_vphi[ir,:]))
```



One sees that the velocity becomes more strongly subkepler high above the midplane. But the overall behavior remains the same as for the vertically isothermal disk.

12.4 1+1D disk models with 2-D approximate radiative transfer

The next level of realism is to replace the 1-D vertical radiative transfer with 2-D radiative transfer. Here we do this still in a simplified (and quick) way, using the two-stage procedure described in Kuiper, Klahr, Dullemond, Kley & Henning (2010), A&A 511, 81. The idea is to do the same two-stage procedure as we did in in the 1-D vertical model of Section 11.1 and following sections, but this time not just in the 1-D vertical approximation but in 2-D axial symmetry. This elimiates the need for the awkward guessing of the flaring angle or flaring index, and it properly allows for radial radiative diffusion in the disk, which can be important for the proper treatment of shadowed regions. One can also do stage 1 (irradiation) in a 2-D way and still do stage 2 (diffusive radiative transfer) in a 1-D vertical way. Strictly speaking one can also do the stage 1 in 1-D flaring angle approach and stage 2 in 2-D axisymmetry, but it is not clear if that makes sense.

12.4.1 2-D method for the irradiation (= stage 1)

The direct irradiation of the disk by the star in a 2-D model is a matter of ray-tracing along radial rays through the disk. The method `radial_raytrace()` does this in a simple way. This method requires that the z-grids at the different radii are radially lined up: i.e. that the grid points obey $z_i(r_1)/r_1 = z_i(r_2)/r_2$, and that we assume that the star is a point source. Then the radial ray-tracing is simply integrating along $z/r = \text{const}$ lines, i.e. at fixed vertical grid point. We therefore define a new, and dimensionless, vertical coordinate

$$\zeta = \frac{z}{r} \tag{12.3}$$

The equation is then:

$$F_*(r,\zeta) = \frac{L_*}{4\pi r_{\rm spher}^2} e^{-\tau_*(r,\zeta)} \tag{12.4}$$

where $r_{\rm spher} = \sqrt{r^2+z^2}$ (compare to Eq. 11.11), where $\tau_*(r,\zeta)$ is the radial optical depth at stellar wavelengths

$$\tau_*(r,\zeta) = \sqrt{1+\zeta^2} \int_{R_*}^{r} \rho_{\rm gas}(r',\zeta)\kappa_* dr' \tag{12.5}$$

Here the $\sqrt{1+\zeta^2}$ factor accounts for the fact that $r_{\rm spher} = \sqrt{r^2+z^2}$, and that, in fact, we integrate along the spherical $r_{\rm spher}$ instead of the cylindrical r.

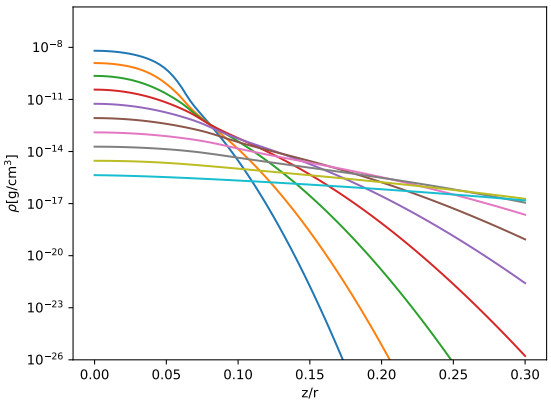
Here is an example. We replace the radius-by-radius flaring-angle irradiation method `irradiate_with_flaring_index()` with the 2-D irradiation method `radial_raytrace()`. The code snippet is `snippet_vertstruct_2d_1.py`. In Python run it as:

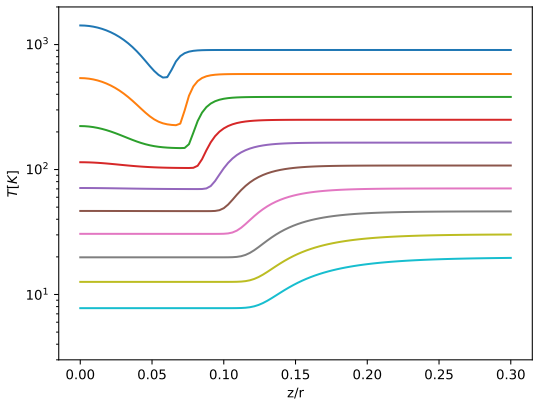
```
%run snippet_vertstruct_2d_1.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, Disk2D, plt, MS, LS, finalize
import copy
mstar  = 1 * MS
lstar  = 1 * LS
mdisk  = 0.01 * MS
dtg    = 0.01
kapdst = 1e2
flang  = 0.025
zrmax  = 0.2
nr     = 10
opac   = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
disk   = DiskRadialModel(mstar=mstar, lstar=lstar, mdisk=mdisk, nr=nr)
disk2d = Disk2D(disk, zrmax=0.3, meanopacitymodel=opac)

for vert in disk2d.verts:
    vert.iterate_vertical_structure()
```

```
disk2d_orig = copy.deepcopy(disk2d)

disk2d.radial_raytrace()
for vert in disk2d.verts:
    vert.solve_vert_rad_diffusion()

# plotting

plt.figure()
for vert in disk2d.verts:
    plt.plot(vert.z / vert.r, vert.rhogas)
plt.xlabel('z/r')
plt.yscale('log')
plt.ylim(bottom=1e-26)
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')

plt.figure()
for vert in disk2d.verts:
    plt.plot(vert.z / vert.r, vert.tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(top=2e3, bottom=3e0)
plt.yscale('log')

finalize()
```





Here we no not iterate the vertical hydrostatic structure. But the irradiation is now done self-consistently. The diffuse radiation field is still done in a 1+1D manner, i.e. thermal energy is still not exchanged between the radii.

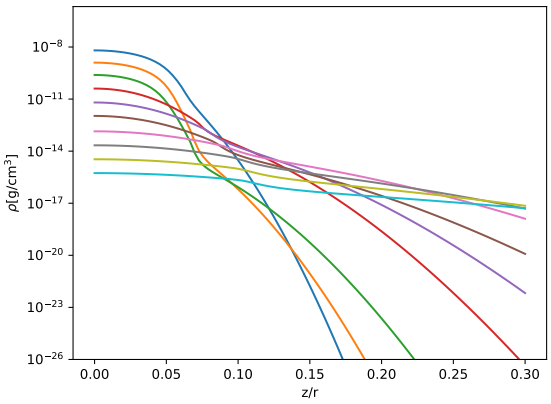
Now let us iterate the vertical structure. The code snippet is `snippet_vertstruct_2d_2.py`. In Python run it as:

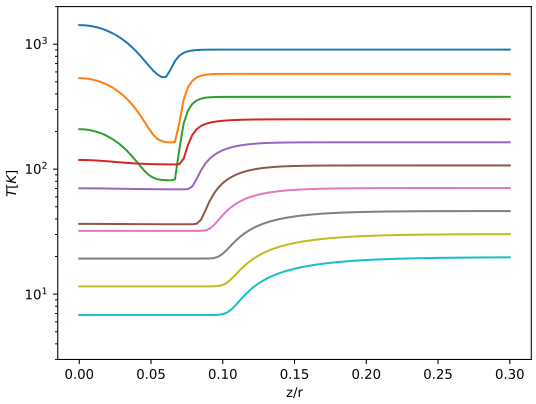
```
%run snippet_vertstruct_2d_2.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, Disk2D, plt, MS, LS, finalize
import copy
mstar  = 1 * MS
lstar  = 1 * LS
mdisk  = 0.01 * MS
dtg    = 0.01
kapdst = 1e2
flang  = 0.025
zrmax  = 0.2
nr     = 10
opac   = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
```

```
disk   = DiskRadialModel(mstar=mstar, lstar=lstar, mdisk=mdisk, nr=nr)
disk2d = Disk2D(disk, zrmax=0.3, meanopacitymodel=opac)

for vert in disk2d.verts:
    vert.iterate_vertical_structure()

disk2d_orig = copy.deepcopy(disk2d)

maxiter = 10
for iter in range(maxiter):
    for vert in disk2d.verts:
        vert.compute_mean_opacity()
    disk2d.radial_raytrace()
    for vert in disk2d.verts:
        vert.solve_vert_rad_diffusion()
        vert.compute_rhogas_hydrostatic()

# plotting

plt.figure()
for vert in disk2d.verts:
    plt.plot(vert.z / vert.r, vert.rhogas)
plt.xlabel('z/r')
plt.yscale('log')
plt.ylim(bottom=1e-26)
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')

plt.figure()
for vert in disk2d.verts:
    plt.plot(vert.z / vert.r, vert.tgas)
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(top=2e3, bottom=3e0)
plt.yscale('log')

finalize()
```





Here the vertical structure is iterated 10 times. We see here a strange effect: the middle of the disk appears to become dark and cold. This is a shadowing effect. The inner disk is vertically too extended, casting a shadow over the middle part of the disk, which accordingly cools and vertically shrinks, exacerbating the shadowing effect. Is this realistic? One problem is that since the diffuse radiative transfer is still done in a 1+1D manner, the diffuse radiation field cannot transport any energy radially through the disk. A shadow is thus *absolute*. In reality, the diffuse radiation field may, however, counteract too strong shadows by "leaking" energy from the illuminated parts of the disk into the shadowed part of the disk. To incorporate this, we need to also replace the 1+1D diffuse radiative

transfer with a true 2-D diffuse radiative transfer, which is the topic of the next subsection.

12.4.2 2-D method for the diffusive radiative transfer (= stage 2)

In Section 11.1 we gave the 1-D equations for the diffuse radiation transfer step (Eqs. 11.14, 11.15). Now we wish to replace this with the true 2-D diffusion equation:

$$\vec{\nabla}\cdot\vec{H} \tag{12.6}$$

with

$$\vec{H} = -\frac{f}{3\rho_{\rm d}\kappa_{\rm Ross}}\vec{\nabla}J \tag{12.7}$$

where f is a factor that, for perfect diffusion theory, is $f=1$, but for Flux-Limited-Diffusion (FLD) is $f=\lambda/3$ with λ being the flux limited from Levermore & Pomraning (1981).

The most natural coordinate system to solve the FLD equations is the spherical coordinate system. This is what DISKLAB does. However, the 1+1D disk model is a series of 1-D vertical structure models that are, by assumption, perfectly vertical. In other words: the 1+1D model is in cylindrical coordinates. So we need to perform a coordinate mapping from cylindrical to spherical coordinates before we solve the FLD equations, and then back to map the result back to the 1+1D model.

For very geometrically thin disks the difference between cylindrical coordinates (r,z) and spherical coordinates $(r_{\rm spher},\theta)$ is tiny. In principle one could, without major error, simply use the same grid for $r_{\rm spher}$ as for r, and simply map each 1-D vertical model to a spherical shell, with θ grid equal to $\theta\simeq\pi/2-z/r$. That is the simplest "coordinate transformation" from cylindrical to spherical, where no interpolation is necessary. The only tricky bit is that the indexing order of θ goes from pole to equator, while z/r goes from equator upward. In Python this is simply done with indexing `[::-1]`. This method of approximate transformation between cylindrical and spherical coordinates is called the *thin-disk approximation*.

However, the proper way is to do a true coordinate transformation between cylindrical and spherical coordinates (and back), which requires a mapping of the physical variables such as density and temperature from one coordinate system to the other. This requires interpolation. In the `disk2d` module of DISKLAB this is done using the `disk2d.coord_trafo_cyl2d_to_spher2d()` and `disk2d.coord_trafo_spher2d_to_cyl2d()` functions.

Although this full-fledged mapping back and forth between the cylindrical and spherical coordinates is formally better than using the thin-disk approximation, it does have some numerical issues. Well above the midplane, there will be regions of space covered by the spherical grid that are outside of the cylindrical grid, and vice versa. A back-and-forth mapping will therefore introduce strong boundary effects if not handled carefully. In the FLD method these boundary effects are taken care of by employing our knowledge of the optically thin solution.

Now let us get to business. We use the irradiation computation from Section 12.4.1. Then we set up a spherical coordinate system. We map the relevant quantities to the spherical coodinates. Then we call the FLD solver. The result is then mapped back to the 1+1D model. Most of this stuff is handled by `disk2d.solve_2d_rad_diffusion()`. You can switch to the simplified thin-disk approximation by setting `simplecoordtrans=True` in that function. By default it is set to `False` (i.e. by default the true coordinate mapping is used).

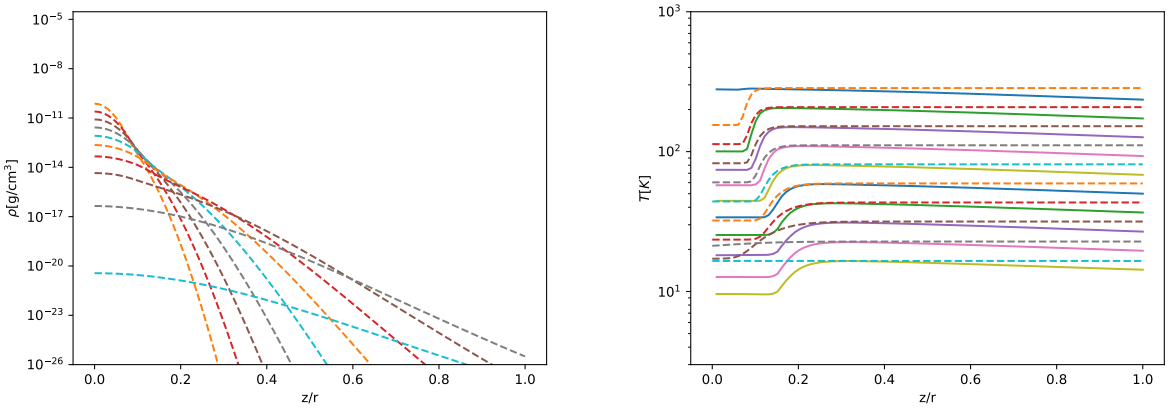
The code snippet is `snippet_vertstruct_2d_3.py`. In Python run it as:

```
%run snippet_vertstruct_2d_3.py
```

Here is the listing:

```
from snippet_header import DiskRadialModel, Disk2D, plt, MS, LS, au, finalize
import copy
mstar  = 1 * MS
lstar  = 1 * LS
sig0   = 10.
r0     = 50. * au
dtg    = 0.01
kapdst = 1e2
flang  = 0.025
```

```
zrmax  = 1.0
nr     = 100
rin    = 1 * au
rout   = 500 * au
alpha  = 1e-6
opac   = ['supersimple', {'dusttogas': 0.01, 'kappadust': 1e2}]
disk   = DiskRadialModel(mstar=mstar, lstar=lstar, nr=nr, rin=rin, rout=rout, alpha=alpha)
disk.make_disk_from_simplified_lbp(sig0, r0, 0.5)
disk2d = Disk2D(disk, zrmax=zrmax, meanopacitymodel=opac)

for vert in disk2d.verts:
    vert.iterate_vertical_structure()

disk11d = copy.deepcopy(disk2d)

maxiter = 1
for iter in range(maxiter):
    print("Vertical structure iteration {}".format(iter))
    for vert in disk2d.verts:
        vert.compute_mean_opacity()
    disk2d.radial_raytrace()
    disk2d.setup_spherical_coordinate_system()
    disk2d.solve_2d_rad_diffusion(linsol_convcrit=1e-6, linsol_itermax=2000,
                                  nonlin_convcrit=1e-3, nonlin_itermax=20,
                                  simplecoordtrans=False)
    for vert in disk2d.verts:
        vert.compute_rhogas_hydrostatic()

plt.figure()
for ir in range(0, nr, 10):
    vert = disk2d.verts[ir]
    plt.plot(vert.z / vert.r, vert.rhogas)
    vert = disk11d.verts[ir]
    plt.plot(vert.z / vert.r, vert.rhogas, '--')
plt.xlabel('z/r')
plt.yscale('log')
plt.ylim(bottom=1e-26)
plt.ylabel(r'$\rho [\mathrm{g}/\mathrm{cm}^3]$')
plt.savefig('fig_snippet_vertstruct_2d_3_1.pdf')

plt.figure()
for ir in range(0, nr, 10):
    vert = disk2d.verts[ir]
    plt.plot(vert.z / vert.r, vert.tgas)
    vert = disk11d.verts[ir]
    plt.plot(vert.z / vert.r, vert.tgas, '--')
plt.xlabel('z/r')
plt.ylabel(r'$T [K]$')
plt.ylim(top=1e3, bottom=3e0)
plt.yscale('log')
plt.savefig('fig_snippet_vertstruct_2d_3_2.pdf')

finalize()
```



This method can also be iterated with vertical hydrostatic equilibrium. The code snippet is `snippet_vertstruct_2d_4.py`. In Python run it as:

```
%run snippet_vertstruct_2d_4.py
```

12.5 Self-gravity of the disk in 2D

If a disk mass is not small compared to the stellar mass, the effect of self-gravity kicks in. In extreme cases it can lead to the gravitational instability, in which case DISKLAB can not be used, because that is a time-dependent phenomenon, and likely also non-axisymmetric.

However, at disk masses below instability (i.e. Toomre Parameter $Q > 2$), self-gravity can still affect the structure of the disk, even though it does not destabilize it. A stable self-gravitating disk will be slightly flatter, and it will have a different Keplerian rotation curve than a non-self-gravitating disk.

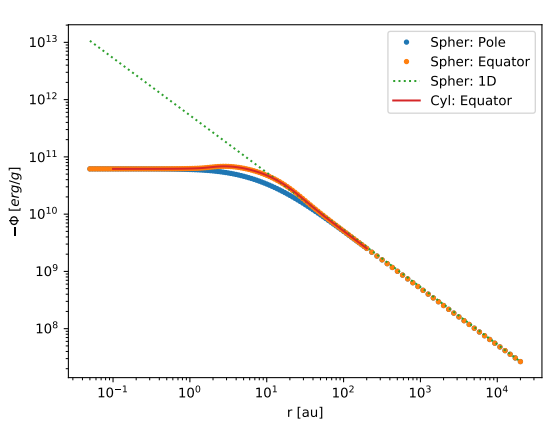
The `Disk2D` class has a method `Disk2D.solve_2d_selfgrav()` that computes the self-gravity forces in 2-D. Internally it uses 2-D spherical coodinates, but it initially performs a transformation from 1+1D to 2-D cylindrical, and then to 2-D spherical coordinates. The key input to the Poisson solver is then the density `Disk2D.spher_rho[:,:,0]`. It then solves the Poisson equation, obtaining the potential in 2-D spherical coordinates (`Disk2D.spher_pot[:,:,0]`). This is then mapped back onto the 2-D cylindrical coordinates `Disk2D.cyl2d_pot[:,:]`, and the gradients of this potential are then numerically evaluated at the cell walls: `Disk2D.cyl2d_gradpot_r[:,:]` for the cylindrically radial gradient and `Disk2D.cyl2d_gradpot_z[:,:]` for the vertical gradient. The negative of these gradients are then the bodyforces belonging to the potential.

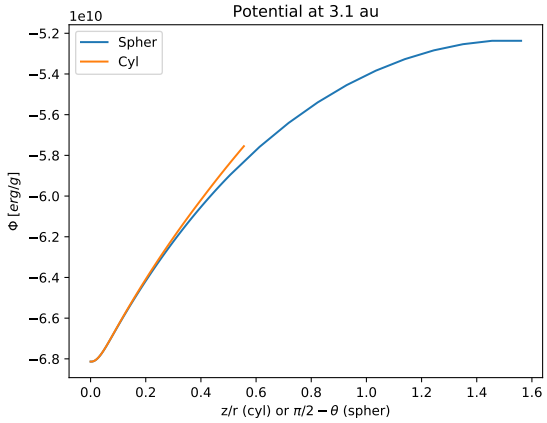
Typically it is prudent to set up a spherical grid that is radially extended well beyond the grid of the cylindrical coordinates. Also the θ-grid has to be extended all the way to (close to) the pole. This is because gravity is a long-range force, and the boundary conditions we impose on the gravitational potential are simple: we set $\vec{n} \cdot \nabla\Phi = 0$ on both boundaries in θ as wel as the inner boundary in $r_{\rm spher}$, and we set $\Phi = 0$ at the outer boundary in $r_{\rm spher}$. In particular the latter is only (approximately) correct if the $\max(r_{\rm spher})$ is very far out, i.e. much larger than the disk radius.

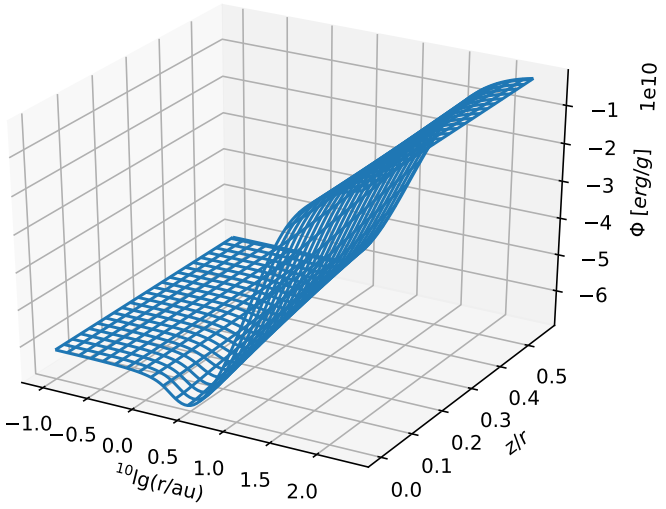
Here is an example of the computation of the self-gravity forces for a given density distribution. The code snippet is `snippet_selfgrav2d_1.py`. In Python run it as:

```
%run snippet_selfgrav2d_1.py
```

It will spawn a set of wireframe surface plots to show the $({}^{10}\log(r), z/r)$ dependence of the gravitational potential Φ, and the two components of the bodyforce f_r and f_z. It will also plot the radial run of the gravitational potential at different θ angle, and compare it to the potential for when the mass would be a point source at the center. As expected: the potential for the real 2-D distribution asymptotically matches the potential for a point source of the same mass for large r. Here is a subset of these plots:







The snippet `snippet_selfgrav_2.py` has the same setup, but now makes a global iteration to get the disk in vertical hydrostatic balance with the self-gravity.

```
%run snippet_selfgrav2d_2.py
```

It plots the vertical structure at $r = 14.7$ au, in the same manner as shown in Section 11.4, but now for the full global self-gravity.

Chapter 13

Connecting to external code packages

13.1 Connecting to the radiative transfer code RADMC-3D

[TODO]

13.2 Connecting to the dust evolution code ????

[TODO]

13.3 Connecting to the hydrodynamics code FARGO-3D

[TODO]

Appendix A

Standard diffusion equation solver in 1-D

Here we describe a general-purpose Python subroutine `solvediffonedee()` for solving (or advancing in time) a diffusion equation of the type:

$$\frac{\partial y(x,t)}{\partial t} + \frac{\partial y(x,t)v(x)}{\partial x} - \frac{\partial}{\partial x}\left(D(x)g(x)\frac{\partial}{\partial x}\left(\frac{y(x,t)}{g(x)}\right)\right) = s(x) \tag{A.1}$$

If a time step is provided, the equation is advanced in time *one time step*. Since the method is fully implicit, the subroutine is stable even for large time steps, though it is more accurate when using many small ones than one big one. If no time step is provided, the stationary case is solved (i.e. $\partial/\partial t \to 0$).

The boundary conditions are:

$$p\frac{dy}{dx} + qy = r \tag{A.2}$$

(type 0) or

$$p\frac{d(y/g)}{dx} + q(y/g) = r \tag{A.3}$$

(type 1). For each of the two boundaries (left and right) you specify a tuple (p,q,r,type).

It is also possible to insert special "internal conditions" somewhere inside the grid.

Example of a time *in*dependent boundary value problem:

```
import numpy as np
import matplotlib.pyplot as plt
from solvediffonedee import *
nx  = 100
x   = np.linspace(0.,1.,nx)
y   = np.zeros(nx)
v   = np.zeros(nx)+0.5
d   = np.ones(nx)
g   = np.ones(nx)
s   = np.ones(nx)
bcl = (0,1,1,0)
bcr = (-1,0,1,0)
y   = solvediffonedee(x,y,v,d,g,s,bcl,bcr)
plt.plot(x,y)
plt.show()
```

which gives a slightly asymmetric inverted parabola. For time-dependent problems you have to set the keyword `dt` to the length of the time step. Please refer to the documentation string for more details, by typing `solvediffonedee?` in the python prompt.

The `solvediffonedee()` function stands at the basis of many of the capabilities of DISKLAB. It allows the viscous disk equations to be time-integrated without insanely small time steps, and it is therefore the key ingredient for the speed of DISKLAB.

Appendix B

Opacities

[MODIFY THIS CHAPTER TO THE NEW OPACITY ROUTINES]

Dust opacities change with grain size and grain composition. To compute your own dust opacity tables DISKLAB provides you with a Mie code that does this for you. The code is called `bhmie.py` and is the Python version of the famous Bohren & Huffman Mie code. It was translated from the f77 code of Bruce Draine into Python. A wrapper routine was created to make it easier to use: `makedustopac.py`.

Before you can make a dust opacity you have to download the corresponding optical constants. You can get the tables from the Jena optical constants database:

`http://www.astro.uni-jena.de/Laboratory/OCDB/index.html`

Here is an example of making the opacity table of a spherical dust grain of 10 μm radius using the `pyrmg70.lnk` datafile:

`http://www.astro.uni-jena.de/Laboratory/OCDB/data/silicate/amorph/pyrmg70.lnk`

for Pyroxene with 70% magnesium and 30% iron.

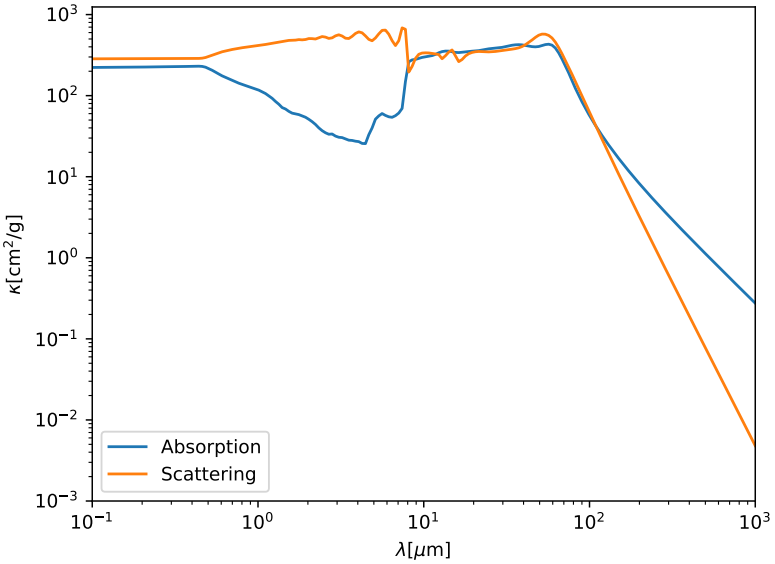
The code snippet is in `snippet_make_dust_opacity_1.py`. In Python run it as:

```
%run snippet_make_dust_opacity_1.py
```

Here is the listing:

```
from disklab.makedustopac import compute_opac_mie, write_radmc3d_kappa_file
from snippet_header import np, plt, finalize
agraincm  = 10 * 1e-4      # Grain size in cm
logawidth = 0.05           # Smear out the grain size by 5% in both directions
na        = 20             # Use 20 grain size samples
optconst  = "pyrmg70"      # The optical constants name
matdens   = 3.0            # The material density in gram / cm^3

# Extrapolate optical constants beyond its wavelength grid, if necessary

extrapol     = True
verbose      = False         # If True, then write out status information
lamcm        = 10.0**np.linspace(-1, 3, 200) * 1e-4
optconstfile = optconst + '.lnk'

print("Running the code. Please wait...")
opac = compute_opac_mie(optconstfile, matdens, agraincm, lamcm,
                        extrapolate=extrapol, logawidth=logawidth, na=na)
write_radmc3d_kappa_file(opac, optconst)

# plotting

plt.figure()
```

```
plt.plot(opac['lamcm'] * 1e4, opac['kabs'], label='Absorption')
plt.plot(opac['lamcm'] * 1e4, opac['kscat'], label='Scattering')
plt.xlabel(r'$\lambda [\mu\mathrm{m}]$')
plt.ylabel(r'$\kappa [\mathrm{cm}^2/\mathrm{g}]$')
plt.xscale('log')
plt.yscale('log')
plt.xlim(1e-1, 1e3)
plt.ylim(bottom=1e-3)
plt.legend(loc='lower left')

finalize()
```



For details on how to compute Rosseland and Planck mean opacities, see Section 9.3.

Appendix C

Embedding your DISKLAB model in an interactive widget

Often it is a nuisance to have to re- and re-run a model by hand to get a feeling for its behavior. It would be useful to be able to make an *interactive* plot that changes automatically when you change the parameters through a set of sliders (one slider per parameter). The module `disklab/interactive_plot.py` does that for you.

The main thing you have to do is to make a Python function that creates your model for a given set of parameters. Let us call this function `modulfunc()`. The function should return an array that you wish to plot. The array is, for example, the surface density as a function of radius. The radius is then an array of r-values that is given as an argument to `modulfunc()`. So within `modulfunc()` is where you design your model. You choose which parameters should have sliders, and what the possible values of these parameters are (e.g. an array of 100 values between 3. and 8., meaning that the slider will slide between 3 and 8, and have 100-2 intermediate values). You also determine exactly what the plot should look like, what the labels, limits, scales, colors etc are. The widget `interactive_plot()` only replaces the values of the curve in the plot with the new ones after it calls `modulfunc()`. In a manner of speaking, the only thing that `interactive_plot()` does is to bring your figure to life. The way that this is done is that you hand `interactive_plot()` the `axes` object returned by `plt.plot()`, and hand it over to `interactive_plot()`. Whenever the "plot" button is clicked by the user, `interactive_plot()` will read the current values of the sliders, hand them over to `modelfunc()`, which will compute the new data. Then these new data are given to the `axes` objects belonging to the curve, which will then replot it.

`interactive_plot()` also allows to have multiple curves being "brought to life". Just plot each of the curves you want to plot, hand over the `axes` objects of these curves in a list to `interactive_plot()`. Your `modelfunc()` should now return a numpy array of two results, instead of just the results for one curve.

It is easiest to simply look at an example.

The code snippet is in `snippet_widget_1.py`. In Python run it as:

```
%run snippet_widget_1.py
```

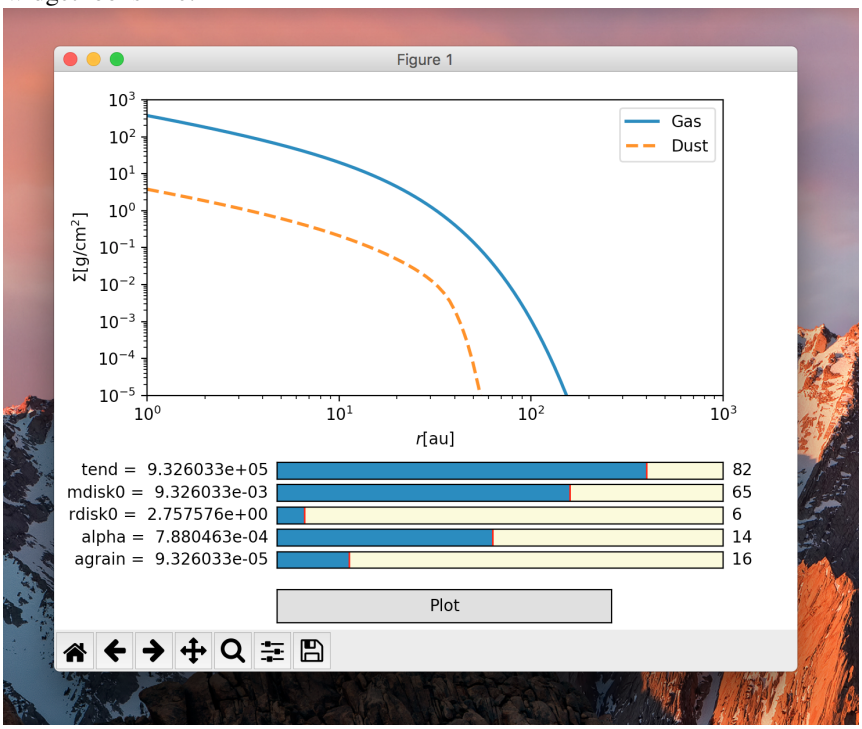
Here is the listing:

```
from snippet_header import np, plt, DiskRadialModel, au, year, MS, LS, finalize
from disklab.interactive_plot import *
#
# Model function in a form that can be used by the interactive_plot widget
#
def modelfunc(rau,param,fixedpar=None):
    #
    # Fixed parameters
    #
    mstar     = 1*MS  # Default stellar mass: Solar mass
    lstar     = 1*LS  # Default stellar luminosity: Solar luminosity
    if fixedpar is not None:
```

```
        if 'mstar' in fixedpar: mstar=fixedpar['mstar']
        if 'lstar' in fixedpar: lstar=fixedpar['lstar']
    #
    # Variable parameters (sliders of widget)
    #
    tend      = param[0]
    mdisk0    = param[1]
    rdisk0    = param[2]
    alpha     = param[3]
    agrain    = param[4]
    #
    # Model setup
    #
    disk      = DiskRadialModel(mstar=mstar,lstar=lstar,rgrid=rau*au)
    disk.make_disk_from_lbp_alpha(mdisk0,rdisk0,alpha,1*year)
    disk.add_dust(agrain=agrain)
    #
    # Run the model
    #
    ntime     = 100
    time      = np.linspace(0.,tend,ntime+1)   # Linear time intervals
    for itime in range(1,ntime+1):
        dt = time[itime]-time[itime-1]
        disk.compute_viscous_evolution_and_dust_drift_next_timestep(dt)
    #
    # Return the result (you can return whatever you want to plot; here
    # we plot two results simultaneously: the gas and dust surface
    # densities)
    #
    return np.array([disk.sigma, disk.dust[0].sigma])

#
# Create the plot we wish to make interactive
#
xmin     = 1.0
xmax     = 1e3
rau      = xmin * (xmax/xmin)**np.linspace(0.,1.,100)
par      = [1e6*year,1e-2*MS,3*au,1e-3,1e-4]
fixedpar = {'mstar':MS,'lstar':LS}
result   = modelfunc(rau,par,fixedpar=fixedpar)
sigmagas = result[0]
sigmadust= result[1]
ymin     = 1e-5
ymax     = 1e+3
fig      = plt.figure()
ax       = plt.axes(xlim=(xmin,xmax),ylim=(ymin,ymax))
axgas,   = ax.loglog(rau,sigmagas,linewidth=2,label='Gas')
axdust,  = ax.loglog(rau,sigmadust,'--',linewidth=2,label='Dust')
axmodel  = [axgas,axdust]
plt.xlabel(r'$r [\mathrm{au}]$')
plt.ylabel(r'$\Sigma [\mathrm{g}/\mathrm{cm}^2]$')
plt.legend()
#
# Now make the plot interactive with sliders. We have to
# specify the slider names, the possible values and (optionally)
# the units. Then call interactive_plot() to bring it to life,
# with the function modelfunction() (see above) being the life-giver.
# Type interactive_plot? to find out more about interactive_plot():
# there are numerous examples in the document string.
#
parnames = ['tend =','mdisk0 =','rdisk0 =','alpha =','agrain = ']
```

```
params    = [1e1*1e6**np.linspace(0.,1.,100)*year,
             1e-4*1e3**np.linspace(0.,1.,100)*MS,
             np.linspace(1.,30.,100)*au,
             1e-6*1e6**np.linspace(0.,1.,100),
             1e-5*1e6**np.linspace(0.,1.,100)]
parunits  = [year,MS,au,1.,1.]
ipar      = interactive_plot(rau,modelfunc,params,parnames=parnames,
                             parunits=parunits,fixedpar=fixedpar,
                             parstart=par,plotbutton=True,
                             fig=fig,ax=ax,axmodel=axmodel,returnipar=True)

finalize([])
```

Here is what this widget looks like:



You can do many things with the `disklab/interactive_plot.py` module. Several examples are in the documentation string. In fact, `interactive_plot.py` is a stand-alone code that can be used for any kind of interactive plotting, not just for DISKLAB. And within DISKLAB you can use it for any kind of plotting you may want, not just for plots as a function of radial coordinate r. The philosophy is: just make a plot you want to make, and store the `axes` objects from the curves you want to bring to life into a list `axmodel`, and pass that to `interactive_plot()`. You can even make multiple plot panels using the `plt.subplots()` function, and have them simultaneously linked to the sliders.

Here is a variant of the above example, where, this time, the thermal emission is plotted on top of some "observed data". The code snippet is in `snippet_widget_2.py`. In Python run it as:

```
%run snippet_widget_2.py
```

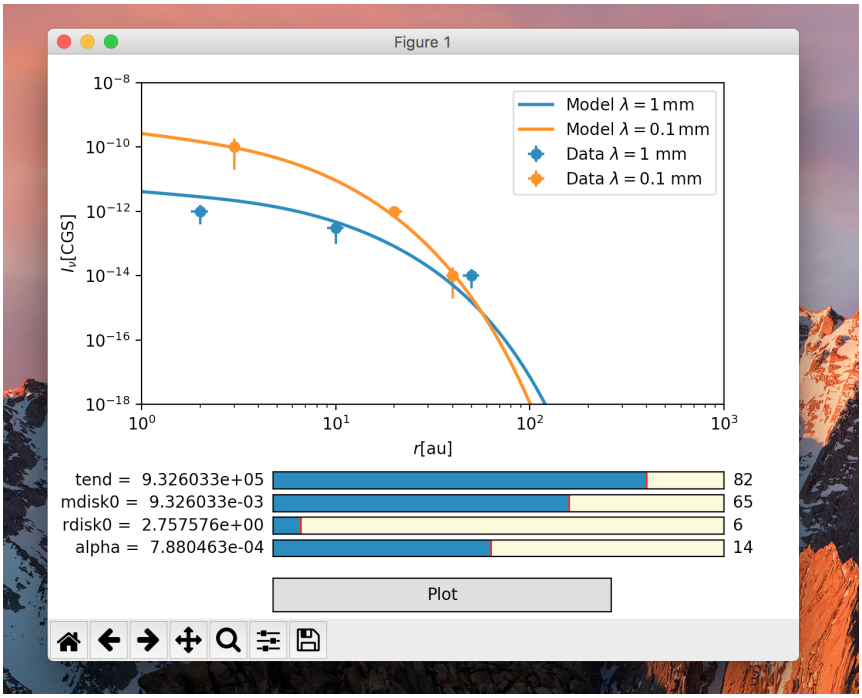


Figure 1
Model λ = 1 mm
Model λ = 0.1 mm
Data λ = 1 mm
Data λ = 0.1 mm
I_ν[CGS]
r[au]
tend = 9.326033e+05
mdisk0 = 9.326033e-03
rdisk0 = 2.757576e+00
alpha = 7.880463e-04
82
65
6
14
Plot