dustpylib

Sebastian Stammler & Tilman Birnstiel

Aug 08, 2023

CONTENTS

1 Installation **3**

1.1 Dynamics . . . 3
1.2 Grid . . . 12
1.3 Planetesimals . . . 14
1.4 Radiative Transfer . . . 20
1.5 Substructures . . . 40
1.6 Appendix A: Contributing/Bugs/Features . . . 48
1.7 Module Reference . . . 49

2 Indices and tables **67**

Python Module Index **69**

Index **71**

`DustPyLib` is a collection of tools and extensions for the `DustPy` package for dust evolution on protoplanetary disks. When using any of the modules provided by `DustPyLib`, please cite the required sources given in the respective example notebook.

For information about the usage of `DustPy`, please have a look at the DustPy documentation.

CHAPTER

ONE

INSTALLATION

To install `DustPyLib` type
`pip install dustpylib`

1.1 Dynamics

The `dustpylib.dynamics` submodule contains methods regarding gas and dust dynamics.

1.1.1 Backreaction

This notebook shows how to use the setup routine to include the effect of the dust backreaction (i.e. the dynamical feedback) into the dustpy `Simulation` object.

We also illustrate the modifications applied to the simulation object by the `setup_backreaction()` routine.

This module follows the implementation by Garate et al. (2019, 2020)

Basic backreaction setup

[1]:
```
from dustpy import Simulation
from dustpy import constants as c
```

[2]:
```
sim = Simulation()
```

The key parameters that impact the intensity of the dust backreaction are: * Initial global dust-to-gas ratio ϵ_0 * Dust fragmentation velocity v_{frag} * Gas turbulence parameter α * Dust turbulence δ_{turb} (if different form α)

[3]:
```
sim.ini.gas.alpha = 1.e-3
sim.ini.dust.d2gRatio = 0.01
sim.ini.dust.vfrag = 1000.0
```

[4]:
```
sim.initialize()
```

Afte initializing we call the `setup_backreaction() routine` that will load the backreaction coefficient updaters into the `sim`.

[5]:
```
from dustpylib.dynamics.backreaction import setup_backreaction
```

[6]:
```
setup_backreaction(sim)
```

```
Setting up the backreaction module.
Please cite the work of Garate et al. (2019, 2020).
```

The setup routine now calculates the backreaction coefficients `A` and `B` which affect the gas radial and azimuthal velocities as in the follows.

$v_{r,\mathrm{gas}} = Av_\nu + 2Bv_\eta$

$\Delta v_{\varphi,\mathrm{gas}} = -Av_\eta + \frac{B}{2}v_\nu$

Here v_ν corresponds to the gas viscous velocity, and $v_\eta = \eta v_K$ corresponds to standard gas deviation from the keplerian speed due its own pressure support.

In a disk with a single dust species, the backreaction coefficients can be written as:

$A = \frac{\epsilon+1+\mathrm{St}^2}{(\epsilon+1)^2+\mathrm{St}^2}$

$B = \frac{\epsilon\mathrm{St}}{(\epsilon+1)^2+\mathrm{St}^2}$

In a disk with low dust content and/or small particle sizes ($\epsilon \to 0, \mathrm{St} \to 0$) we have that the coefficients tend to:

$A \to 1$

$B \to 0$

where we recover the values for the gas evolution of a dust free disk.

In `dustpy` the backreaction coefficients are stored in the `sim.dust.backreaction` group

[7]:
```
sim.dust.backreaction
```

[7]:
```
Group (Backreaction coefficients)
---------------------------------
    A             : Field (Pull factor)
    B             : Field (Push factor)
  -----
```

The backreaction coefficients are updated simultaneously when the `sim.dust.backreaction.update()` function is called.

The corresponding updater can be found in `dustpylib.dynamics.backreaction.functions_backreaction.BackreactionCoefficients()`

Now the simulation is fully setup can can be executed with `sim.run()`

Effect on dust diffusivity

The dust content in the disk is also expected to slow down the dust diffusivity.

To implement this effect, the diffusivity is modified as follows:

$D_d = \frac{\delta_\mathrm{rad} c_\mathrm{s}^2 \Omega_\mathrm{K}^{-1}}{(1+\epsilon)(1+\mathrm{St}^2)}$

The dust diffusivity updater is included automatically in the `setup_backreaction()` routine.

The updater can be found in `dustpylib.dynamics.backreaction.functions_backreaction.dustDiffusivity_Backreaction()`

Vertical Backreaction Setup

The previous setup assumes that the gas and dust uniformly mixed in the vertical direction, and that therefore are equally affected by the effect of the dust back-reaction.

However, we know that the dust tends to settle, and that therefore the midplane should be more intensly affected by the dust feedback than the upper layers.

To do so we assume that the gas and dust are vertically distributed following a gaussian profile:

$\rho_{\mathrm{g,d}}(z) = \frac{\Sigma_{\mathrm{g,d}}}{\sqrt{2\pi}h_{\mathrm{g,d}}} \exp\left(\frac{z^2}{2h^2_{\mathrm{g,d}}}\right)$

With these densities we can calculate the backreaction coefficients at every height $A(z), B(z)$, and obtain a vertically weighted average to calculate the velocities with:

$\left(\bar{A}_{\mathrm{g,d}}, \bar{B}_{\mathrm{g,d}}\right) = \frac{1}{\Sigma_{\mathrm{g,d}}} \int \rho_{\mathrm{g,d}} \left(A(z), B(z)\right) \mathrm{d}z$

Because the gas and dust have different characteristic scale heights, this results in one pair of `A,B` backreaction coefficients for both the gas, and for each dust species.

The origin of this approach can be found in Garate et al., 2020 (Section 2.2.1)

To implement the vertically weighted backreaction coefficients, it is only necessary to mark the corresponding flag in the setup as follows:

```
setup_backreaction(sim, vertical_setup=True)
```

[8]:
```
sim = Simulation()
sim.initialize()
setup_backreaction(sim, vertical_setup=True)
```

```
Setting up the backreaction module.
Please cite the work of Garate et al. (2019, 2020).
```

which creates the additional fields for the new backreaction coefficients

[9]:
```
sim.dust.backreaction
```

[9]:
```
Group (Backreaction coefficients)
---------------------------------
    A             : Field (Pull factor (gas), accounting for dust settling)
    A_dust_se...  : Field (Pull factor (dust), accounting for dust settling)
    B             : Field (Push factor (gas), accounting for dust settling)
    B_dust_se...  : Field (Push factor (dust), accounting for dust settling)
  -----
```

`dust.backreaction.A` and `dust.backreaction.B` correspond to the effective backreaction experienced by the gas.

`dust.backreaction.A_dust_settling` and `dust.backreaction.B_dust_settling` correspond to the effective backreaction experienced by the each dust species, settling taken into account.

This setup also modifies the updater of `dust.v.rad` such that the maximum drift velocity is calculated for each dust species.

Guided example of a backreaction study

Backreaction is a complex interaction that arises from the exchange of angular momentum between gas and dust. In order for backreaction to be effective the dust-to-gas ratios need to be high, and the Stokes number of the grains (i.e. their dynamical size) large.

In this guided example we will learn how to quantify the dynamical effect of the backreaction, by looking at the gas surface density, gas radial velocity, and the back-reaction coefficients in three simulations:

- One simulation without backreaction and ISM dust-to-gas ratio ($\epsilon_0 = 1\%$)
- One simulation with backreaction and ISM dust-to-gas ratio ($\epsilon_0 = 1\%$)
- One simulation with backreaction, a high dust-to-gas ratio ($\epsilon_0 = 5\%$), and a snowline

[10]:
```
import numpy as np
```

Running the simulations

To simplify this notebook, we will define a routine that creates and initializes a standard simulation object, with custom initial dust-to-gas ratio:

[11]:
```
def get_simulation(d2gRatio = 0.01):
    sim = Simulation()


    # Relevant Gas and Dust parameters
    sim.ini.gas.gamma = 1.0
    sim.ini.gas.alpha = 1.e-3
    sim.ini.dust.vfrag = 1000.0
    sim.ini.dust.d2gRatio = d2gRatio


    # Radial Grid Parameters
    sim.ini.grid.Nr = 200
    sim.ini.grid.rmin = 5 * c.au
    sim.ini.grid.rmax = 500 * c.au

    # Initialization and snapshots
    sim.initialize()
    sim.t.snapshots = np.linspace(0.1, 2.0, 20) * 1.e5 * c.year


    return sim
```

First we will create our first control simulation, without backreaction.

This simulation will evolve by standard viscous evolution, without any particular features.

[12]:
```
sim_control = get_simulation(d2gRatio = 0.01)
sim_control.writer.datadir = "backreaction_examples/control/"
```

Execute `sim_control.run()`

Secondly, for the second simulation we will include the basic backreaction setup, after creating the simulation object.

[13]:
```
sim_backreaction = get_simulation(d2gRatio = 0.01)
setup_backreaction(sim_backreaction)
sim_backreaction.writer.datadir = "backreaction_examples/backreaction/"
```

```
Setting up the backreaction module.
Please cite the work of Garate et al. (2019, 2020).
```

Execute `sim_backreaction.run()`

And lastly for the third simulation we will create a more complex setup with an iceline (modeled as a change in the the fragmentation velocity at a certain location).

[14]:
```
sim_backreaction_iceline = get_simulation(d2gRatio = 0.05)
setup_backreaction(sim_backreaction_iceline)

r_iceline = 20 * c.au
r = sim_backreaction_iceline.grid.r
sim_backreaction_iceline.dust.v.frag[r < r_iceline] = 100.
sim_backreaction_iceline.update()
sim_backreaction_iceline.writer.datadir = "backreaction_examples/backreaction_iceline/"
```

```
Setting up the backreaction module.
Please cite the work of Garate et al. (2019, 2020).
```

Execute `sim_backreaction_iceline.run()`

Analyzing the simulations

[15]:
```
import matplotlib.pyplot as plt
from dustpy import hdf5writer
reader = hdf5writer()
```

[16]:
```
plt.rcParams["figure.dpi"] = 150.
```

We read the last snapshot in the simulation from the `backreaction_example` folder.

(At this point we already ran the simulations through a script and saved a compact version of the output files for this notebook plotting routine).

We read the snapshot 10, which corresponds to a time of `0.1 Myr`

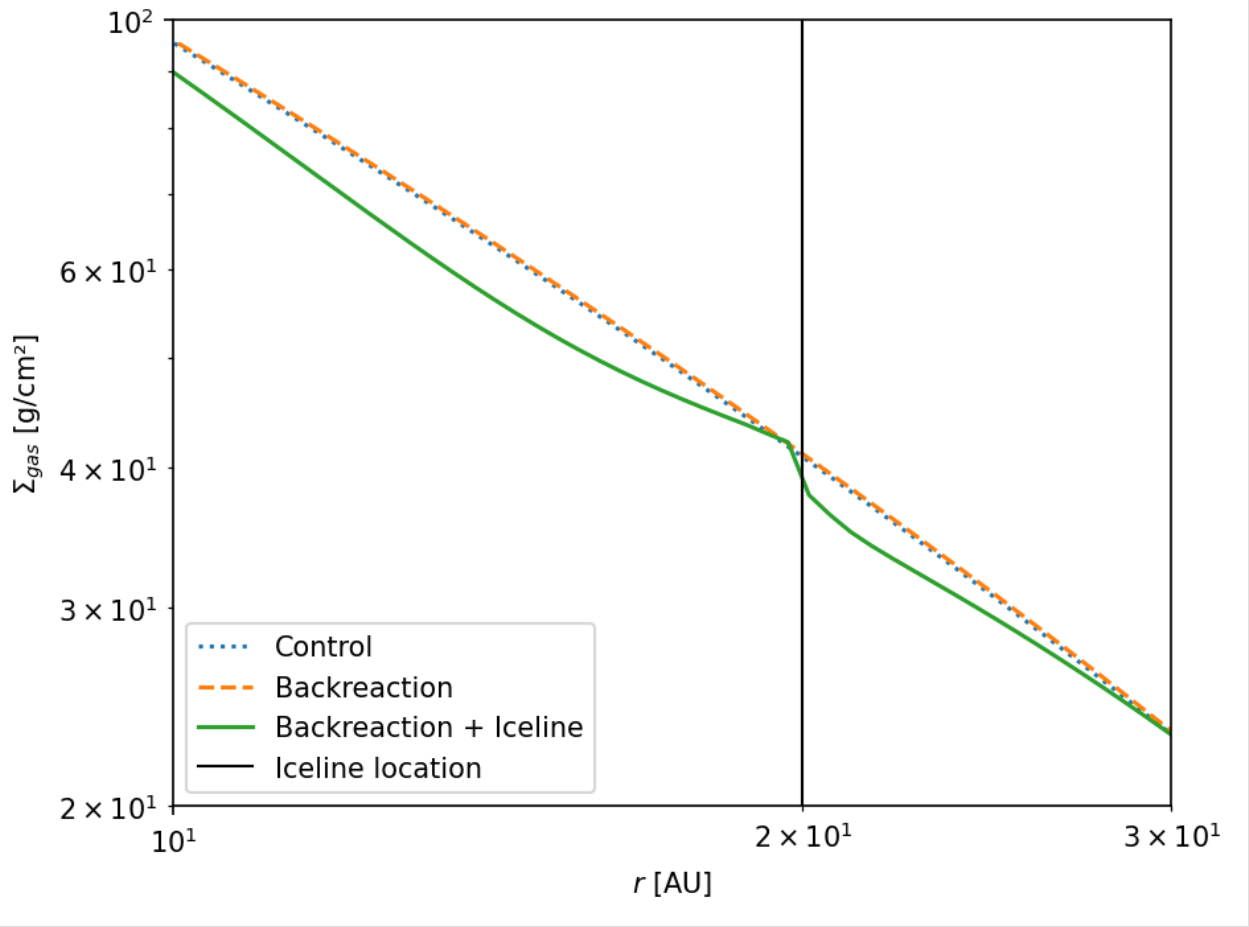
[17]:
```
snapshot = 10
reader.datadir = sim_control.writer.datadir
sim_control = reader.read.output(snapshot)

reader.datadir = sim_backreaction.writer.datadir
sim_backreaction = reader.read.output(snapshot)

reader.datadir = sim_backreaction_iceline.writer.datadir
sim_backreaction_iceline = reader.read.output(snapshot)
```

The radial grid is the same, so we read it here

[18]:
```
r = sim_control.grid.r
```

We start by taking a look at the gas and dust surface densities of the three simulations:

[19]:
```
fig, ax = plt.subplots()
ax.loglog(r/c.au, sim_control.gas.Sigma, ls=':', label = 'Control')
ax.loglog(r/c.au, sim_backreaction.gas.Sigma, ls='--', label = 'Backreaction')
ax.loglog(r/c.au, sim_backreaction_iceline.gas.Sigma, ls='-', label = 'Backreaction +␣
↪Iceline')
ax.axvline(r_iceline/c.au, lw=1, c="black", label="Iceline location")
ax.set_xlim([10,30])
ax.set_ylim([20,100])
ax.set_xlabel('$r$ [AU]')
ax.set_ylabel(r'$\Sigma_{gas}$ [g/cm$^2$]')
ax.legend()
fig.tight_layout()
```



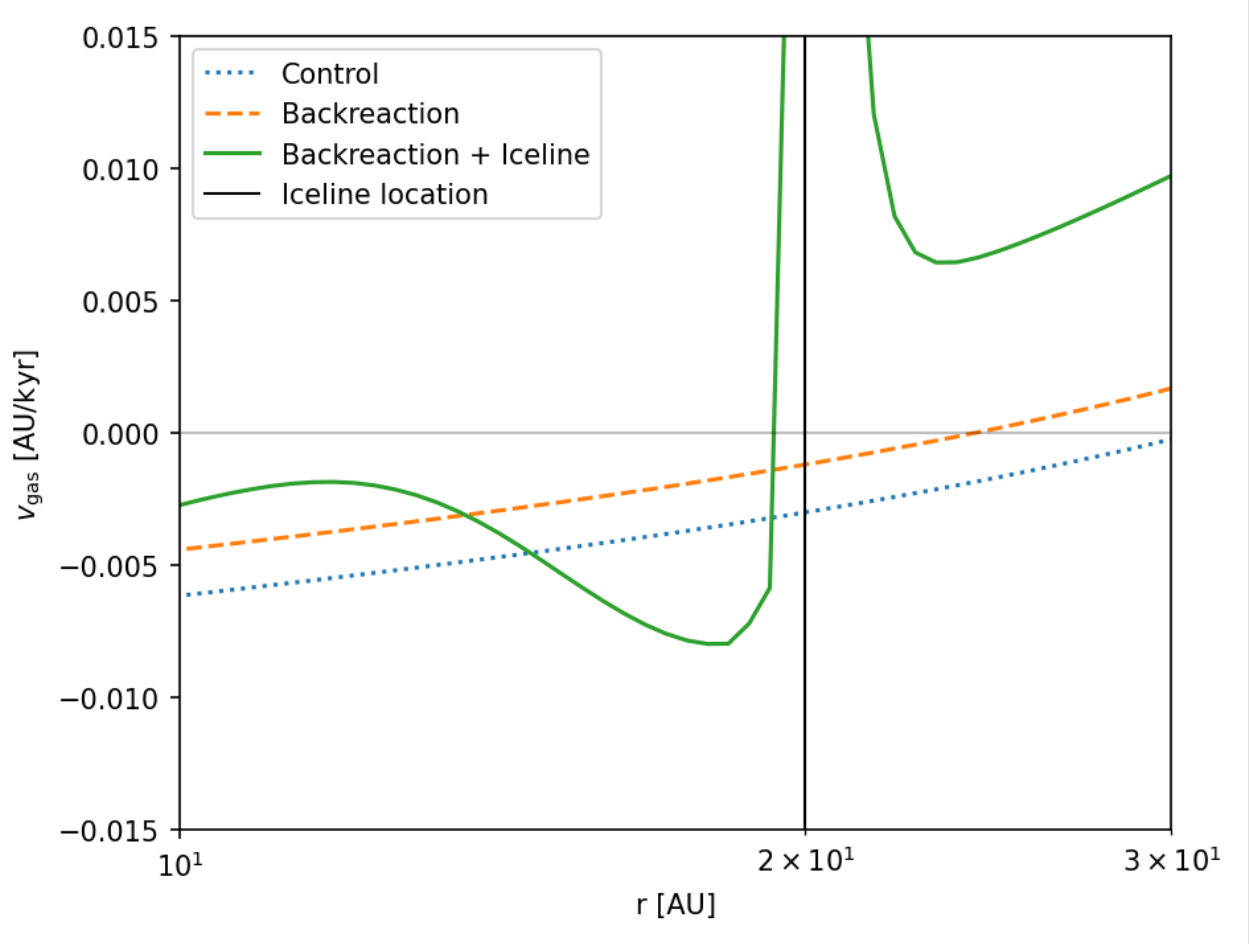
From the surface density profiles we see just about no difference between the `Control` and `Backreaction` simulations.

However we see that the gas surface density has changed when the iceline was considered in a high dust-to-gas ratio disk, creating a small bump in the disk.

Let's take a look at the gas velocities, which are now a contribution of two terms: a damped viscous accretion, and a backreaction push against the pressure gradient.

$v_{r,\mathrm{gas}} = Av_\nu + 2Bv_\eta$

[20]:
```
fig, ax = plt.subplots()
ax.semilogx(r/c.au, sim_control.gas.v.rad/(c.au/(1.e3*c.year)), ls=':', label = 'Control
↪')
ax.semilogx(r/c.au, sim_backreaction.gas.v.rad/(c.au/(1.e3*c.year)), ls='--', label =
↪'Backreaction')
ax.semilogx(r/c.au, sim_backreaction_iceline.gas.v.rad/(c.au/(1.e3*c.year)), ls='-',␣
↪label = 'Backreaction + Iceline')
ax.axvline(r_iceline/c.au, lw=1, c="black", label="Iceline location")
ax.axhline(0., lw=1, ls="-", c="black", alpha=0.25)
ax.set_xlim([10,30])
ax.set_ylim([-1.5e-2,1.5e-2])
ax.set_xlabel('r [AU]')
ax.set_ylabel(r'$v_\mathrm{gas}$ [AU/kyr]')
ax.legend()
fig.tight_layout()
```



Now the difference in the radial velocity of the gas is clear.

In the simulation with the iceline, the backreaction effect is strong enough to revert the gas flow at the snowline location, which leads to the creation of the small bump in the disk.

As a rule of thumb, the backreaction pushing term becomes dominant over the viscous evolution when:

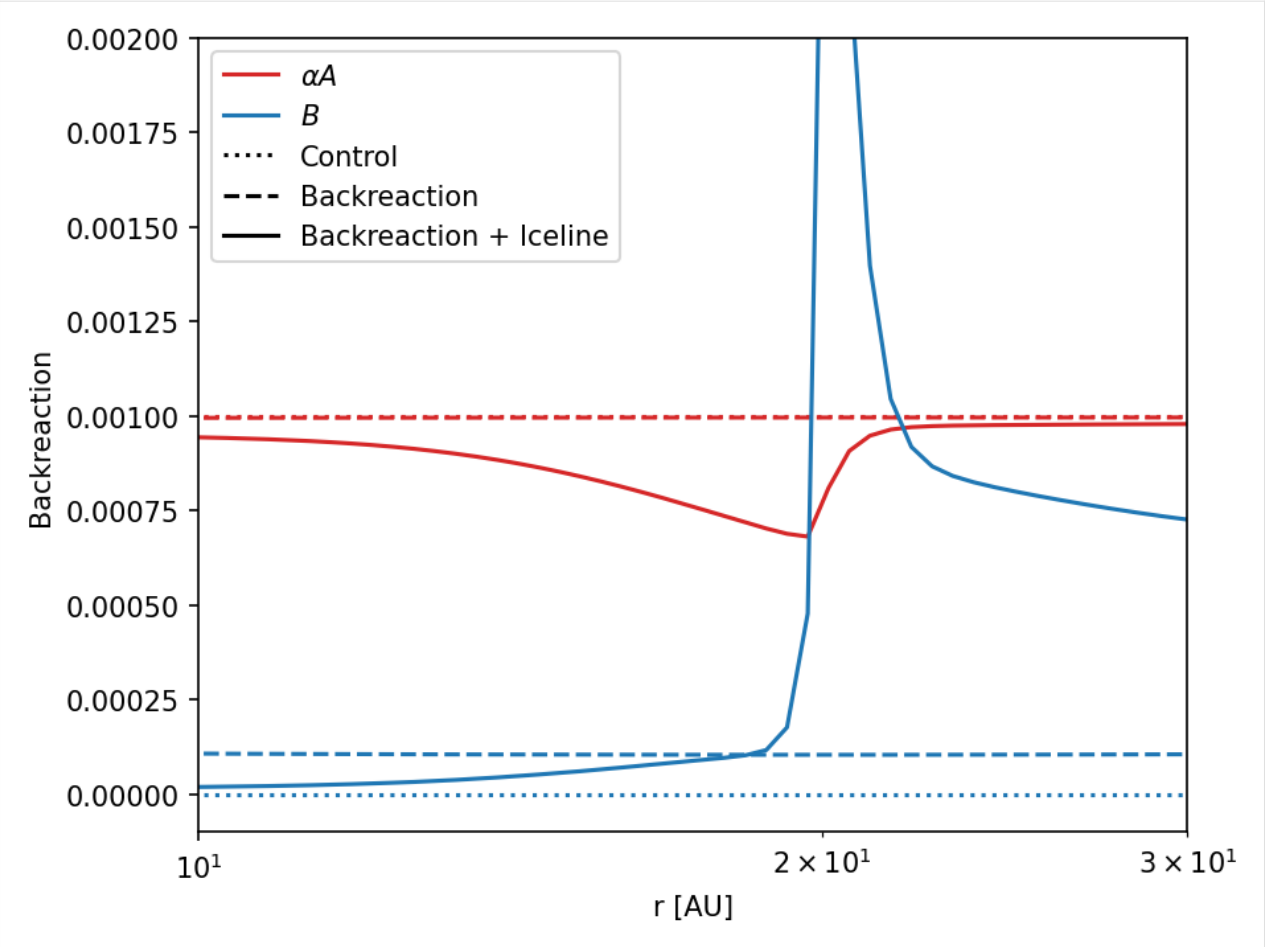
$\alpha A \lesssim B$

which is approximately equivalent to:

$1 \lesssim \mathrm{St}\,\epsilon\,\alpha^{-1}$

[21]:
```
A_control = sim_control.dust.backreaction.A
A_backreaction = sim_backreaction.dust.backreaction.A
A_iceline = sim_backreaction_iceline.dust.backreaction.A
B_control = sim_control.dust.backreaction.B
B_backreaction = sim_backreaction.dust.backreaction.B
B_iceline = sim_backreaction_iceline.dust.backreaction.B
alpha = sim_control.gas.alpha
```

[22]:
```
fig, ax = plt.subplots()

ax.semilogx(r/c.au, A_control * alpha, c="C3", ls=':')
ax.semilogx(r/c.au, A_backreaction * alpha, c="C3", ls='--')
ax.semilogx(r/c.au, A_iceline * alpha, c="C3", ls='-')
ax.semilogx(r/c.au, B_control, c="C0", ls=':')
ax.semilogx(r/c.au, B_backreaction, c="C0", ls='--')
ax.semilogx(r/c.au, B_iceline, c="C0", ls='-')

ax.semilogx(r/c.au, -np.ones_like(r), c="C3", label = r'$\alpha A$')
ax.semilogx(r/c.au, -np.ones_like(r), c="C0", label = '$B$')
ax.semilogx(r/c.au, -np.ones_like(r), c="black", ls=':', label = 'Control')
ax.semilogx(r/c.au, -np.ones_like(r), c="black", ls='--', label = 'Backreaction')
ax.semilogx(r/c.au, -np.ones_like(r), c="black", ls='-' , label = 'Backreaction + Iceline
↪')
ax.legend()

ax.set_xlim([10,30])
ax.set_ylim([-1.e-4,2.e-3])
ax.set_xlabel('r [AU]')
ax.set_ylabel(r'Backreaction')

fig.tight_layout()
```



This plot show that in the `Backreaction + Iceline` simulation the backreaction coefficient B, that pushes the gas against the pressure gradient, dominates over the viscous spreading measured by αA (i.e. viscous spreading with backreaction damping)

In contrast, the `Backreaction` simulation without the iceline and a normal dust-to-gas ratio only shows a mild pushing coefficient, with $B \ll \alpha A$, which indicates that the evolution is mostly viscous.

We note that the backreaction effects only last while the dust continues to flow across the disk, and weakens if the dust-to-gas ratio decreases.

1.2 Grid

The `dustpylib.grid` submodule contains methods modify the grids.

1.2.1 Grid refinement

The purpose of this package is to provide simple functions to refine the radial grid around certain locations.

In this example we want to refine the grid at a distance of $10\,\mathrm{au}$. We first create a `DustPy` simulation object.

[1]:
```
from dustpy import Simulation
from dustpy import constants as c
```

[2]:
```
r0 = 10. * c.au
```

[3]:
```
s = Simulation()
```

Since we want to provide a custom grid, we do not initialize the simulation object at this point. Instead we create our own grid cell interfaces, which are at this points identical to the default interfaces.

[4]:
```
import numpy as np
```

[5]:
```
ri = np.geomspace(s.ini.grid.rmin, s.ini.grid.rmax, s.ini.grid.Nr)
```

We can now use the helper function `refine_radial_local()` to locally refine our grid. We do this for different refinement levels to demonstrate the differences.

[6]:
```
from dustpylib.grid.refinement import refine_radial_local
```

[7]:
```
ri_fine_1 = refine_radial_local(ri, r0, num=1)
ri_fine_3 = refine_radial_local(ri, r0, num=3)
ri_fine_5 = refine_radial_local(ri, r0, num=5)
```

We can plot the cummulative distribution of grid cell interfaces.

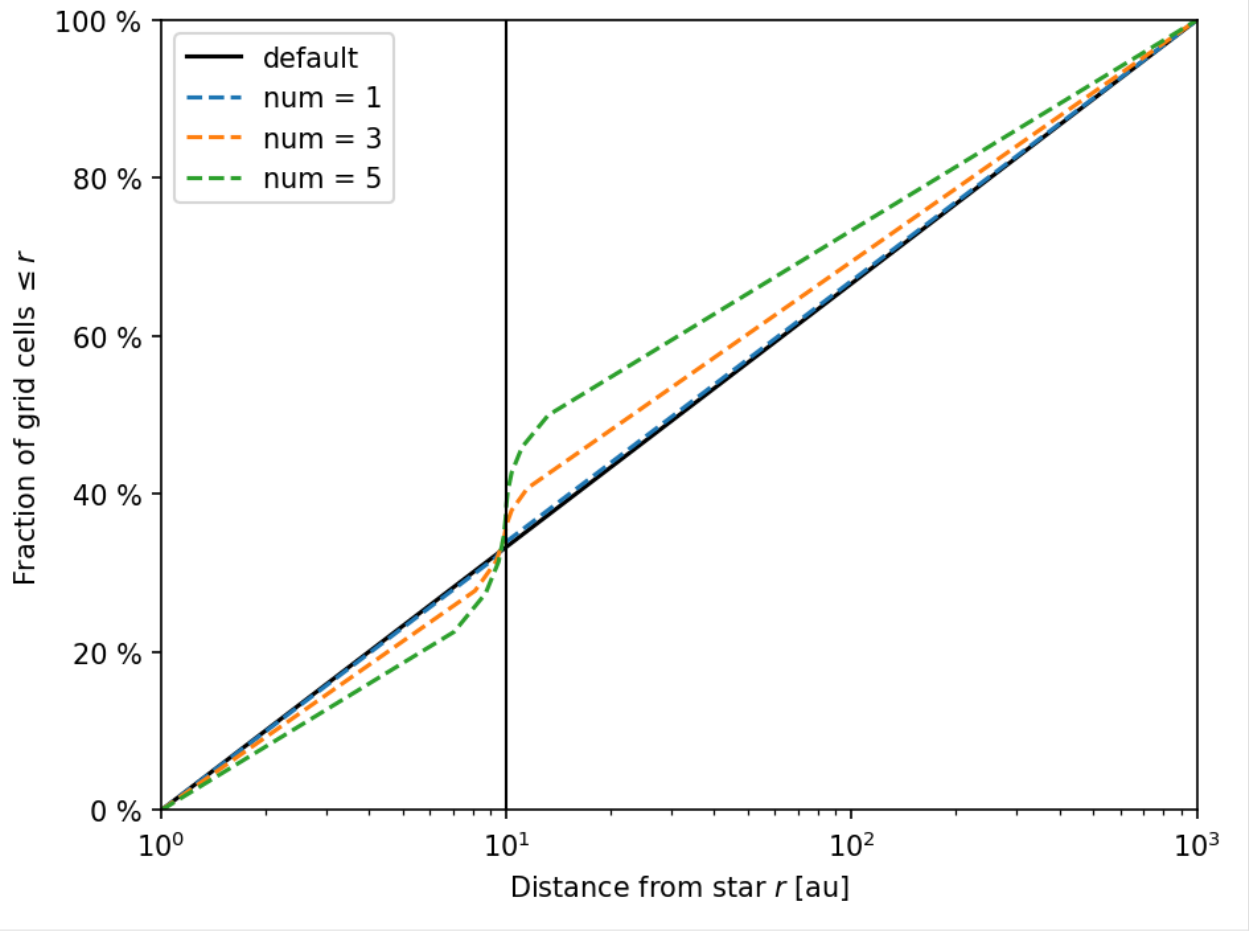
[8]:
```
import matplotlib.pyplot as plt
plt.rcParams["figure.dpi"] = 150.
```

[9]:
```
fig, ax = plt.subplots()
ax.semilogx(ri/c.au, np.linspace(0., 100., ri.shape[0]), label="default", color="black")
ax.semilogx(ri_fine_1/c.au, np.linspace(0., 100., ri_fine_1.shape[0]), "--", label="num␣
↪= 1")
ax.semilogx(ri_fine_3/c.au, np.linspace(0., 100., ri_fine_3.shape[0]), "--", label="num␣
↪= 3")
ax.semilogx(ri_fine_5/c.au, np.linspace(0., 100., ri_fine_5.shape[0]), "--", label="num␣
↪= 5")
ax.axvline(r0/c.au, lw=1, color="black")
ax.set_xlim(ri[0]/c.au, ri[-1]/c.au)
ax.set_ylim(0., 100.)
ax.set_yticks(ax.get_yticks())
ax.set_yticklabels(["{:.0f} %".format(t) for t in ax.get_yticks()])
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Distance from star $r$ [au]")
ax.set_ylabel("Fraction of grid cells $\leq r$")
ax.legend()
fig.tight_layout()
```



To initialize the model with the new refined grid we have to set the radial grid cell interfaces before initialization.

[10]:
```
s.grid.ri = ri_fine_3
```

[11]:
```
s.initialize()
```

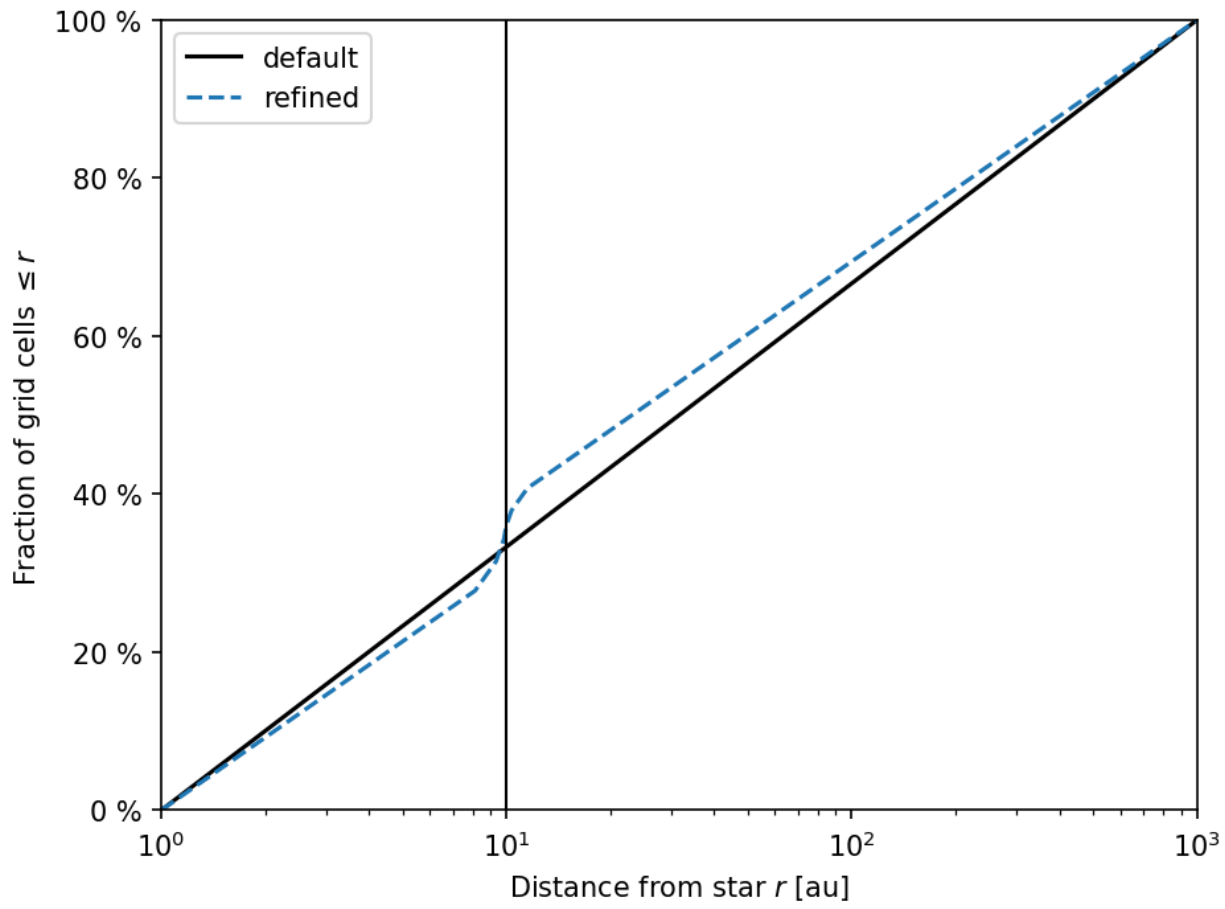
The model can now be run with the new grid with `su.run()`.

[12]:
```
fig, ax = plt.subplots()
ax.semilogx(ri/c.au, np.linspace(0., 100., ri.shape[0]), label="default", color="black")
ax.semilogx(s.grid.ri/c.au, np.linspace(0., 100., s.grid.ri.shape[0]), "--", label=
↪"refined")
ax.axvline(r0/c.au, lw=1, color="black")
ax.set_xlim(ri[0]/c.au, ri[-1]/c.au)
ax.set_ylim(0., 100.)
ax.set_yticks(ax.get_yticks())
ax.set_yticklabels(["{:.0f} %".format(t) for t in ax.get_yticks()])
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Distance from star $r$ [au]")
ax.set_ylabel("Fraction of grid cells $\leq r$")
ax.legend()
fig.tight_layout()
```



1.3 Planetesimals

The `dustpylib.planetesimals` submodule contains methods to implement planetesimal formation into `DustPy`.

1.3.1 Planetesimal formation

`dustpylib.planetesimals.formation` contains several prescriptions for planetesimal formation. They are briefly described here. For more detailed discussions, please have a look at their respective publications, which are referenced here.

Drążkowska et al. (2016)

In Drążkowska et al. (2016) only pebbles contribute to planetesimal formation. Pebbles are defined as particles with sizes above a certain cristical Stokes number $\mathrm{St}_{\mathrm{crit}}$. In the default case $\mathrm{St}_{\mathrm{crit}} = 0.01$.

$\Sigma_{\mathrm{peb}} = \Sigma_{\mathrm{dust}} \left(\mathrm{St} \geq \mathrm{St}_{\mathrm{crit}}\right)$

If the midplane pebbles-to-gas ratio is above a critical threshold, planetesimal formation is triggered. In the default case, this critical ratio is 1.
If planetesimal formation is triggered, a fraction ζ of the pebbles is converted into planetesimals per orbit, where $\zeta = 0.01$ in the default case.

Implementing this into `DustPy` would work as follows. First we define the hyperparameters of the method. These are the critical pebbles-to-gas ratio, the critical Stokes number and ζ.

[1]:
```
p2g_crit = 1.
St_crit = 0.01
zeta = 0.01
```

Then we create a simulation object and initialize it.

[2]:
```
from dustpy import Simulation
```

[3]:
```
s = Simulation()
```

[4]:
```
s.initialize()
```

In the next step we add an updater to the external sources of the dust surface density, that removes dust if planetesimal formation is triggered with the the method above. Here we can import the `drazkowska2016()` function, which returns the source term of dust due to planetesimal formation. If will be negative if planetesimal formation is triggered.

[5]:
```
from dustpylib.planetesimals.formation import drazkowska2016
```

[6]:
```
def S_ext(s):
    return drazkowska2016(
        s.grid.OmegaK,
        s.dust.rho,
        s.gas.rho,
        s.dust.Sigma,
        s.dust.St,
        p2g_crit=p2g_crit,
        St_crit=St_crit,
        zeta=zeta
    )
```

[7]:
```
s.dust.S.ext.updater = S_ext
```

Then we are going to add a group for the planetesimals and a field to store their surface density.

[8]:
```
import numpy as np
```

[9]:
```
s.addgroup("planetesimals", description="Planetesimal quantities")
s.planetesimals.addfield("Sigma", np.zeros_like(s.gas.Sigma), description="Surface␣
↪density of planetesimals [g/cm$^2$]")
```

We are going to let DustPy integrate this field over time to get the evolution of the planetesimal surface density. Therefore we need to define a derivative of the planetesimal surface density. The derivative is simply the negative sum over the external source terms defined above.

[10]:
```
def dSigma_planetesimals(s, t, Sigma_planetesimals):
    return -s.dust.S.ext.sum(-1)
```

This function is added to the differentiator of the field.

[11]:
```
s.planetesimals.Sigma.differentiator = dSigma_planetesimals
```

In the next step we have to create an integration instruction of this field. We are going to integrate the planetesimal surface density with a simple explicit 1st-order Euler scheme.

[12]:
```
from simframe import Instruction
from simframe import schemes
```

[13]:
```
instruction = Instruction(schemes.expl_1_euler, s.planetesimals.Sigma, description=
↪"Planetesimals: explicit 1st-order Euler")
```

This instruction is added to the existing integration instructions.

[14]:
```
s.integrator.instructions.append(instruction)
```

[15]:
```
s.integrator.instructions
```

[15]:
```
[Instruction (Dust: implicit 1st-order direct solver),
 Instruction (Gas: implicit 1st-order direct solver),
 Instruction (Planetesimals: explicit 1st-order Euler)]
```

We can now update the simulation object.

[16]:
```
s.update()
```

The simulation is now ready to go and can be started with `s.run()`.
Note, that this setup will not create any planetesimals, since the conditions for planetesimal formation will never be fulfilled without any mechanism to concentrate dust above the given threshold.

Schoonenberg et al. (2018)

The prescription of Schoonenberg et al. (2018) is very similar to Drążkowska et al. (2016). But instead of only considering pebbles, all particles contribute to planetesimal formation.

If the midplanet dust-to-gas ratio is above a critical threshold, planetesimal formation is triggered. The critical threshold is 1 in the default case. As soon as planetesimal formation is triggered, a fraction of ζ of the dust particles is converted to planetesimal per settling time scale

$t_{\mathrm{sett}} = \frac{1}{\mathrm{St}\Omega_{\mathrm{K}}}$.

Since the settling time scale is inversly proportional to the Stokes number of the particles, the contribution of small particles to planetesimal formation is supressed. In the default case the planetesimal formation efficiency is $\zeta = 0.1$

Implementation into `DustPy` is identical to the example above. First, we define the hyperparameters of the method.

[17]:
```
d2g_crit = 1.
zeta = 0.1
```

Then we create a simulation object and initialize it.

[18]:
```
from dustpy import Simulation
```

[19]:
```
s = Simulation()
```

[20]:
```
s.initialize()
```

In the next step we add an updater to the external sources of the dust surface density, that removes dust if planetesimal formation is triggered with the the method above. Here we can import the `schoonenberg2018()` function, which returns the source term of dust due to planetesimal formation. If will be negative if planetesimal formation is triggered.

[21]:
```
from dustpylib.planetesimals.formation import schoonenberg2018
```

[22]:
```
def S_ext(s):
    return schoonenberg2018(
        s.grid.OmegaK,
        s.dust.rho,
        s.gas.rho,
        s.dust.Sigma,
        s.dust.St,
        d2g_crit=d2g_crit,
        zeta=zeta
    )
```

[23]:
```
s.dust.S.ext.updater = S_ext
```

Then we are going to add a group for the planetesimals and a field to store their surface density.

[24]:
```
import numpy as np
```

[25]:
```
s.addgroup("planetesimals", description="Planetesimal quantities")
s.planetesimals.addfield("Sigma", np.zeros_like(s.gas.Sigma), description="Surface␣
↪density of planetesimals [g/cm$^2$]")
```

We are going to let `DustPy` integrate this field over time to get the evolution of the planetesimal surface density. Therefore we need to define a derivative of the planetesimal surface density. The derivative is simply the negative sum over the external source terms defined above.

[26]:
```
def dSigma_planetesimals(s, t, Sigma_planetesimals):
    return -s.dust.S.ext.sum(-1)
```

This function is added to the differentiator of the field.

[27]:
```
s.planetesimals.Sigma.differentiator = dSigma_planetesimals
```

In the next step we have to create an integration instruction of this field. We are going to integrate the planetesimal surface density with a simple explicit 1st-order Euler scheme.

[28]:
```
from simframe import Instruction
from simframe import schemes
```

[29]:
```
instruction = Instruction(schemes.expl_1_euler, s.planetesimals.Sigma, description=
↪"Planetesimals: explicit 1st-order Euler")
```

This instruction is added to the existing integration instructions.

[30]:
```
s.integrator.instructions.append(instruction)
```

[31]:
```
s.integrator.instructions
```

[31]:
```
[Instruction (Dust: implicit 1st-order direct solver),
 Instruction (Gas: implicit 1st-order direct solver),
 Instruction (Planetesimals: explicit 1st-order Euler)]
```

We can now update the simulation object.

[32]:
```
s.update()
```

The simulation is now ready to go and can be started with `s.run()`.
Note, that as above this setup will not create any planetesimals, since the conditions for planetesimal formation will never be fulfilled without any mechanism to concentrate dust above the given threshold.

Miller et al. (2021)

The presciption of Miller et al. (2021) is very similar to Schoonenberg et al. (2018). But instead of a hard threshold in midplane dust-to-gas ratio it employs a smoooth transition. The probability that planetesimal formation is triggered is given by

$$\mathcal{P} = \frac{1}{2}\left[1 + \tanh\left(\frac{\log(\varepsilon) - \log(\varepsilon_{\mathrm{crit}})}{n}\right)\right],$$

where ε and $\varepsilon_{\mathrm{crit}}$ are the midplane dust-to-gas ratio and its critical value at which this transition occurs (the default is $\varepsilon_{\mathrm{crit}} = 1$) and n is a smoothness parameters (the default is $n = 0.03$).

Setting up this prescription in `DustPy` works identical to the examples above. First, we define the hyperparameters of the method.

[33]:
```
d2g_crit = 1.
n = 0.03
zeta = 0.1
```

Then we create a simulation object and initialize it.

[34]:
```
from dustpy import Simulation
```

[35]:
```
s = Simulation()
```

[36]:
```
s.initialize()
```

In the next step we add an updater to the external sources of the dust surface density, that removes dust if planetesimal formation is triggered with the the method above. Here we can import the `miller2021()` function, which returns the source term of dust due to planetesimal formation. If will be negative if planetesimal formation is triggered.

[37]:
```
from dustpylib.planetesimals.formation import miller2021
```

[38]:
```
def S_ext(s):
    return miller2021(
        s.grid.OmegaK,
        s.dust.rho,
        s.gas.rho,
        s.dust.Sigma,
        s.dust.St,
        d2g_crit=d2g_crit,
        n=n,
        zeta=zeta
    )
```

[39]:
```
s.dust.S.ext.updater = S_ext
```

Then we are going to add a group for the planetesimals and a field to store their surface density.

[40]:
```
import numpy as np
```

[41]:
```
s.addgroup("planetesimals", description="Planetesimal quantities")
s.planetesimals.addfield("Sigma", np.zeros_like(s.gas.Sigma), description="Surface␣
↪density of planetesimals [g/cm$^2$]")
```

We are going to let `DustPy` integrate this field over time to get the evolution of the planetesimal surface density. Therefore we need to define a derivative of the planetesimal surface density. The derivative is simply the negative sum over the external source terms defined above.

[42]:
```
def dSigma_planetesimals(s, t, Sigma_planetesimals):
    return -s.dust.S.ext.sum(-1)
```

This function is added to the differentiator of the field.

[43]:
```
s.planetesimals.Sigma.differentiator = dSigma_planetesimals
```

In the next step we have to create an integration instruction of this field. We are going to integrate the planetesimal surface density with a simple explicit 1st-order Euler scheme.

[44]:
```
from simframe import Instruction
from simframe import schemes
```

[45]:
```
instruction = Instruction(schemes.expl_1_euler, s.planetesimals.Sigma, description=
↪"Planetesimals: explicit 1st-order Euler")
```

This instruction is added to the existing integration instructions.

[46]:
```
s.integrator.instructions.append(instruction)
```

[47]:
```
s.integrator.instructions
```

[47]:
```
[Instruction (Dust: implicit 1st-order direct solver),
 Instruction (Gas: implicit 1st-order direct solver),
 Instruction (Planetesimals: explicit 1st-order Euler)]
```

We can now update the simulation object.

[48]:
```
s.update()
```

The simulation is now ready to go and can be started with `s.run()`.
Note, that as above this setup will not create any planetesimals, since the conditions for planetesimal formation will never be fulfilled without any mechanism to concentrate dust above the given threshold.

1.4 Radiative Transfer

The `dustpylib.radtrans` submodule contains interfaces to radiative transfer codes. The scope is to create input files from *DustPy* models that can be executed with the listed codes and optionally to provide methods to analyze the radiative transfer calculations.

1.4.1 RADMC-3D

RADMC-3D is a Monte Carlo Radiative Transfer software package for astrophysics. The purpose of `dustpylib.radtrans.radmc3d` is to produce `RADMC-3D` input files from `DustPy` simulations.
For details on the usage of `RADMC-3D` please have a look at the RADMC-3D documentation.

This module creates axisymmetric disk models where the dust particles are vertically distributed according their scale heights. Dust opacities are created using the dsharp_opac package (Birnstiel et al., 2018).
The intent of this module is not to provide a fully customizable interface to `RADMC-3D`, but rather to provide a minimum working setup, that can be further customized manually.

In this notebook we are going to produce `RADMC-3D` images from the planetary gaps example in the `DustPy` documentation. This repository contains a reduced `DustPy` output file of the final snapshot of the example model only containing the fields that are required to produce the `RADMC-3D` model. The example contains the default `DustPy` setup with Jupiter and Saturn added at their current locations.

This notebook only demonstrates the usage of the `DustPyLib` module. `RADMC-3D` has to be installed and executed manually.

All quantities are in CGS units.

In the first step we need to load the `DustPy` data file using the `hdf5writer` for this.

[1]:
```
from dustpy import hdf5writer
```

[2]:
```
writer = hdf5writer()
writer.datadir = "example_planetary_gaps"
```

[3]:
```
data = writer.read.output(21)
```

Creating the RADMC-3D model

We can now use `dustpylib.radtrans.radmc3d` to produce the model intput files. The `Model` class accepts either a `DustPy.Simulation` object directly or a namespace as returned by the `Writer.read.output()` method.

[4]:
```
from dustpylib.radtrans import radmc3d
```

[5]:
```
rt = radmc3d.Model(data)
```

The `Model` object contains several attributes, that can be modified, where by definition attibutes with trailing _ are imported from the `DustPy` model, while the ones without are used in the `RADMC-3D` model, e.g. the grid.

By default the outermost grid cell of the `DustPy` simulation will be ignored. By default the outer boundary condition of the gas is the floor value. This leads to very large Stokes numbers of the particles and therefore to very small dust scale heights and very large midplane dust densities. This can lead to unwanted features when the data is interpolated onto the `RADMC3D` grid. To use the last radial grid cell nevertheless, set the `ignore_last` keyword argument to `False`, for example `rt = radmc3d.Model(data, ignore_last=False)`.

The `RADMC-3D` model parameters are chosen reasonably, but can be customized if needed.
Some scattering modes of `RADMC-3D` require both hemispheres (θ-grid) of the disk and an azimuthal grid (φ-grid), while other modes can work with a single hemisphere or a single azimuthal cell. The module will by default create both hemispheres (θ-grid from 0 to π with 256 grid cells) and a coarse azimuthal grid (φ-grid from 0 to 2π with 16 grid cells).

In this notebook we are going to use scattering mode 5, which requires both hemispheres but does not require an azimuthal grid. We can therefore turn off the azimuthal grid by only providing two grid cell interfaces. You can customize all grids by setting the grid cell interfaces. Do not set the grid cell centers directly. This will happen automatically.

[6]:
```
import numpy as np
```

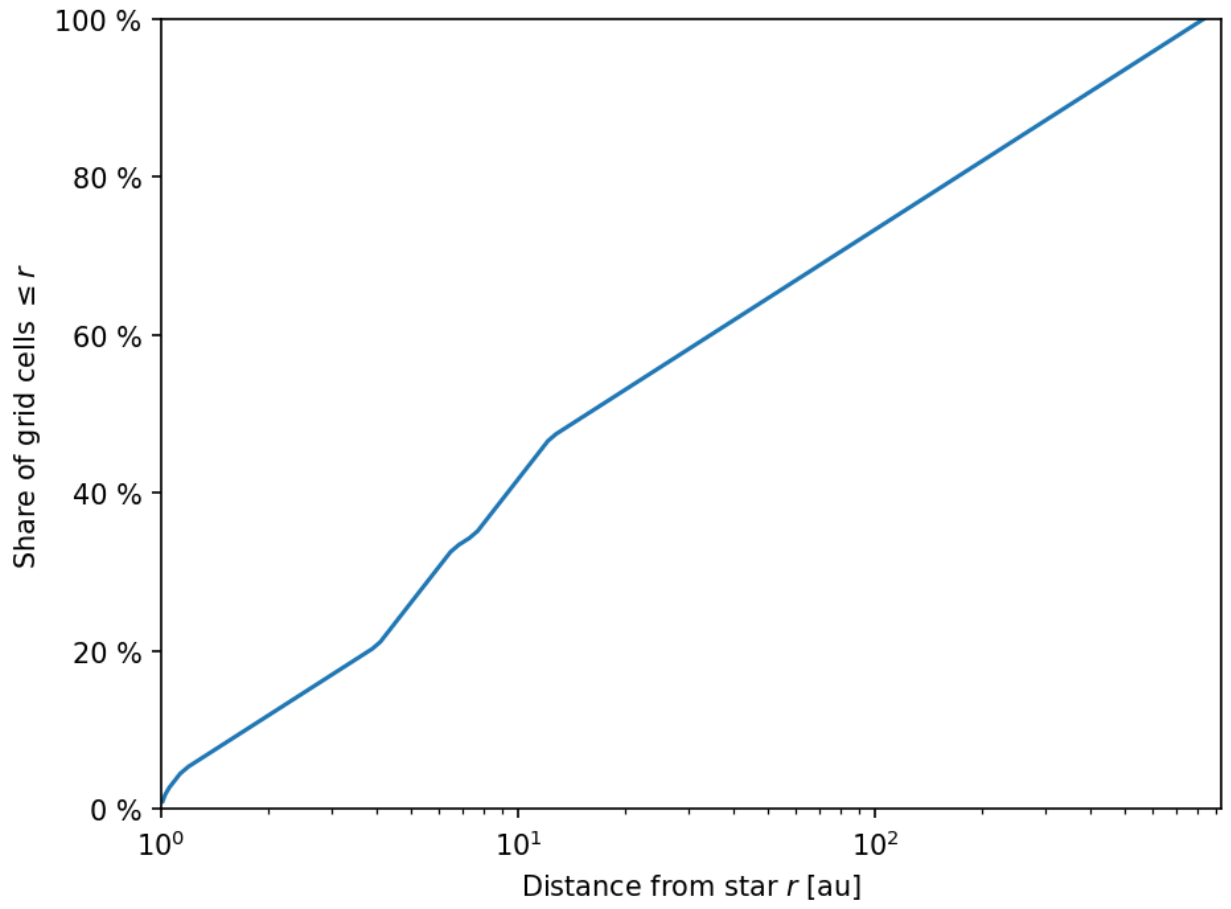
[7]:
```
rt.phii_grid = np.array([0., 2.*np.pi])
```

The inner edge of the disk is directly illuminated by the star and typically the densest part of the disk and therefore optically thick. Because of this the radial grid of the `RADMC-3D` setup is slightly refined in the inner part.
Note, that in this specific example the `DustPy` grid has been refined at the location of the planets, which will then be also the case in the default `RADMC-3D` model setup. A steeper slope means a finer radial grid.

[8]:
```
import dustpy.constants as c
import matplotlib.pyplot as plt
```

[9]:
```
plt.rcParams["figure.dpi"] = 150.
```

[10]:
```
fig, ax = plt.subplots()
Ncells = rt.rc_grid.shape[0]
ax.semilogx(rt.rc_grid/c.au, np.linspace(1., 101., Ncells))
ax.set_xlabel("Distance from star $r$ [au]")
ax.set_ylabel("Share of grid cells $\leq r$")
ax.set_xlim(rt.ri_grid[0]/c.au, rt.ri_grid[-1]/c.au)
ax.set_ylim(0., 100.)
ax.set_yticks(ax.get_yticks())
ax.set_yticklabels(["{:.0f} %".format(t) for t in ax.get_yticks()])
fig.tight_layout()
```



DustPy typically uses more particle species than one would want to use in radiative transfer models for performance reasons. By default this module uses only 16 dust species between the smallest and largest sizes in the DustPy setup. However, as can be seen in the example notebook some of the larger bins are actually empty. It is therefore advisable to inspect the actual particle sizes and adjust the RADMC-3D size grid accordingly.

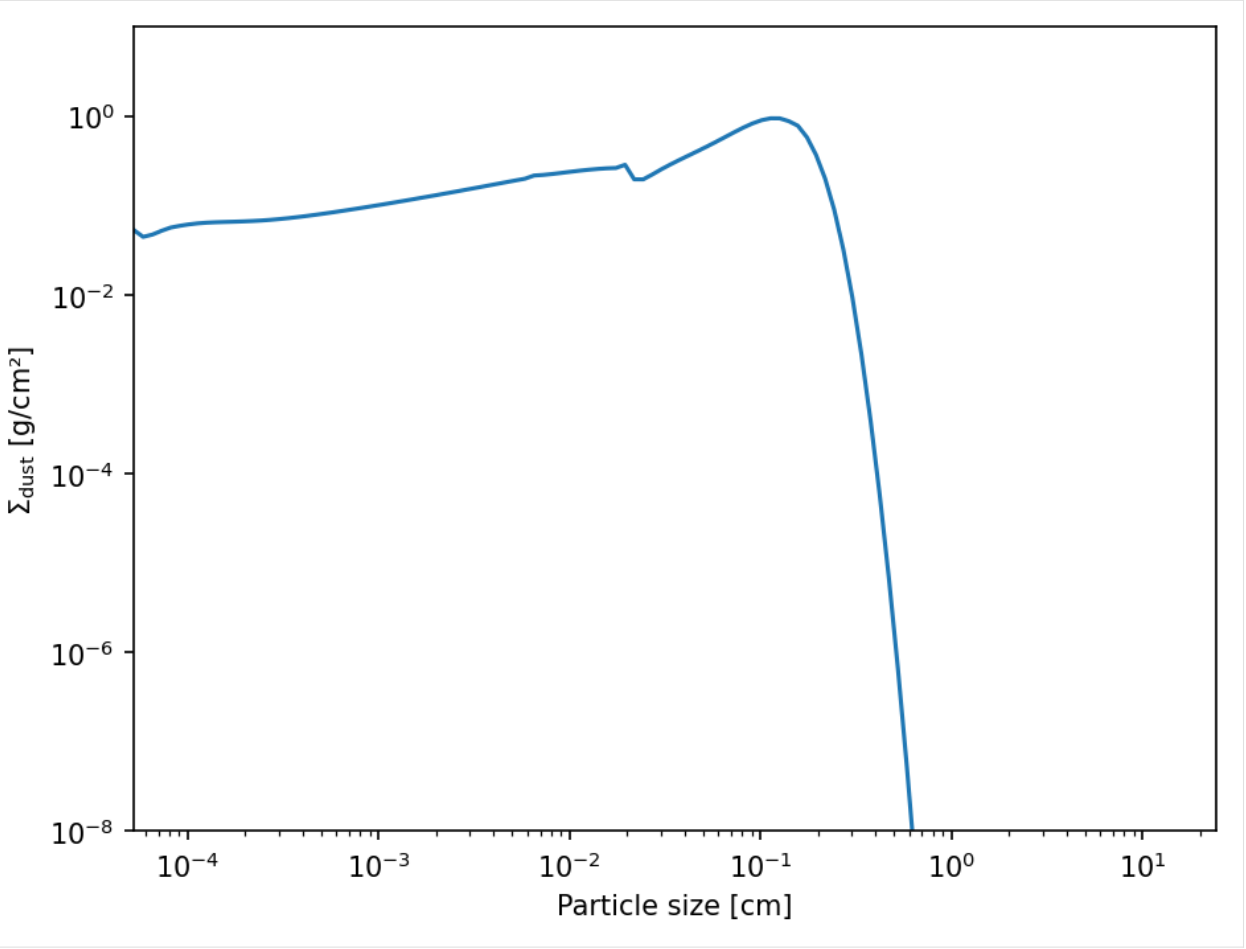
The largest particle are in the inner disk. We therefore have a look at the innermost DustPy grid cell.

[11]:
```
fig, ax = plt.subplots()
ax.loglog(rt.a_dust_[0, :], rt.Sigma_dust_[0, :])
ax.set_xlabel("Particle size [cm]")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel(r"$\Sigma_\mathrm{dust}$ [g/cm$^2$]")
ax.set_xlim(rt.a_dust_.min(), rt.a_dust_.max())
ax.set_ylim(1.e-8, 1.e1)
fig.tight_layout()
```



We therefore re-define the particle size grid of the `RADMC-3D` model with a maximum particle size of 1 cm. Note, that we have to set the interfaces of the size bins and there are 17 grid cell interfaces needed for 16 particle species. The hundreds of `DustPy` particle species will then be re-binned onto this grid while conserving mass.

[12]:
```
rt.ai_grid = np.geomspace(rt.a_dust_.min(), 1., 17)
```

The central particle size of a size bin is automatically calculated.

[13]:
```
rt.ac_grid
```

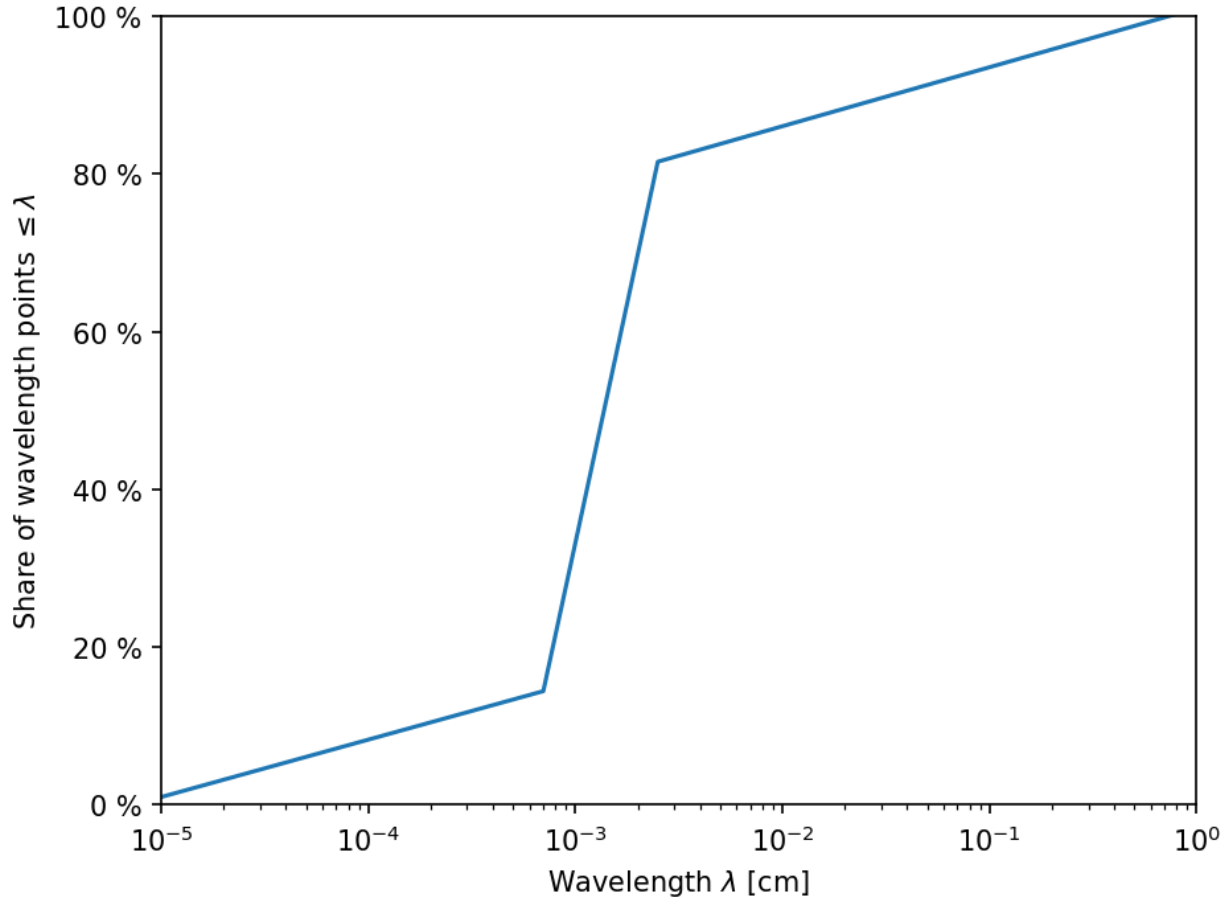
[13]:
```
array([7.45574916e-05, 1.38067479e-04, 2.55676905e-04, 4.73469062e-04,
       8.76782175e-04, 1.62364776e-03, 3.00671264e-03, 5.56790773e-03,
       1.03107946e-02, 1.90937944e-02, 3.53583790e-02, 6.54775548e-02,
       1.21253018e-01, 2.24539455e-01, 4.15807933e-01, 7.70003814e-01])
```

The default wavelength grid will cover wavelengths from $0.1\,\mu\mathrm{m}$ to $1\,\mathrm{cm}$ with a refinement around the $10\,\mu\mathrm{m}$ dust

feature. The wavelength grid needs to cover those wavelengths at which the star mostly radiates and those wavelengths at which the dust particles predominantly cool. For hot stars ($\gtrsim 10\,000$ K) the wavelengths grid has to be extended to smaller wavelengths. If the wavelengths are too small, however, computation of the opacities can fail, if the wavelengths are out of range of the optical constants used to calculate the opacities.

Note, that `RADMC-3D` uses microns for wavelengths. This module, however, uses exclusively CGS units and converts the quantities internally, if needed.

[14]:
```
fig, ax = plt.subplots()
Ncells = rt.lam_grid.shape[0]
ax.semilogx(rt.lam_grid, np.linspace(1., 101., Ncells))
ax.set_xlabel("Wavelength $\lambda$ [cm]")
ax.set_ylabel("Share of wavelength points $\leq \lambda$")
ax.set_xlim(rt.lam_grid[0], rt.lam_grid[-1])
ax.set_ylim(0., 100.)
ax.set_yticks(ax.get_yticks())
ax.set_yticklabels(["{:.0f} %".format(t) for t in ax.get_yticks()])
fig.tight_layout()
```



Additional options for the `radmc3d.inp` file can be added to the `radmc3d_options` dictionary.

[15]:
```
rt.radmc3d_options
```

[15]:
```
{'modified_random_walk': 1, 'iranfreqmode': 1, 'istar_sphere': 1}
```

We want to use $10\,000\,000$ photons for the thermal run, $1\,000\,000$ photons for the scattering run, and $10\,000$ for the

spectral run.

[16]:
```
rt.radmc3d_options["nphot"] = 10_000_000
rt.radmc3d_options["nphot_scat"] = 1_000_000
rt.radmc3d_options["nphot_spec"] = 10_000
```

The maximum optical depth of absorption at which photons get destroyed is by default 30. In this notebook we want to decrease this value to 5.

[17]:
```
rt.radmc3d_options["mc_scat_maxtauabs"] = 5.
```

`DustPyLib` will create opacity files that allow for the most complex scattering mode. In this notebook we want to use the largest possible mode. We therefore do not specify `scattering_mode_max`.

For the scattering mode we are using here, `RADMC-3D` internally creates a φ-grid. To speed up the computation we want to reduce the number of grid cells from the default of 360 to 60.

[18]:
```
rt.radmc3d_options["dust_2daniso_nphi"] = 60
```

We want to store the `RADMC-3D` input files in the following directory

[19]:
```
rt.datadir = "radmc3d"
```

To compute the dust opacities `DustPyLib` uses the dsharp_opac package. Two different opacity models can be used: The DSHARP opacities (`Model.opacity="birnstiel2018"`, Birnstiel et al., 2018, default) or the Ricci opacities (`Model.opacity="ricci2010"`, Ricci et al., 2010). Please cite all references that are printed during the computations.

The `Model.write_files()` has additional options. If you do not want to compute opacities, you can set `writeopacities=False`. If you want to use smoothed opacities, use `smooth_opacities=True`. In that case the opacities are not calculated for a singular particle size per bin, but for several sizes and then averaged. Note, that this will take significantly longer.

This module furthermore writes the file `metadata.npz`, which is not needed for the `RADMC-3D` model setup, but contains additional data that may be useful for postprocessing. In this case the particle sizes.

We can now create the `RADMC-3D` model files.

[20]:
```
rt.write_files()
```

```
Writing radmc3d/radmc3d.inp...done.
Writing radmc3d/stars.inp...done.
Writing radmc3d/wavelength_micron.inp...done.
Writing radmc3d/amr_grid.inp...done.
Writing radmc3d/dust_density.inp...done.
Writing radmc3d/dust_temperature.dat...done.
Writing radmc3d/dustopac.inp...done.

Computing opacities...
Using dsharp_opac. Please cite Birnstiel et al. (2018).
Using DSHARP mix. Please cite Birnstiel et al. (2018).
Please cite Warren & Brandt (2008) when using these optical constants
Please cite Draine 2003 when using these optical constants
Reading opacities from troilitek
Please cite Henning & Stognienko (1996) when using these optical constants
Reading opacities from organicsk
Please cite Henning & Stognienko (1996) when using these optical constants
```

(continues on next page)

(continued from previous page)

```
| material                              | volume fractions | mass fractions |
|---------------------------------------|------------------|----------------|
| Water Ice (Warren & Brandt 2008)      | 0.3642           | 0.2            |
| Astronomical Silicates (Draine 2003)| 0.167            | 0.329          |
| Troilite (Henning)                    | 0.02578          | 0.07434        |
| Organics (Henning)                    | 0.443            | 0.3966         |
Mie ... Done!
```

```
/home/stammler/anaconda3/envs/j/lib/python3.11/site-packages/dsharp_opac/dsharp_opac.py:
↪2778: RuntimeWarning: invalid value encountered in scalar divide
  g[grain, i] = -2 * np.pi * np.trapz(zscat[grain, i, :, 0] * mu, x=mu) / k_sca[grain, i]
```

```
Writing radmc3d/dustkapscatmat_00.inp...done.
Writing radmc3d/dustkapscatmat_01.inp...done.
Writing radmc3d/dustkapscatmat_02.inp...done.
Writing radmc3d/dustkapscatmat_03.inp...done.
Writing radmc3d/dustkapscatmat_04.inp...done.
Writing radmc3d/dustkapscatmat_05.inp...done.
Writing radmc3d/dustkapscatmat_06.inp...done.
Writing radmc3d/dustkapscatmat_07.inp...done.
Writing radmc3d/dustkapscatmat_08.inp...done.
Writing radmc3d/dustkapscatmat_09.inp...done.
Writing radmc3d/dustkapscatmat_10.inp...done.
Writing radmc3d/dustkapscatmat_11.inp...done.
Writing radmc3d/dustkapscatmat_12.inp...done.
Writing radmc3d/dustkapscatmat_13.inp...done.
Writing radmc3d/dustkapscatmat_14.inp...done.
Writing radmc3d/dustkapscatmat_15.inp...done.

Writing radmc3d/metadata.npz...done.
```

The RADMC-3D model is now ready to go. Of course, all files, especially `radmc3d.inp`, can be customized manually.

Inspecting the RADMC-3D model

`dustpylib.radtrans.radmc3d` has a simple method to load the RADMC-3D model files to inspect the setup before executing RADMC-3D. It does only work for models created with `DustPyLib` due to the assumed symmetries. For more general methods for analyzing models, use `radmc3dPy` that comes with RADMC-3D.

[21]:
```
model = radmc3d.read_model(datadir="radmc3d")
```

We can now plot the densities and temperatures of the RADMC-3D model.

[22]:
```
def plot_model(model, spec=-1):
    """
    Function plots the RADMC-3D model.

    Parameters
    ----------
    model : namespace
        The RADMC-3D model data
    spec : integer, optional, default : -1
```

(continues on next page)

(continued from previous page)

```
        Particle species to be plotted. If -1, the total
        dust densities are plotted.
    """
    width = 6.
    height = width/1.3

    if spec==-1:
        rho = np.maximum(np.hstack((model.rho[:, :1, 0, :].sum(-1), model.rho[:, :, 0, :
↪].sum(-1))), 1.e-100)
        T = np.hstack((model.T[:, :1, :, 0], model.T[:, :, :, 0])).mean(-1)
        a_rho = "total"
        a_T = "$a$ = {:.2e} cm".format(model.grid.a[0])
    else:
        rho = np.maximum(np.hstack((model.rho[:, :1, 0, spec], model.rho[:, :, 0,␣
↪spec])), 1.e-100)
        T = np.hstack((model.T[:, :1, :, spec], model.T[:, :, :, spec])).mean(-1)
        a_rho = "$a$ = {:.2e} cm".format(model.grid.a[spec])
        a_T = "$a$ = {:.2e} cm".format(model.grid.a[spec])

    rho = np.hstack((rho, np.flip(rho[:, 1:, ...], 1)))
    T = np.hstack((T, np.flip(T[:, 1:, ...], 1)))

    theta = np.hstack((model.grid.theta[0]-(model.grid.theta[1]-model.grid.theta[0]),␣
↪model.grid.theta))
    theta = np.hstack((theta, theta[1:]+np.pi))
    lev_rho_max = np.ceil(np.log10(rho.max()))
    levels_rho = np.arange(lev_rho_max-6, lev_rho_max+1, 1.)

    fig, ax = plt.subplots(ncols=2, figsize=(2*width, height), subplot_kw={"projection":
↪"polar"})

    p0 = ax[0].contourf(theta, model.grid.r/c.au, np.log10(rho), levels=levels_rho,␣
↪extend="both", cmap="viridis")
    ax[0].contourf(-theta, model.grid.r/c.au, np.log10(rho), levels=levels_rho, extend=
↪"both", cmap="viridis")
    ax[0].set_theta_zero_location("N")
    ax[0].set_rlim(0)
    ax[0].set_rscale("symlog")
    ax[0].set_rgrids([1., 10., 100.], angle=-45)
    ax[0].tick_params(axis='y', colors='white')
    ax[0].set_xticks([])
    ax[0].plot(0., 0., "*", color="C3", markeredgewidth=0, markersize=12)
    cbar0 = plt.colorbar(p0)
    cbar0.set_ticks(cbar0.get_ticks())
    cbar0.set_ticklabels(["$10^{{{:d}}}$".format(int(t)) for t in cbar0.get_ticks()])
    cbar0.set_label(r"$\rho_\mathrm{dust}$ [g/cm$^3$]")
    ax[0].set_title("Dust density, {}".format(a_rho))

    p1 = ax[1].contourf(theta, model.grid.r/c.au, T, extend="both", cmap="coolwarm")
    ax[1].contourf(-theta, model.grid.r/c.au, T, extend="both", cmap="coolwarm")
    ax[1].set_theta_zero_location("N")
    ax[1].set_rlim(0)
```

(continues on next page)

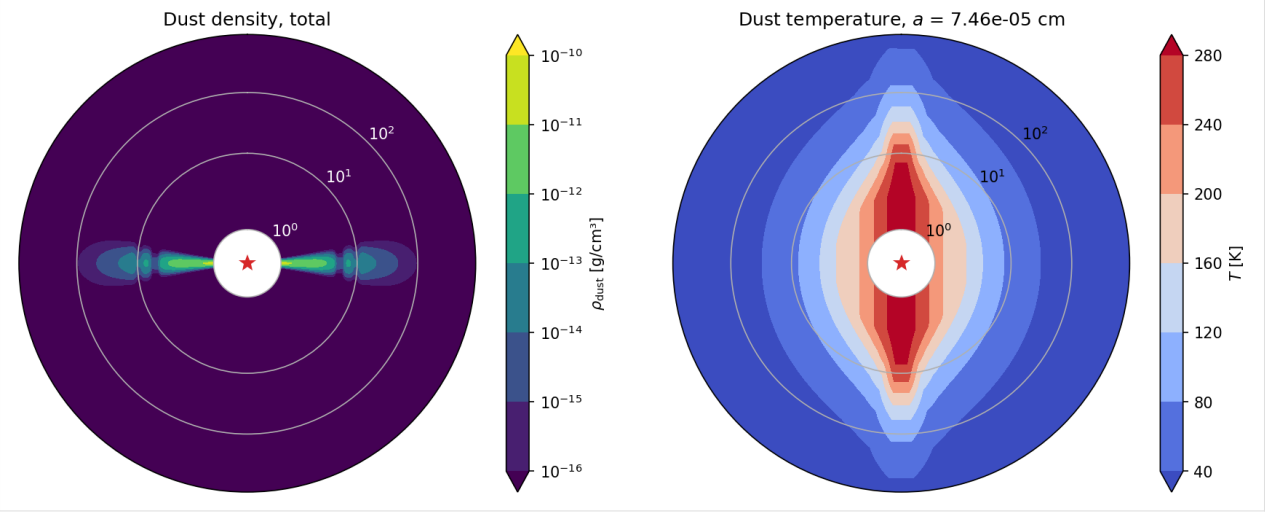
(continued from previous page)

```
    ax[1].set_rscale("symlog")
    ax[1].set_rgrids([1., 10., 100.], angle=-45)
    ax[1].tick_params(axis='y', colors='black')
    ax[1].set_xticks([])
    ax[1].plot(0., 0., "*", color="C3", markeredgewidth=0, markersize=12)
    cbar1 = plt.colorbar(p1)
    cbar1.set_ticks(cbar1.get_ticks())
    cbar1.set_label(r"$T$ [K]")
    ax[1].set_title("Dust temperature, {}".format(a_T))

    fig.tight_layout()
```

This `DustPyLib` modules creates the `dust_temperature.dat` file, which should be self-consistently calculated with `RADMC-3D`.
However, `DustPyLib` stores the temperatures that `DustPy` used during the simulation and assumes a vertically isothermal temperature and, furthermore, that all dust species have the same temperature as the gas. Note, the temperatures in the plot do not look vertically isothermal due to a combination of the coarse θ-grid and the logarithmic r-axis.

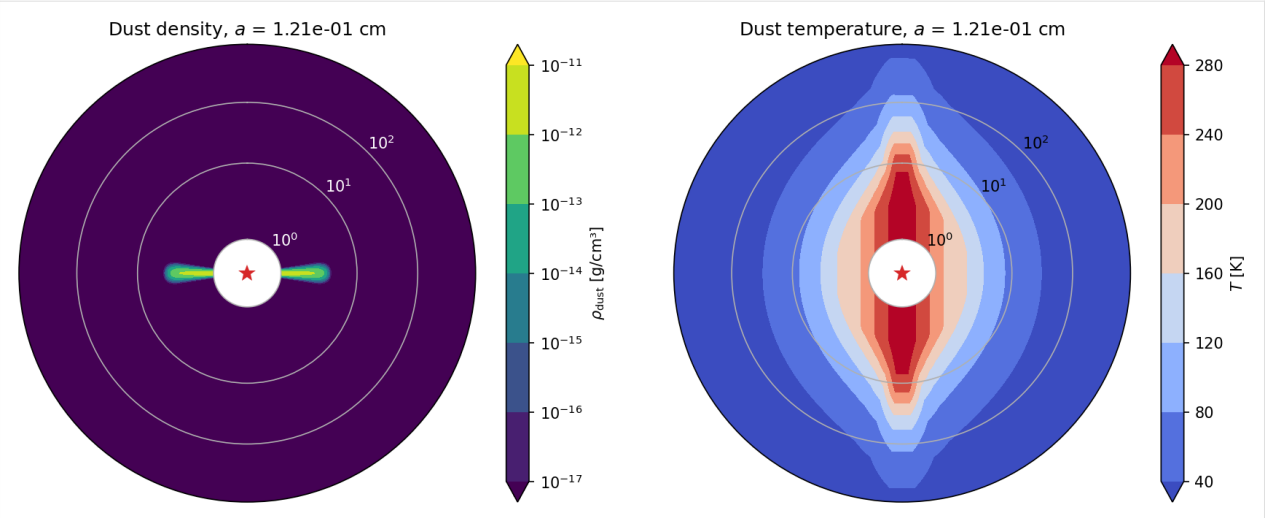
We can plot the total dust densities with `spec=-1`.

[23]:
```
plot_model(model, spec=-1)
```



Or plot specific dust species.

[24]:
```
plot_model(model, spec=12)
```



Thermal Monte Carlo run

In the first step we run the thermal Monte Carlo process in which the dust temperatures are computed. This can be done by executing the following command in the directory containing the `RADMC-3D` input files.

```
radmc3d mctherm
```

This will overwrite the `DustPy` temperatures in the `dust_temperature.dat` file with the temperatures self-consistently computed by `RADMC-3D`. This has been done for this notebook already and the temperature file can be simply copied.

[25]:
```
import shutil
```

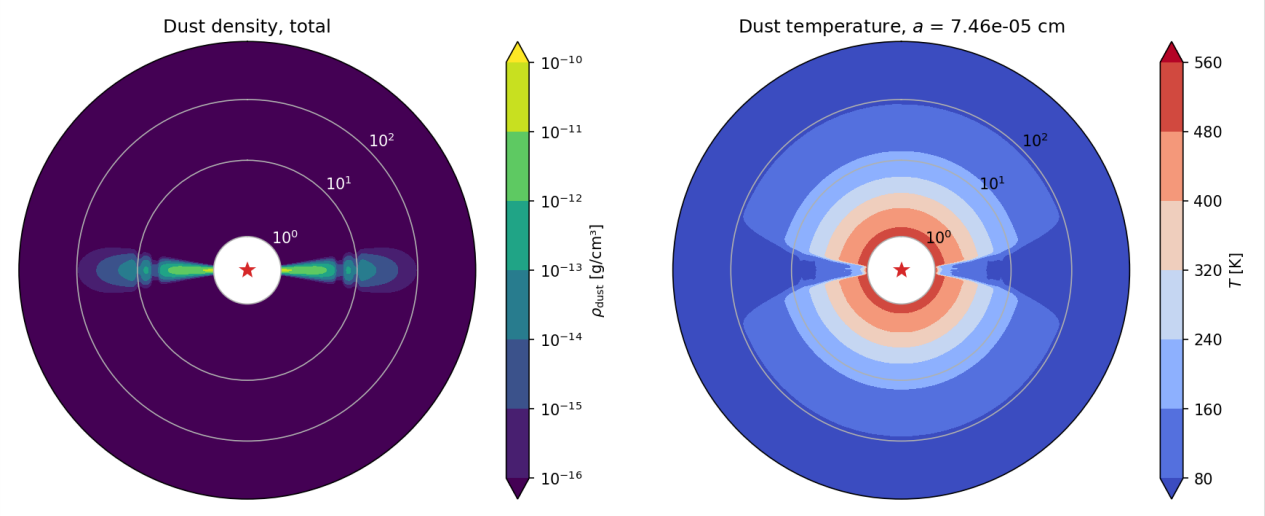
[26]:
```
shutil.copy("radmc3d/dust_temperature_mctherm.dat", "radmc3d/dust_temperature.dat")
```

[26]:
```
'radmc3d/dust_temperature.dat'
```

To plot the new temperatures we have to read the model files again.

[27]:
```
model_mc = radmc3d.read_model(datadir="radmc3d")
```

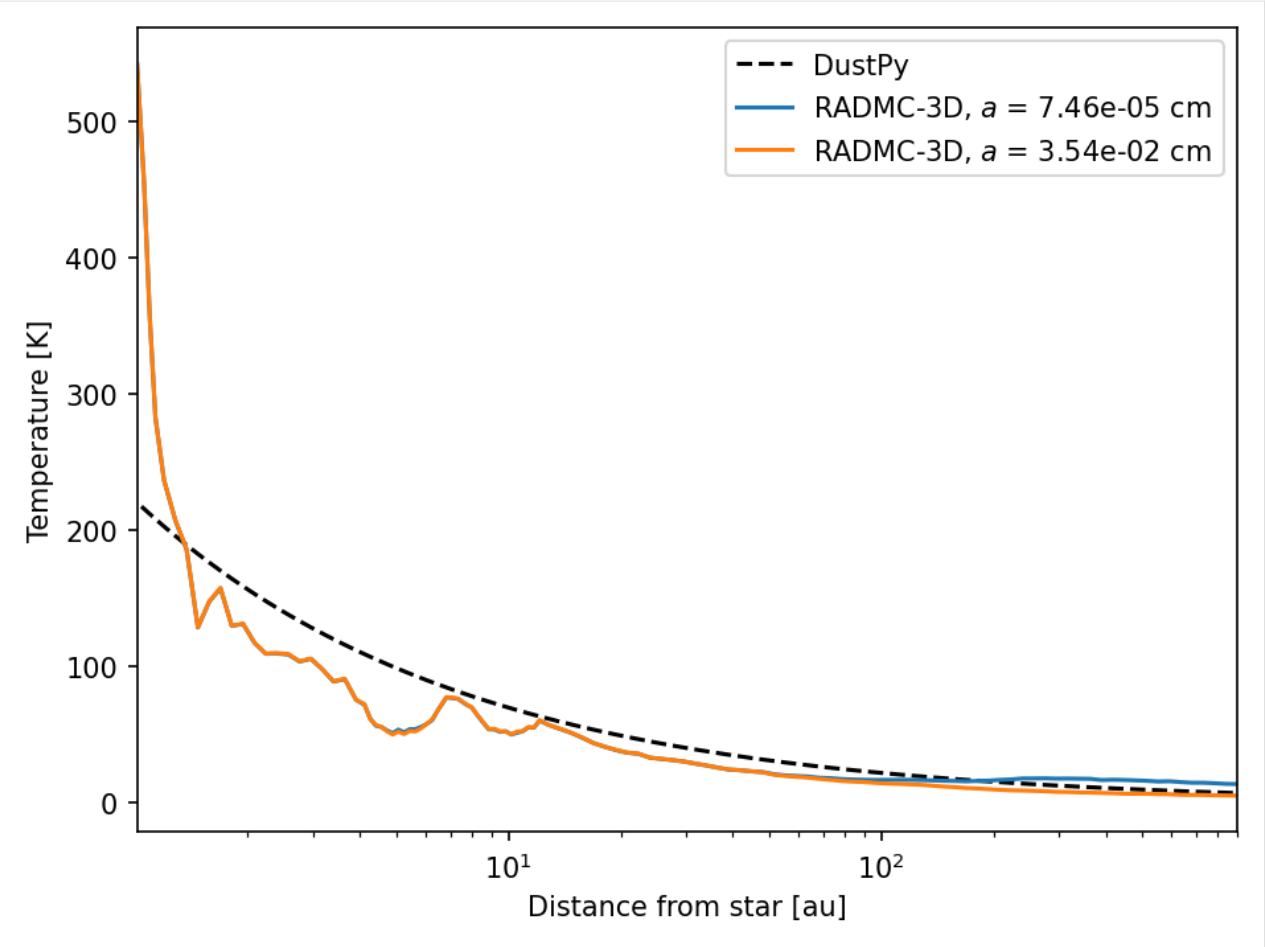
[28]:
```
plot_model(model_mc, spec=-1)
```



We can now compare the midplane temperature that `DustPy` uses with the temperatures that `RADMC-3D` calculated. We are taking the mean temperature in azimuthal direction (only one grid cell in this setup) and the mean temperature of 5 grid cells above and below the midplane to reduce the Monte Carlo noise.

Note that the temperature at the inner edge of the disk is significantly increased in the `RADMC-3D` model, since it it directly illuminated by the star. `DustPy`, on the other hand, does not assume that the disk ends at the inner edge of the radial grid.

[29]:
```
imid = int(model_mc.grid.theta.shape[0]/2)-1
irange = 5
fig, ax = plt.subplots()
ax.plot(data.grid.r/c.au, data.gas.T, "--", label="DustPy", c="black")
ax.plot(model_mc.grid.r/c.au, model_mc.T[:, imid-irange:imid+irange, :, 0].mean((-2, -
↪1)), label="RADMC-3D, $a$ = {:.2e} cm".format(rt.ac_grid[0]))
ax.plot(model_mc.grid.r/c.au, model_mc.T[:, imid-irange:imid+irange, :, 10].mean((-2, -
↪1)), label="RADMC-3D, $a$ = {:.2e} cm".format(rt.ac_grid[10]))
ax.set_xscale("log")
ax.set_xlim(model_mc.grid.r[0]/c.au, model_mc.grid.r[-1]/c.au)
ax.set_xlabel("Distance from star [au]")
ax.set_ylabel("Temperature [K]")
ax.legend()
fig.tight_layout()
```



Radio images

We can now use `RADMC-3D` to create millimeter observations as obtained for example with ALMA.

We are going to create two images: one at $\lambda = 0.88\,\mathrm{mm}$ (Band 7) and one at $\lambda = 1.3\,\mathrm{mm}$ (Band 6). Furthermore, we are focussing on the inner $50\,\mathrm{au}$ of the disk with 512 pixels in each direction. This can be done with the following `RADMC-3D` command:

```
radmc3d image lambdarange 880 1300 nlam 2 sizeau 100 npixx 512 npixy 512
```

Afterwards we rename the image file with:

```
mv image.out image_radio.out
```

For more details have a look at the imaging chapter of the `RADMC-3D` documentation.

We can now use `DustPyLib` to load the image with the following helper function, which returns a dictionary.

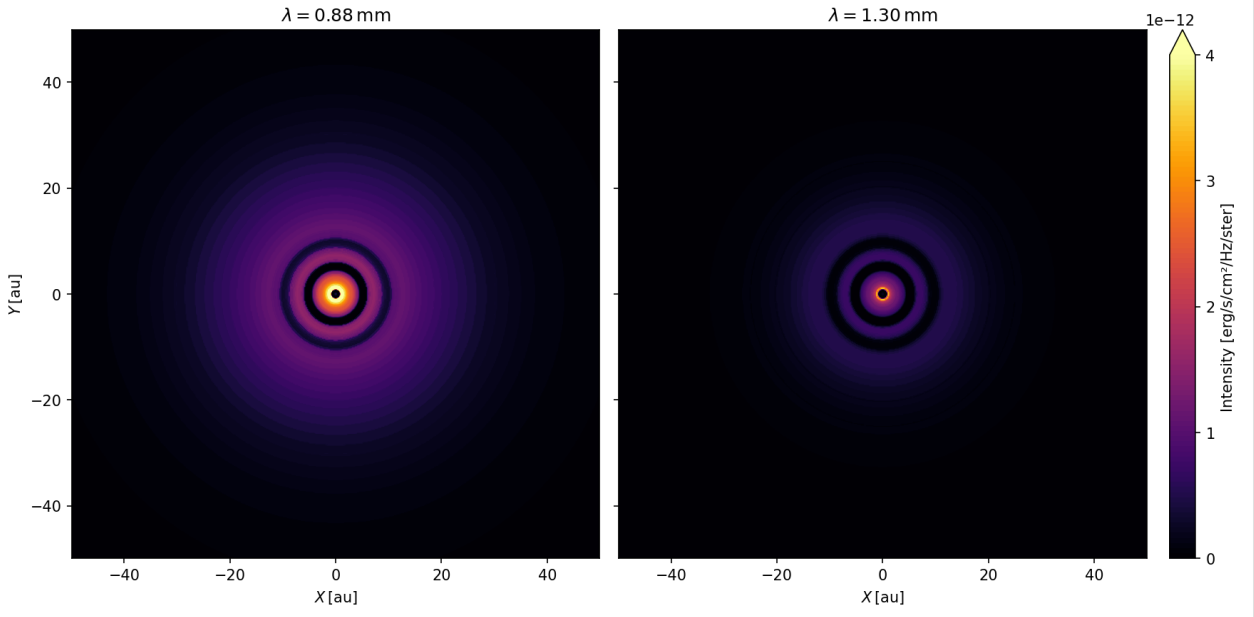
[30]:
```
image_radio = radmc3d.read_image("radmc3d/image_radio.out")
```

[31]:
```
width = 6.
height = width
x, y = image_radio["x"]/c.au, image_radio["y"]/c.au
Imax = 0.5*image_radio["I"].max()
```

(continues on next page)

(continued from previous page)

```
mag = np.floor(np.log10(Imax))
logmax = np.ceil(Imax * 10**(-mag))
levels = np.linspace(0., logmax, 100) * 10**mag
ticks = np.arange(int(logmax)+1) * 10**mag
fig, ax = plt.subplots(ncols=2, figsize=(2*width, height), sharey=True)
ax[0].set_aspect(1)
ax[0].contourf(x, y, image_radio["I"][..., 0].T, cmap="inferno", levels=levels, extend=
↪"max")
ax[0].set_title("$\lambda = {:.2f}\,\mathrm{{mm}}$".format(image_radio["lambda"][0]*10.))
ax[0].set_xlabel(r"$X\,\left[\mathrm{au}\right]$")
ax[0].set_ylabel(r"$Y\,\left[\mathrm{au}\right]$")
ax[1].set_aspect(1)
p = ax[1].contourf(x, y, image_radio["I"][..., 1].T, cmap="inferno", levels=levels,␣
↪extend="max")
ax[1].set_title("$\lambda = {:.2f}\,\mathrm{{mm}}$".format(image_radio["lambda"][1]*10.))
ax[1].set_xlabel(r"$X\,\left[\mathrm{au}\right]$")
fig.tight_layout()
fig.subplots_adjust(right=0.9)
pos01 = ax[1].get_position()
cb_ax = fig.add_axes([1.02*pos01.x1, pos01.y0, 0.02, pos01.y1-pos01.y0])
cbar = fig.colorbar(p, cax=cb_ax)
cbar.set_ticks(ticks)
cbar.set_label(r"Intensity [erg/s/cm$^2$/Hz/ster]")
```



Note that `RADMC-3D` comes with a Python support package to create `.fits` files from images which can the be further processed by CASA.

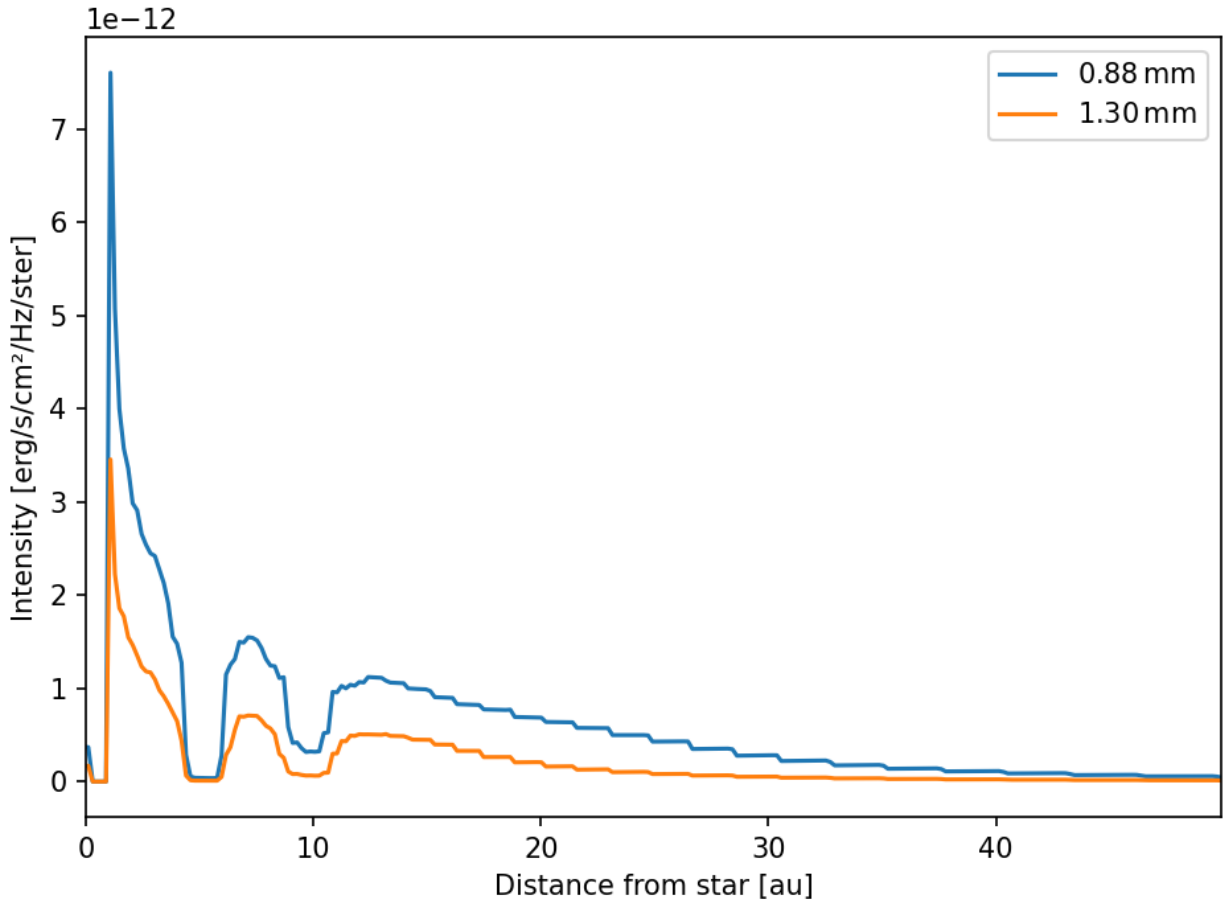
We can additionally plot the radial intensity profiles.

[32]:
```
x = image_radio["x"]/c.au
Nx = x.shape[0]
fig, ax = plt.subplots()
ax.plot(x, image_radio["I"][:, int(Nx/2), 0], label="${:.2f}\,\mathrm{{mm}}$".
```

(continues on next page)

(continued from previous page)

```
↪format(image_radio["lambda"][0]*10.))
ax.plot(x, image_radio["I"][:, int(Nx/2), 1], label="${:.2f}\,\mathrm{{mm}}$".
↪format(image_radio["lambda"][1]*10.))
ax.set_xlim(0., x.max())
ax.set_xlabel("Distance from star [au]")
ax.set_ylabel(r"Intensity [erg/s/cm$^2$/Hz/ster]")
ax.legend()
fig.tight_layout()
```



Spectral index maps

Since we created images at two different wavelengths we can create spectral index maps.

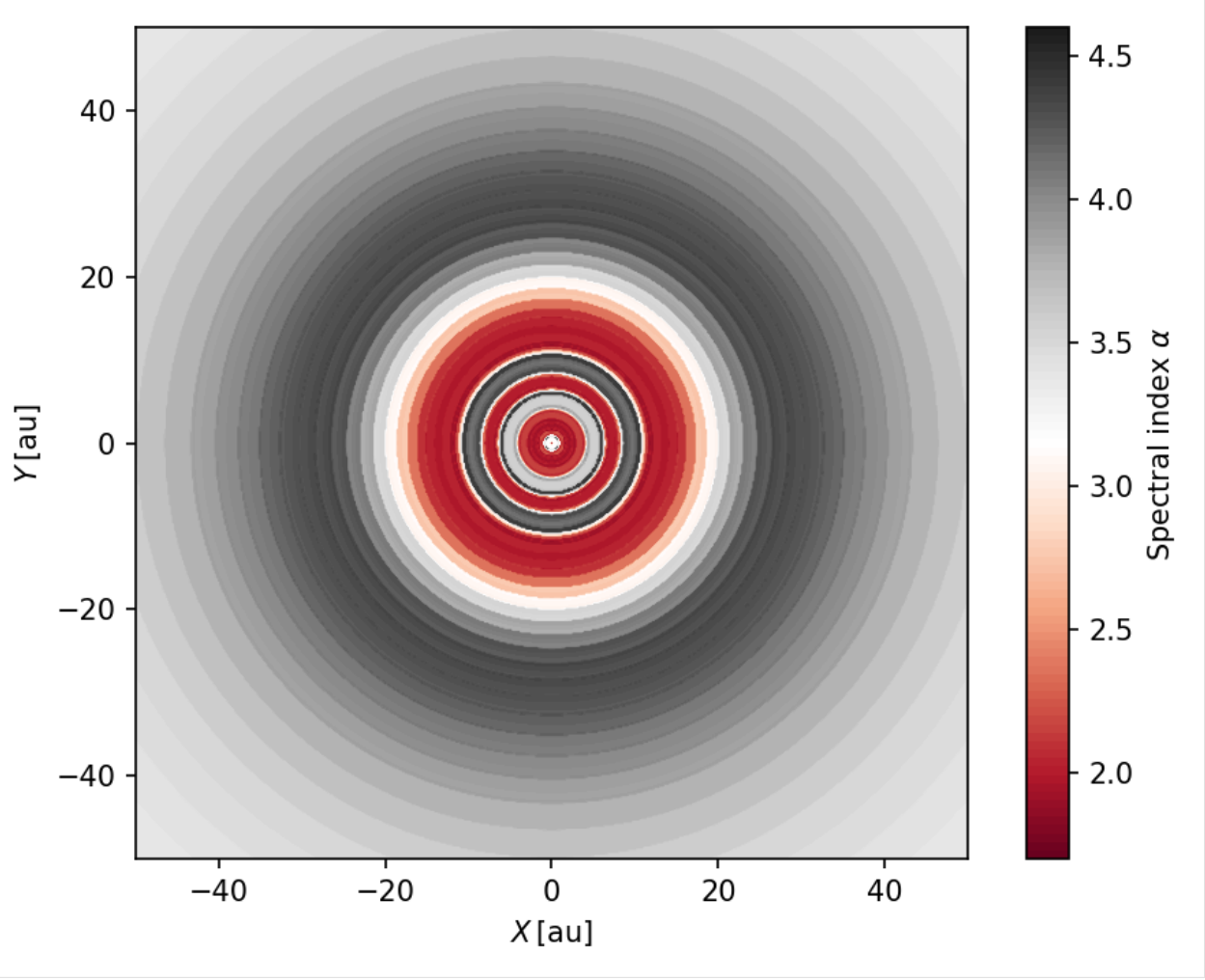
[33]:
```
alpha = -np.log10(image_radio["I"][..., 0]/image_radio["I"][..., 1]) / np.log10(image_
↪radio["lambda"][0]/image_radio["lambda"][1])
```

[34]:
```
x, y = image_radio["x"], image_radio["y"]
levels = np.linspace(1.7, 4.6, 100)
ticks = np.arange(2.0, 4.6, 0.5)
fig, ax = plt.subplots()
ax.set_aspect(1)
```

(continues on next page)

(continued from previous page)

```
p = ax.contourf(x/c.au, y/c.au, alpha.T, cmap="RdGy", levels=levels)
ax.set_xlabel(r"$X\,\left[\mathrm{au}\right]$")
ax.set_ylabel(r"$Y\,\left[\mathrm{au}\right]$")
cbar = plt.colorbar(p)
cbar.set_ticks(ticks)
cbar.set_label(r"Spectral index $\alpha$")
ax.set_xlim(-50., 50)
ax.set_ylim(-50., 50)
fig.tight_layout()
```



Polarization maps

To create images with polarization information we have to append the `stokes` flag to the `RADMC-3D` command. We only do this here for a single wavelength

```
radmc3d image lambda 880 sizeau 100 npixx 512 npixy 512 stokes
```

and rename the image file

```
mv image.out image_polarization.out
```

[35]:
```
image_polarization = radmc3d.read_image("radmc3d/image_polarization.out")
```

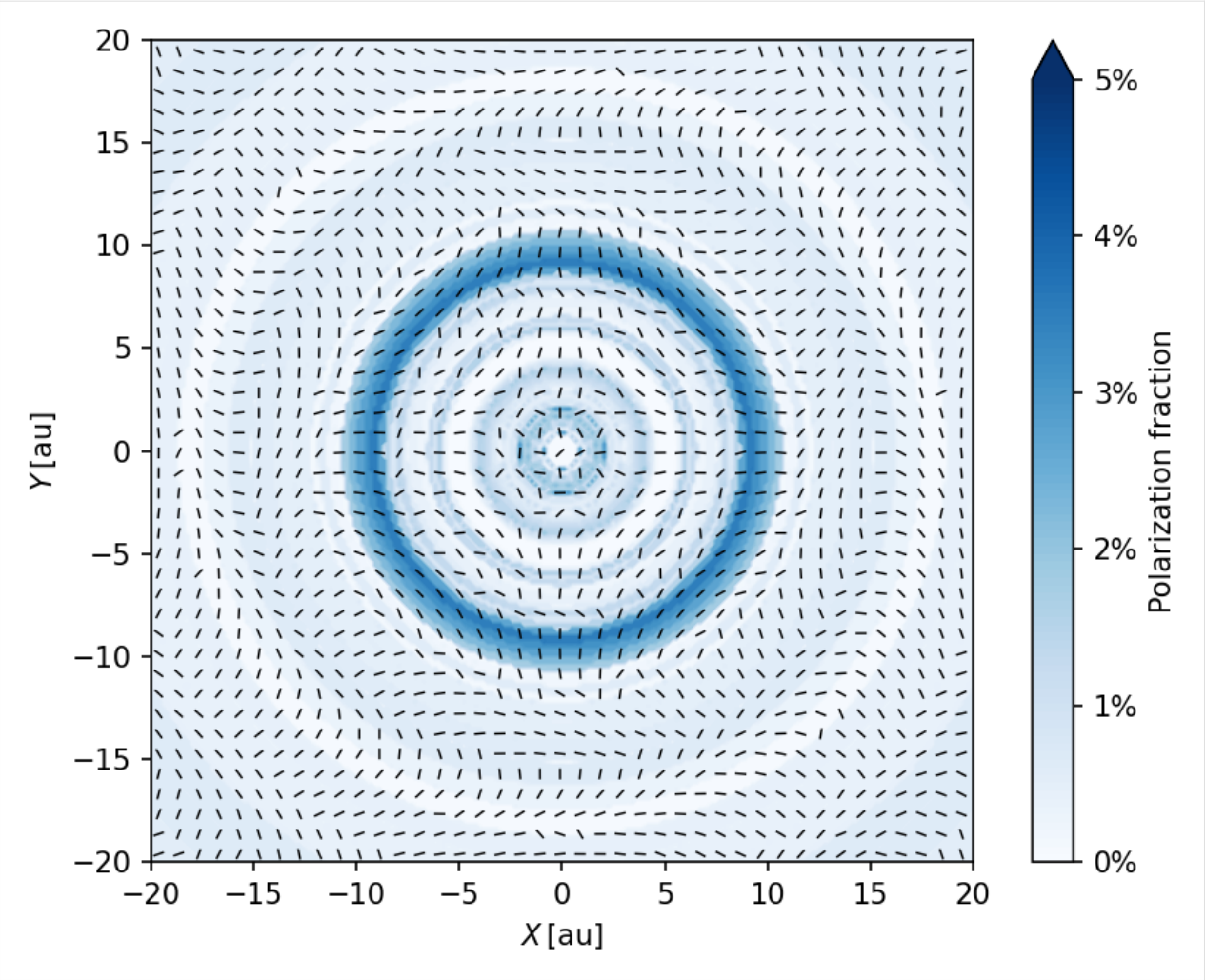
Now we can calculate the polarization fraction.

[36]:
```
I, Q, U = image_polarization["I"], image_polarization["Q"], image_polarization["U"]
P = np.sqrt(Q**2+U**2)/I
```

Then we have to convert the Stokes-Q and Stokes-U into the (x, y)-directions of the electric fields. Details on this can be found int the `RADMC-3D` documentation.

[37]:
```
chi = 0.5 * np.arctan2(U, Q)
Ex = np.cos(chi)
Ey = np.sin(chi)
```

[38]:
```
skip = 5
x, y = image_polarization["x"]/c.au, image_polarization["y"]/c.au
levels = np.linspace(0., 0.05, 100)
ticks = np.arange(0., 0.06, 0.01)
fig, ax = plt.subplots()
ax.set_aspect(1)
p = ax.contourf(x, y, P[..., 0].T, cmap="Blues", levels=levels, extend="max")
ax.quiver(x[::skip], y[::skip], Ex[::skip, ::skip, 0], Ey[::skip, ::skip, 0], pivot="mid
↪", headwidth=0, headlength=0, headaxislength=0, color="black", scale=75)
ax.set_xlabel(r"$X\,\left[\mathrm{au}\right]$")
ax.set_ylabel(r"$Y\,\left[\mathrm{au}\right]$")
cbar = plt.colorbar(p)
cbar.set_ticks(ticks)
cbar.set_ticklabels(["{:.0f}%".format(t*100.) for t in cbar.get_ticks()])
cbar.set_label("Polarization fraction")
ax.set_xlim(-20, 20)
ax.set_ylim(-20, 20)
fig.tight_layout()
```



For more information on this polarization pattern, please have a look at Kataoka et al. (2015).

Optical light images

While images at radio wavelengths typically trace the disk midplane, images at optical wavelengths, such as taken with SPHERE at the VLT are heavily affected by scattering at the disk surface.

The approach is identical to the radio images. However, we additionally incline the disk and use a larger field of view to give a better impression of the three-dimensional structure of the disk. We are taking an image at $\lambda = 1.6\,\mu\mathrm{m}$ with the following command

```
radmc3d image lambda 1.6 sizeau 400 npixx 512 npixy 512 incl 50 posang 53
```

and rename the image file

```
mv image.out image_optical.out
```

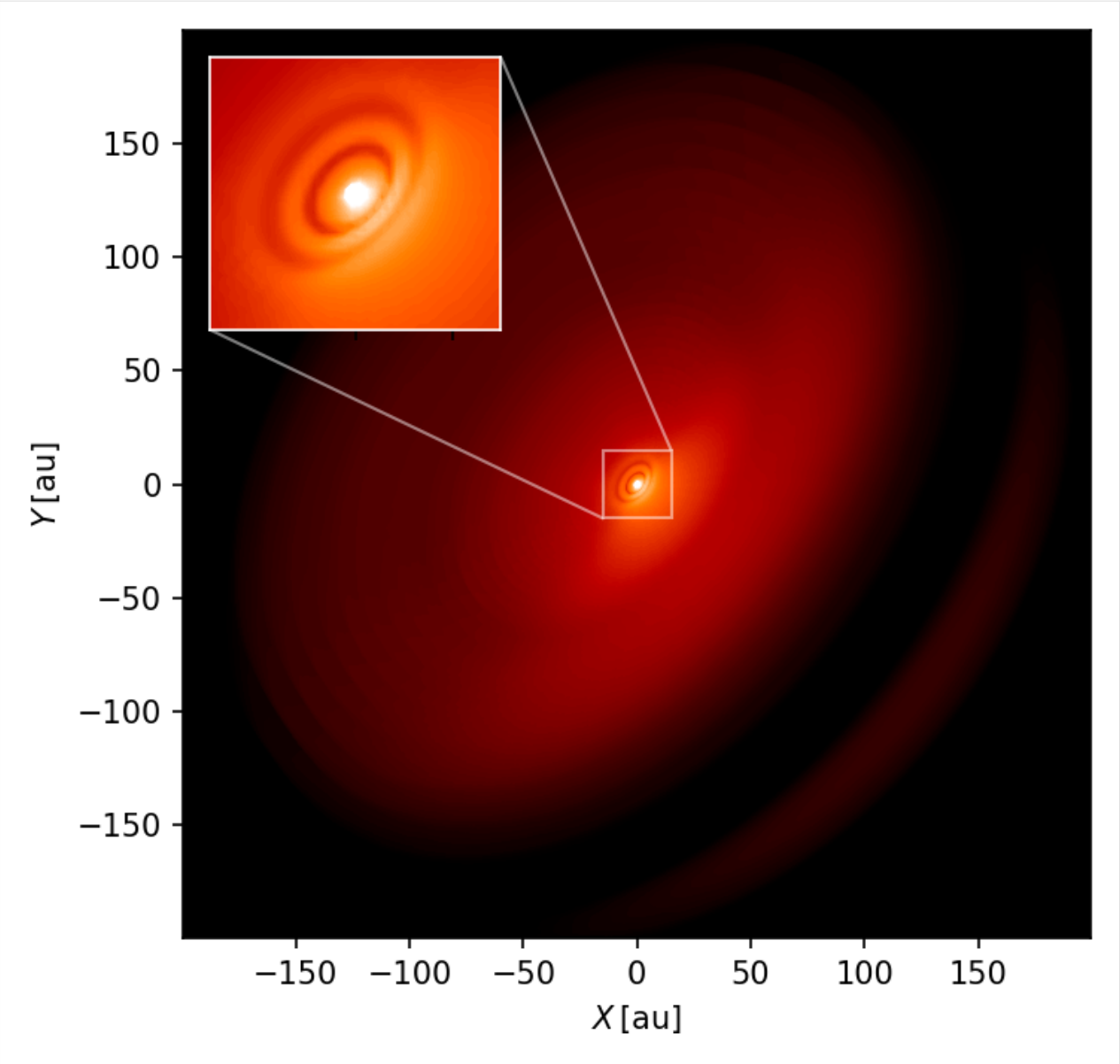
[39]:
```
image_optical = radmc3d.read_image("radmc3d/image_optical.out")
```

[40]:
```
x, y = image_optical["x"]/c.au, image_optical["y"]/c.au
I = image_optical["I"]
```

(continues on next page)

(continued from previous page)

```
Imax = image_optical["I"].max()
levmax = np.log10(Imax)-2
levels = np.linspace(levmax-6, levmax, 100)
fig, ax = plt.subplots()
ax.set_aspect(1)
ax.contourf(x, y, np.log10(I[..., 0].T), cmap="gist_heat", levels=levels, extend="both")
ax.set_xlabel(r"$X\,\left[\mathrm{au}\right]$")
ax.set_ylabel(r"$Y\,\left[\mathrm{au}\right]$")
ax_ins = ax.inset_axes([0.03, 0.67, 0.32, 0.30])
ax_ins.contourf(x, y, np.log10(I[..., 0].T), cmap="gist_heat", levels=levels, extend=
↪"both")
ax_ins.set_xlim(-15, 15)
ax_ins.set_ylim(-15, 15)
ax_ins.xaxis.set_tick_params(labelbottom=False)
ax_ins.yaxis.set_tick_params(labelleft=False)
ax_ins.spines[["top", "left", "bottom", "right"]].set_color("white")
ax.indicate_inset_zoom(ax_ins, edgecolor="white")
fig.tight_layout()
```



The gaps created by the planets are barely visible in the inner part of the disk in this unconvolved image. Furthermore, we can see the dark midplane at the edge of the disk and some light scattered from the backside of the disk.

Spectral energy distribution

To create spectra we can use the following command

```
radmc3d sed
```

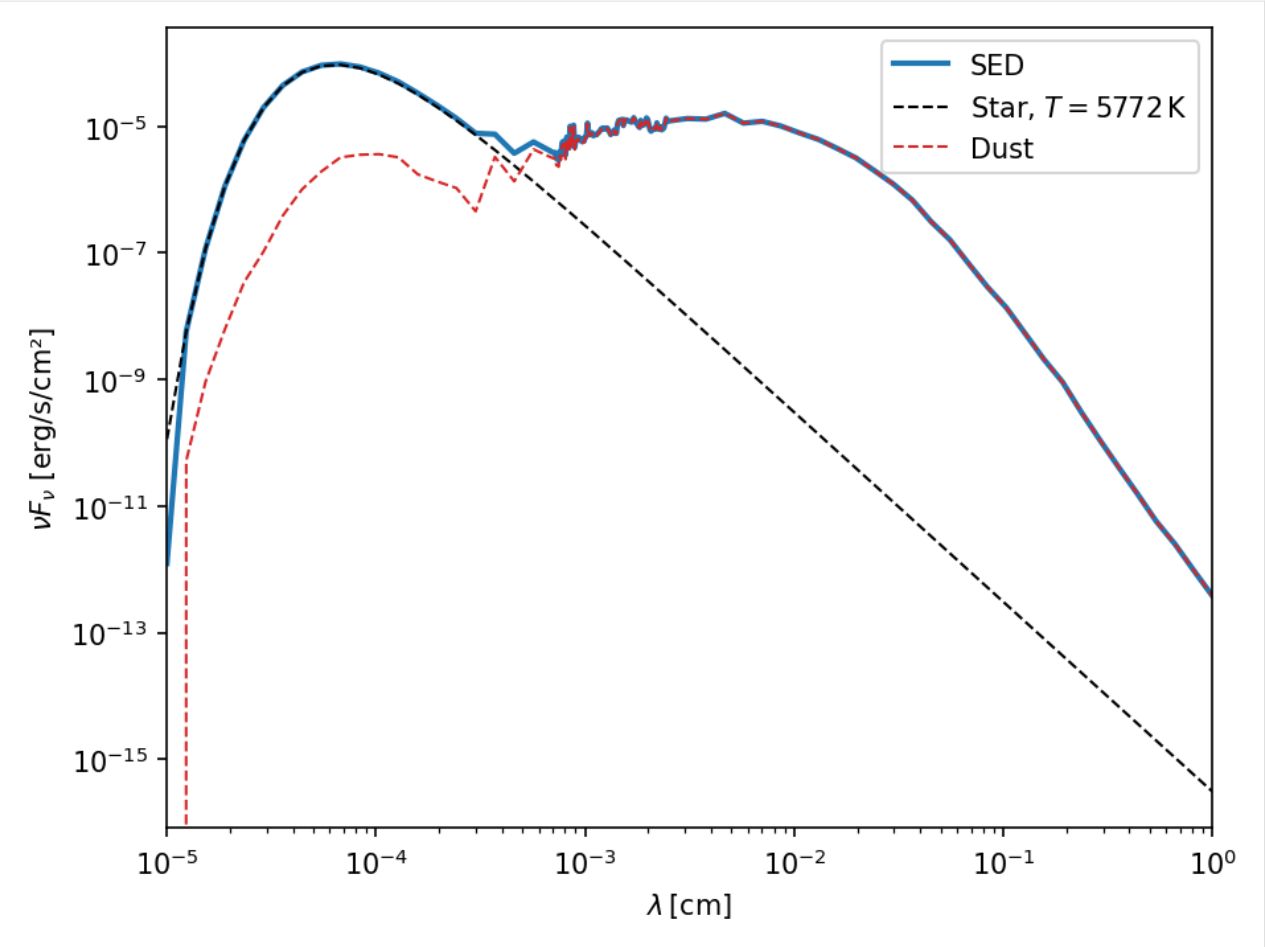
This will create an SED at every wavelength point of the `RADMC-3D` model. This will take a while, since it is creating an image at every wavelength point.

`RADMC-3D` will return an SED from a distance of $1\,\mathrm{pc}$.

[41]:
```
spectrum = radmc3d.read_spectrum("radmc3d/spectrum.out")
```

[42]:
```
from astropy.modeling.models import BlackBody
import astropy.constants as ac
import astropy.units as u
```

[43]:
```
lam, F = spectrum["lambda"], spectrum["flux"]
nu = ac.c/(lam*u.cm)
bb = BlackBody(temperature=rt.T_star_*u.K)
conv = (np.pi * ((rt.R_star_*u.cm)/(1.*u.pc))**2.).cgs.value
star = nu.cgs.value*bb(lam*u.cm)*conv
dust = nu.cgs.value*F-star.cgs.value
fig, ax = plt.subplots()
ax.plot(lam, nu.cgs.value*F, label="SED", lw=2)
ax.plot(lam, star, c="black", lw=1, ls="--", label="Star, $T={:.0f}\,\mathrm{{K}}$".
↪format(rt.T_star_))
ax.plot(lam, dust, c="C3", lw=1, ls="--", label="Dust")
ax.set_xscale("log")
ax.set_yscale("log")
ax.set_xlim(lam[0], lam[-1])
ax.set_xlabel(r"$\lambda\,\left[\mathrm{cm}\right]$")
ax.set_ylabel(r"$\nu F_\nu$ [erg/s/cm$^2$]")
ax.legend()
fig.tight_layout()
```



1.5 Substructures

The `dustpylib.substructures` submodule contains methods to impose substructure onto the disk profile.

1.5.1 Planetary gaps

This notebook discusses ways to impose gaps on the surface density profiles. Since `DustPy` is one-dimensional it is difficult to self-consistently model gaps carved by planets into the protoplanetary disk. One way of doing this, would be to set the gas surface density profile to analytical gap profile taken from multi-dimensional hydrodynamical simulation. However, in this case the gas would be static and non-evolving.

Another way would be to utilize the fact that the product of viscosity and gas surface density is constant in steady state.

$\nu\Sigma_{\mathrm{gas}} = \mathrm{const.}$

Imposing the inverse gap profile onto the viscosity would therefore create the desired gap profile on the gas surface density. Since the viscosity is proportial to the α-parameter, we therefore impose the inverse gap profile onto α.

This notebook adds gaps carved by the Solar System giant planet to `DustPy` simulations with different gap shapes. We therefore create a dictionary with the masses of the planets in g and their semi-major axis in cm.

[1]:
```
import dustpy.constants as c
```

[2]:
```
planets = {
    "jupiter": {
        "a": 5.2038 * c.au,
        "M": 317.8 * c.M_earth,
    },
    "saturn": {
        "a": 9.5826 * c.au,
        "M": 95.159 * c.M_earth,
    },
    "uranus": {
        "a": 19.19126 * c.au,
        "M": 14.536 * c.M_earth,
    },
    "neptune": {
        "a": 30.07 * c.au,
        "M": 17.147 * c.M_earth,
    },
}
```

Kanagawa et al. (2017)

`DustPyLib` contains the analytical function of Kanagawa et al. (2017) fitted to gap profiles obtained from hydrodynamical simulations.

We first create a `DustPy` simulation object.

[3]:
```
from dustpy import Simulation
```

[4]:
```
s = Simulation()
```

Before we initialize the simulation object, we refine the radial grid around the planet locations.

[5]:
```
from dustpylib.grid.refinement import refine_radial_local
import numpy as np
```

[6]:
```
ri = np.geomspace(s.ini.grid.rmin, s.ini.grid.rmax, s.ini.grid.Nr)
for planet in planets.values():
    ri = refine_radial_local(ri, planet["a"], num=3)
```

[7]:
```
s.grid.ri = ri
```

Now we can initialize the simulation object with the new grid.

[8]:
```
s.initialize()
```

In the next step we add a group for the planets and add their masses as fields.

[9]:
```
s.addgroup("planets", description="Planets")
for name, planet in planets.items():
```

(continues on next page)

(continued from previous page)

```
    s.planets.addgroup(name, description="Planet {}".format(name.title()))
    s.planets.__dict__[name].addfield("M", planet["M"], description="Mass in g")
    s.planets.__dict__[name].addfield("a", planet["a"], description="Semi-major axis in␣
↪cm")
```

[10]:
```
s.planets
```

[10]:
```
Group (Planets)
---------------
    jupiter      : Group (Planet Jupiter)
    neptune      : Group (Planet Neptune)
    saturn       : Group (Planet Saturn)
    uranus       : Group (Planet Uranus)
  -----
```

[11]:
```
s.planets.jupiter
```

[11]:
```
Group (Planet Jupiter)
----------------------
    a            : Field (Semi-major axis in cm)
    M            : Field (Mass in g)
  -----
```

Since we are going to impose the inverse gap profile on α, we need to copy and store the unperturbed α-profile for later usage.

[12]:
```
alpha0 = s.gas.alpha.copy()
```

In the next step we need to define an updater function for α, which imposes the inverse gap profile. For this we can use the `kanagawa2017()` function of `DustPyLib`.

[13]:
```
from dustpylib.substructures.gaps import kanagawa2017
from scipy.interpolate import interp1d
```

[14]:
```
def alpha(s):
    # Unperturbed profile
    alpha = alpha0.copy()
    # Iteration over all planets
    for name, planet in s.planets.__dict__.items():
        # Skip hidden fields:
        if name.startswith("_"):
            continue
        # Dimensionless planet mass
        q = planet.M/s.star.M
        # Interpolation of aspect ratio and alpha0 onto planet position
        f_h = interp1d(s.grid.r, s.gas.Hp/s.grid.r)
        h = f_h(planet.a)
        f_alp = interp1d(s.grid.r, alpha0)
        alp = f_alp(planet.a)
        # Inverse alpha-profile
        alpha /= kanagawa2017(s.grid.r, planet.a, q, h, alp)
    return alpha
```

This function can now be added as updater of the α field.

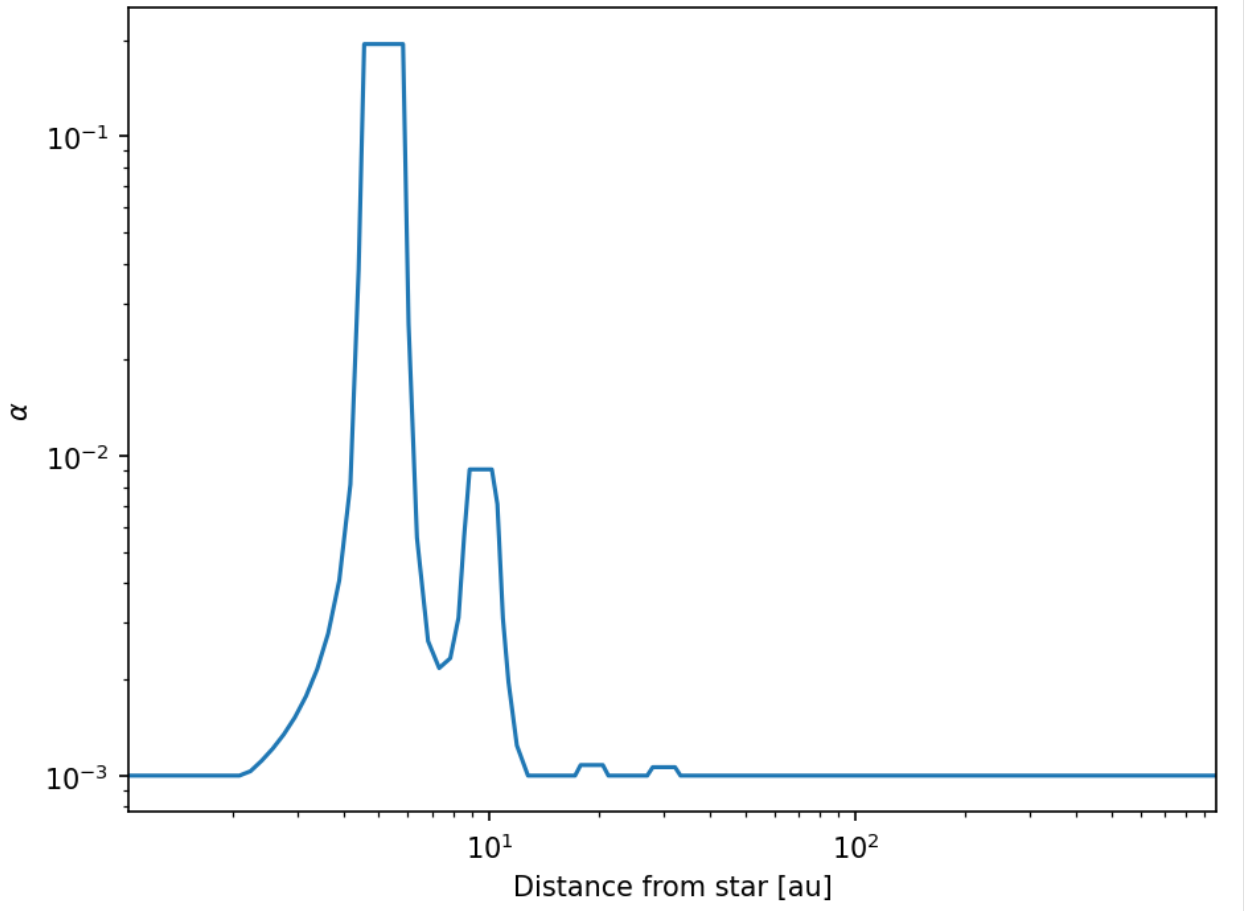
[15]:
```
s.gas.alpha.updater = alpha
```

After updating the simulation object, α should have the desired profile.

[16]:
```
s.update()
```

[17]:
```
import matplotlib.pyplot as plt
plt.rcParams["figure.dpi"] = 150.
```

[18]:
```
fig, ax = plt.subplots()
ax.loglog(s.grid.r/c.au, s.gas.alpha)
ax.set_xlim(s.grid.r[0]/c.au, s.grid.r[-1]/c.au)
ax.set_xlabel("Distance from star [au]")
ax.set_ylabel(r"$\alpha$")
fig.tight_layout()
```

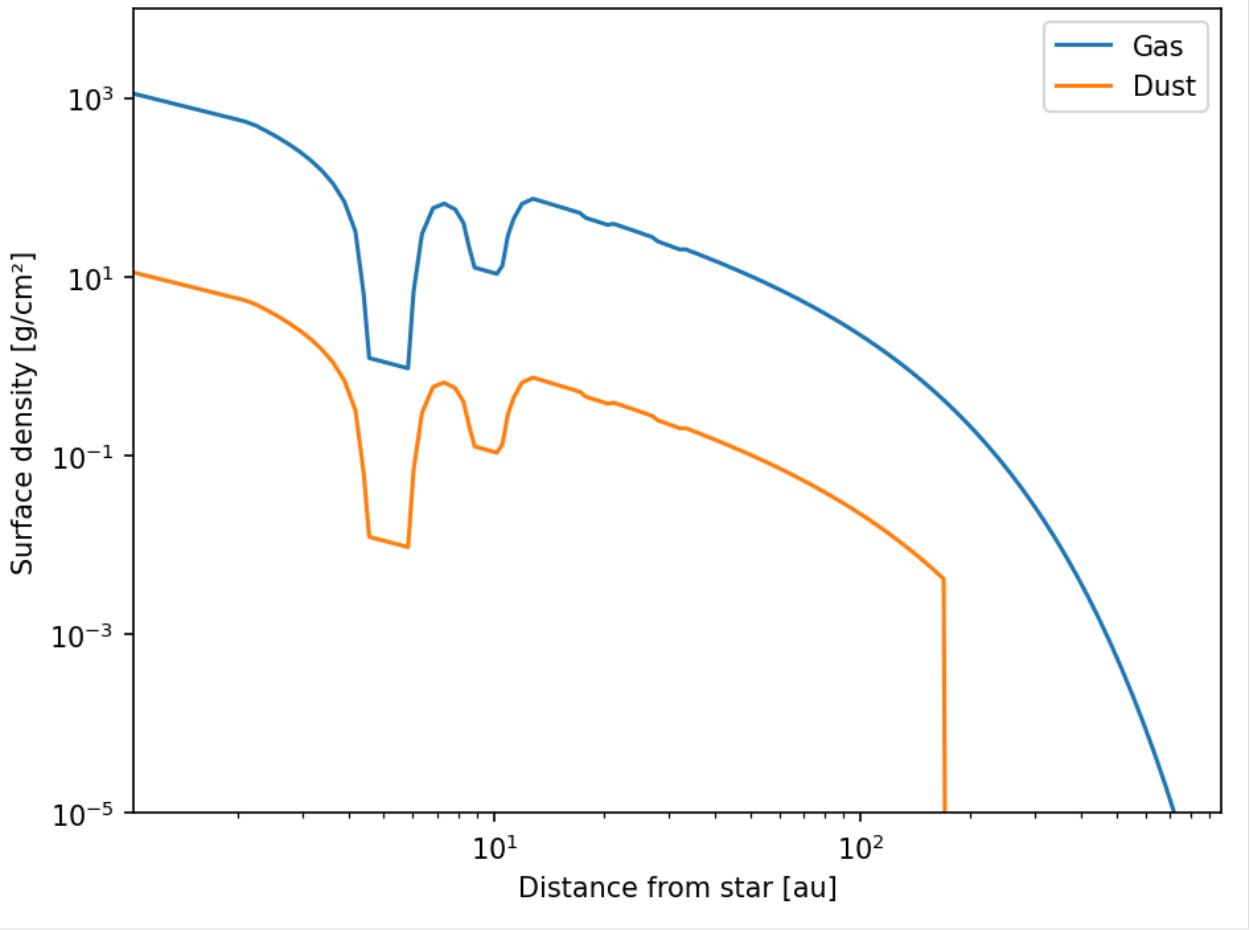


The simulation would now be ready to go by executing `s.run()` and the gap profile would be imposed on the gas surface density on viscous time scales.

But we could also impose this profile on the gas and dust surface densities from the beginning.

[19]:
```
s.gas.Sigma[...] /= s.gas.alpha/alpha0
s.dust.Sigma[...] /= (s.gas.alpha/alpha0)[:, None]
```

[20]:
```
fig, ax = plt.subplots()
ax.loglog(s.grid.r/c.au, s.gas.Sigma, label="Gas")
ax.loglog(s.grid.r/c.au, s.dust.Sigma.sum(-1), label="Dust")
ax.set_xlim(s.grid.r[0]/c.au, s.grid.r[-1]/c.au)
ax.set_ylim(1.e-5, 1.e4)
ax.set_xlabel("Distance from star [au]")
ax.set_ylabel("Surface density [g/cm$^2$]")
ax.legend()
fig.tight_layout()
```



Duffell (2020)

`DustPyLib` furthermore contains the empirically derived gap shapes of Duffell (2020).

This example analogous to the example above and implements the Solar System giants to a `DustPy` simulation.

[21]:
```
from dustpy import Simulation
```

[22]:
```
s = Simulation()
```

Before we initialize the simulation object, we refine the radial grid around the planet locations.

[23]:
```
from dustpylib.grid.refinement import refine_radial_local
import numpy as np
```

[24]:
```
ri = np.geomspace(s.ini.grid.rmin, s.ini.grid.rmax, s.ini.grid.Nr)
for planet in planets.values():
    ri = refine_radial_local(ri, planet["a"], num=3)
```

[25]:
```
s.grid.ri = ri
```

Now we can initialize the simulation object with the new grid.

[26]:
```
s.initialize()
```

In the next step we add a group for the planets and add their masses as fields.

[27]:
```
s.addgroup("planets", description="Planets")
for name, planet in planets.items():
    s.planets.addgroup(name, description="Planet {}".format(name.title()))
    s.planets.__dict__[name].addfield("M", planet["M"], description="Mass in g")
    s.planets.__dict__[name].addfield("a", planet["a"], description="Semi-major axis in␣
↪cm")
```

[28]:
```
s.planets
```

[28]:
```
Group (Planets)
---------------
    jupiter      : Group (Planet Jupiter)
    neptune      : Group (Planet Neptune)
    saturn       : Group (Planet Saturn)
    uranus       : Group (Planet Uranus)
  -----
```

[29]:
```
s.planets.jupiter
```

[29]:
```
Group (Planet Jupiter)
----------------------
    a            : Field (Semi-major axis in cm)
    M            : Field (Mass in g)
  -----
```

Since we are going to impose the inverse gap profile on α, we need to copy and store the unperturbed α-profile for later usage.

[30]:
```
alpha0 = s.gas.alpha.copy()
```

In the next step we need to define an updater function for α, which imposes the inverse gap profile. For this we can use the `duffell2020()` function of `DustPyLib`.

[31]:
```
from dustpylib.substructures.gaps import duffell2020
from scipy.interpolate import interp1d
```

[32]:
```
def alpha(s):
    # Unperturbed profile
```

(continues on next page)

(continued from previous page)

```
    alpha = alpha0.copy()
    # Iteration over all planets
    for name, p in s.planets.__dict__.items():
        # Skip hidden fields:
        if name.startswith("_"):
            continue
        # Dimensionless planet mass
        q = p.M/s.star.M
        # Interpolation of aspect ratio and alpha0 onto planet position
        f_h = interp1d(s.grid.r, s.gas.Hp/s.grid.r)
        h = f_h(p.a)
        f_alp = interp1d(s.grid.r, alpha0)
        alp = f_alp(p.a)
        # Inverse alpha-profile
        alpha /= duffell2020(s.grid.r, p.a, q, h, alp)
    return alpha
```

This function can now be added as updater of the α field.

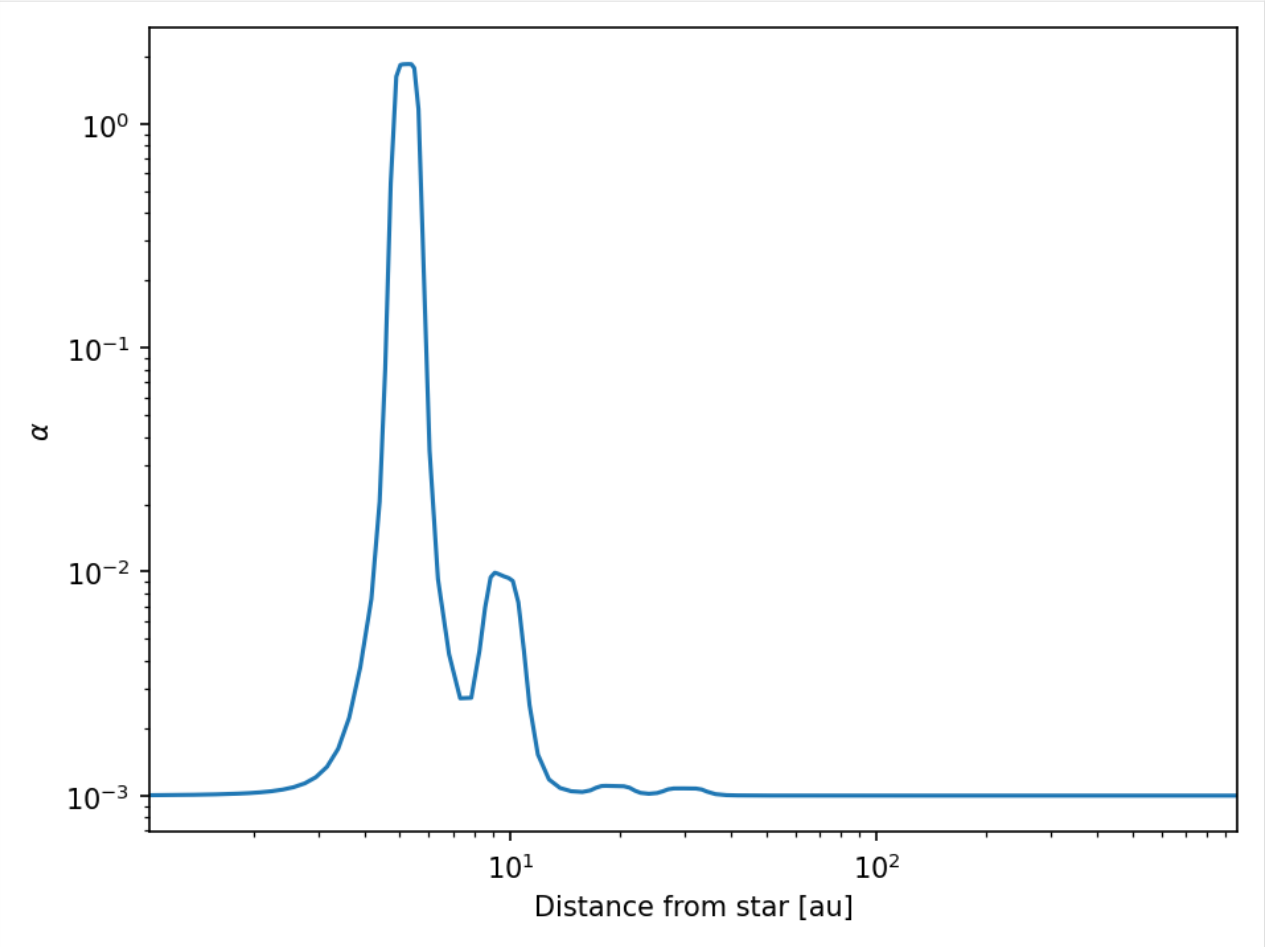
[33]:
```
s.gas.alpha.updater = alpha
```

After updating the simulation object, α should have the desired profile.

[34]:
```
s.update()
```

[35]:
```
import matplotlib.pyplot as plt
plt.rcParams["figure.dpi"] = 150.
```

[36]:
```
fig, ax = plt.subplots()
ax.loglog(s.grid.r/c.au, s.gas.alpha)
ax.set_xlim(s.grid.r[0]/c.au, s.grid.r[-1]/c.au)
ax.set_xlabel("Distance from star [au]")
ax.set_ylabel(r"$\alpha$")
fig.tight_layout()
```

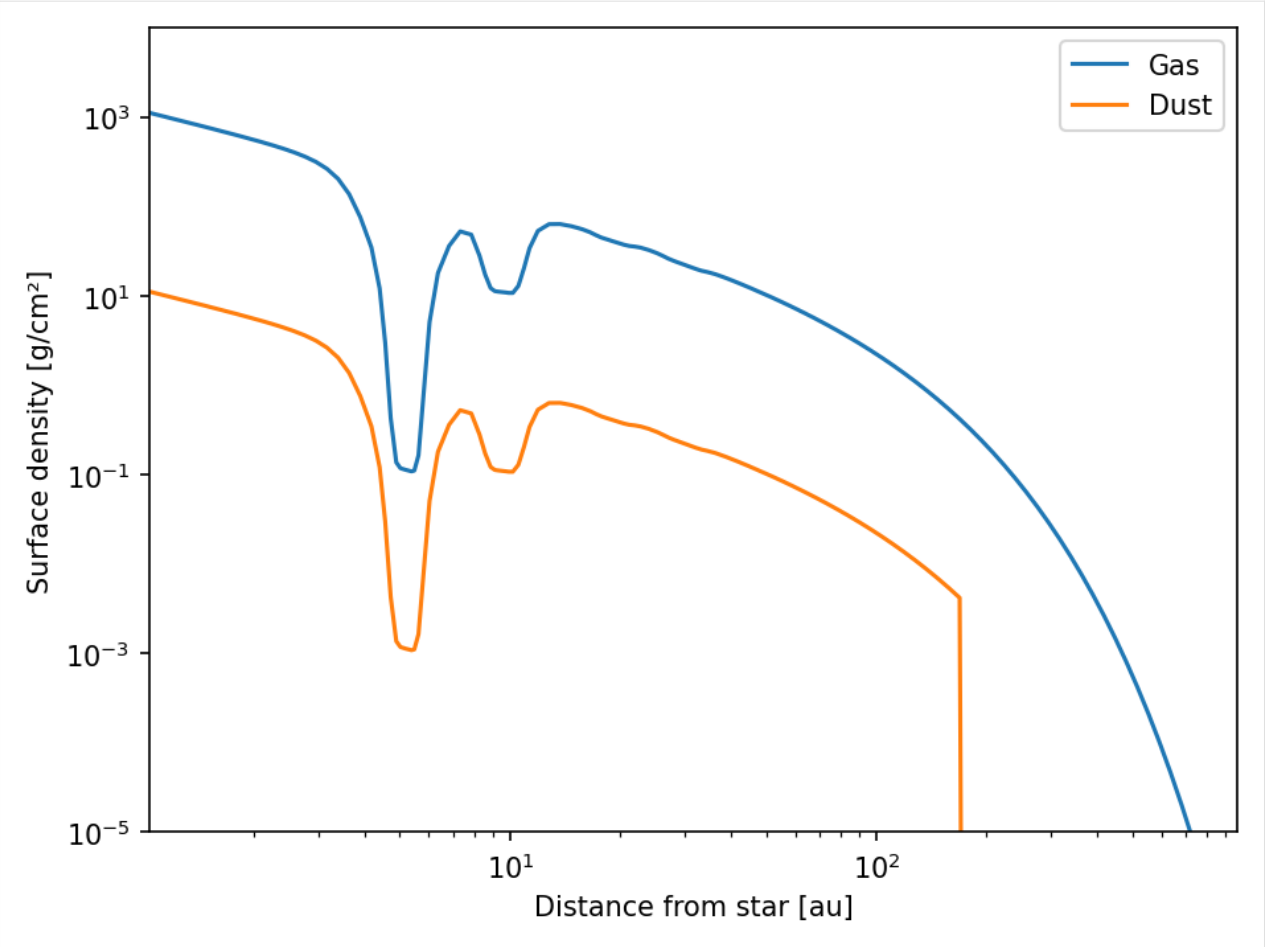


The simulation would now be ready to go by executing `s.run()` and the gap profile would be imposed on the gas surface density on viscous time scales.

But we could also impose this profile on the gas and dust surface densities from the beginning.

[37]:
```
s.gas.Sigma[...] /= s.gas.alpha/alpha0
s.dust.Sigma[...] /= (s.gas.alpha/alpha0)[:, None]
```

[38]:
```
fig, ax = plt.subplots()
ax.loglog(s.grid.r/c.au, s.gas.Sigma, label="Gas")
ax.loglog(s.grid.r/c.au, s.dust.Sigma.sum(-1), label="Dust")
ax.set_xlim(s.grid.r[0]/c.au, s.grid.r[-1]/c.au)
ax.set_ylim(1.e-5, 1.e4)
ax.set_xlabel("Distance from star [au]")
ax.set_ylabel("Surface density [g/cm$^2$]")
ax.legend()
fig.tight_layout()
```



1.6 Appendix A: Contributing/Bugs/Features

`DustPyLib` is a community project. New modules are always welcome. Please have a look at the Contribution Guidelines and the Code of Conduct.

1.6.1 Contributing

To contribute code please open a new pull request and describe the scope of the module your pull request introduces.

Please note, that additional features must also be described in the documentation. If usage of the module requires a citation, please descibe this in the respective example notebook.

1.6.2 Bug Reports

If you encounter a bug in any module of DustPyLib, please open a new bug report issue and describe the bug, the expected behavior, and steps how to reproduce the bug.

1.6.3 Feature Request

If you have an idea of a new feature, that is missing in DustPyLib, or if you want to suggest an improvement, please open a new feature request issue.

1.7 Module Reference

1.7.1 dustpylib Package

DustPyLib is a package with auxiliary tools and extensions for the dust evolution software DustPy.

1.7.2 dustpylib.dynamics Package

This package contains modules about dynamical extensions.

1.7.3 dustpylib.dynamics.backreaction Package

This package contains methods to implement the dust backreaction coefficients. The setup_backreaction(sim) function automatically implements all the required modifications to the Simulation object.

dustpylib.dynamics.backreaction.**BackreactionCoefficients**(*sim*)

Updater of the dust.backreaction Group.

Obtain the backreaction coefficients considering the contribution of each dust species. For more information check Garate et al. (2019), equations 23 - 26 in Appendix. This implementation does not consider the vertical structure. Hence, all the dust species and the gas feel the same backreaction.

Assigns the backreaction coefficients are returned to: sim.dust.backreaction.A sim.dust.backreaction.B

dustpylib.dynamics.backreaction.**BackreactionCoefficients_VerticalStructure**(*sim*)

Updater of the dust.backreaction Group.

Obtain the backreaction coefficients considering the vertical structure. For more information check Garate et al. (2019), equations 23 - 26 in Appendix.

Considers that the vertical distribution is gaussian for the gas and the dust. The final velocity is the mass flux vertical average at each location. For more information check Garate et al. (2019), equations 31 - 35 in Appendix.

This updater assigns: - the backreaction coefficients used for the gas calculations, accounting for dust vertical settling sim.dust.backreaction.A sim.dust.backreaction.B

- the backreaction coefficients used for the dust calculations, accounting for dust vertical settling

sim.dust.backreaction.A_dust_settling sim.dust.backreaction.B_dust_settling

dustpylib.dynamics.backreaction.**dustDiffusivity_Backreaction**(*sim*)

Reduces the dust diffusivity, accounts for the effect of the local dust-to-gas ratio.

dustpylib.dynamics.backreaction.**setup_backreaction**(*sim*, *vertical_setup=False*)

Add the backreaction setup to your simulation object. Call the backreaction setup function after the initialization and then run, as follows:

sim.initialize() setup_backreaction(sim) sim.run()

Functions

BackreactionCoefficients(sim)	Updater of the dust.backreaction Group.
BackreactionCoefficients_VerticalStructure(s	Updater of the dust.backreaction Group.
dustDiffusivity_Backreaction(sim)	Reduces the dust diffusivity, accounts for the effect of the local dust-to-gas ratio.
setup_backreaction(sim[, vertical_setup])	Add the backreaction setup to your simulation object.
vrad_dust_BackreactionVerticalStructure(sim)	

BackreactionCoefficients

dustpylib.dynamics.backreaction.**BackreactionCoefficients**(*sim*)

Updater of the dust.backreaction Group.

Obtain the backreaction coefficients considering the contribution of each dust species. For more information check Garate et al. (2019), equations 23 - 26 in Appendix. This implementation does not consider the vertical structure. Hence, all the dust species and the gas feel the same backreaction.

Assigns the backreaction coefficients are returned to: sim.dust.backreaction.A sim.dust.backreaction.B

BackreactionCoefficients_VerticalStructure

dustpylib.dynamics.backreaction.**BackreactionCoefficients_VerticalStructure**(*sim*)

Updater of the dust.backreaction Group.

Obtain the backreaction coefficients considering the vertical structure. For more information check Garate et al. (2019), equations 23 - 26 in Appendix.

Considers that the vertical distribution is gaussian for the gas and the dust. The final velocity is the mass flux vertical average at each location. For more information check Garate et al. (2019), equations 31 - 35 in Appendix.

This updater assigns: - the backreaction coefficients used for the gas calculations, accounting for dust vertical settling sim.dust.backreaction.A sim.dust.backreaction.B

- the backreaction coefficients used for the dust calculations, accounting for dust vertical settling

sim.dust.backreaction.A_dust_settling sim.dust.backreaction.B_dust_settling

dustDiffusivity_Backreaction

dustpylib.dynamics.backreaction.**dustDiffusivity_Backreaction**(*sim*)

Reduces the dust diffusivity, accounts for the effect of the local dust-to-gas ratio.

setup_backreaction

dustpylib.dynamics.backreaction.**setup_backreaction**(*sim*, *vertical_setup=False*)

Add the backreaction setup to your simulation object. Call the backreaction setup function after the initialization and then run, as follows:

sim.initialize() setup_backreaction(sim) sim.run()

vrad_dust_BackreactionVerticalStructure

dustpylib.dynamics.backreaction.**vrad_dust_BackreactionVerticalStructure**(*sim*)

1.7.4 dustpylib.grid Package

This package contains methods to manipulate the grid.

1.7.5 dustpylib.grid.refinement Package

This package contains methods to refine the grid.

dustpylib.grid.refinement.**refine_radial_local**(*ri*, *r0*, *num=3*)

Function refines the radial grid locally bysplitting grid cells recursively at a specific location.

Parameters

- **ri** (*array-like, (Nr,)*) – Radial grid cell interfaces
- **r0** (*float*) – Radial location to be refined
- **num** (*int, optional, default: 3*) – Number of refinement steps

Returns

ri_fine – Refined radial grid cell interfaces

Return type

array-like, (Nr+,)

Functions

refine_radial_local(ri, r0[, num])	Function refines the radial grid locally bysplitting grid cells recursively at a specific location.

refine_radial_local

dustpylib.grid.refinement.**refine_radial_local**(*ri*, *r0*, *num=3*)

Function refines the radial grid locally bysplitting grid cells recursively at a specific location.

Parameters

- **ri** (*array-like, (Nr,)*) – Radial grid cell interfaces
- **r0** (*float*) – Radial location to be refined
- **num** (*int, optional, default: 3*) – Number of refinement steps

Returns
ri_fine – Refined radial grid cell interfaces

Return type
array-like, (Nr+,)

1.7.6 dustpylib.planetesimals Package

This package contains modules about planetesimals.

1.7.7 dustpylib.planetesimals.formation Package

This package contains extensions about planetesimal formation.

dustpylib.planetesimals.formation.**drazkowska2016**(*OmegaK*, *rho_dust*, *rho_gas*, *Sigma_dust*, *St*, *p2g_crit=1.0*, *St_crit=0.01*, *zeta=0.01*)

Function calculates the dust source term due to planetesimal formation of Darzkowska et al. (2016).

Parameters

- **OmegaK** (*array-like, (Nr,)*) – Keplerian frequency
- **rho_dust** (*array-like, (Nr, Nm)*) – Midplane dust volume density
- **rho_gas** (*array-like, (Nr,)*) – Midplane gas volume density
- **Sigma_dust** (*array-like, (Nr, Nm)*) – Dust surface density
- **St** (*array-like, (Nr, Nm)*) – Stokes numbers
- **p2g_crit** (*float, optional, default: 1.*) – Critical midplane pebbles-to-gas ratio of particles above St_crit above which planetesimal formation is triggered
- **St_crit** (*float, optional, default: 0.01*) – Critical Stokes number above which dust particles contribute to trigger planetesimal formation
- **zeta** (*float, optional, default: 0.1*) – Planetesimal formation efficiency

Returns
S – Dust source terms due to planetesimal formation

Return type
array-like, (Nr, Nm)

dustpylib.planetesimals.formation.**miller2021**(*OmegaK*, *rho_dust*, *rho_gas*, *Sigma_dust*, *St*, *d2g_crit=1.0*, *n=0.03*, *zeta=0.1*)

Function calculates the dust source term due to planetesimal formation of Miller et al. (2021).

Parameters

- **OmegaK** (*array-like, (Nr,)*) – Keplerian frequency
- **rho_dust** (*array-like, (Nr, Nm)*) – Midplane dust volume density
- **rho_gas** (*array-like, (Nr,)*) – Midplane gas volume density
- **Sigma_dust** (*array-like, (Nr, Nm)*) – Dust surface density
- **St** (*array-like, (Nr, Nm)*) – Stokes numbers
- **d2g_crit** (*float, optional, default: 1.*) – Critical midplane dust-to-gas ratio above which planetesimal formation is triggered
- **n** (*float, optional, default: 0.03*) – Smoothness parameter of dust-to-gas ratio transition
- **zeta** (*float, optional, default: 0.1*) – Planetesimal formation efficiency

Returns
S – Dust source terms due to planetesimal formation

Return type
array-like, (Nr, Nm)

dustpylib.planetesimals.formation.**schoonenberg2018**(*OmegaK*, *rho_dust*, *rho_gas*, *Sigma_dust*, *St*, *d2g_crit=1.0*, *zeta=0.1*)

Function calculates the dust source term due to planetesimal formation of Schoonenberg et al. (2018).

Parameters

- **OmegaK** (*array-like, (Nr,)*) – Keplerian frequency
- **rho_dust** (*array-like, (Nr, Nm)*) – Midplane dust volume density
- **rho_gas** (*array-like, (Nr,)*) – Midplane gas volume density
- **Sigma_dust** (*array-like, (Nr, Nm)*) – Dust surface density
- **St** (*array-like, (Nr, Nm)*) – Stokes numbers
- **d2g_crit** (*float, optional, default: 1.*) – Critical midplane dust-to-gas ratio above which planetesimal formation is triggered
- **zeta** (*float, optional, default: 0.1*) – Planetesimal formation efficiency

Returns
S – Dust source terms due to planetesimal formation

Return type
array-like, (Nr, Nm)

Functions

drazkowska2016(OmegaK, rho_dust, rho_gas, ...)	Function calculates the dust source term due to planetesimal formation of Darzkowska et al. (2016).
miller2021(OmegaK, rho_dust, rho_gas, ...[, ...])	Function calculates the dust source term due to planetesimal formation of Miller et al. (2021).
schoonenberg2018(OmegaK, rho_dust, rho_gas, ...)	Function calculates the dust source term due to planetesimal formation of Schoonenberg et al. (2018).

drazkowska2016

dustpylib.planetesimals.formation.**drazkowska2016**(*OmegaK*, *rho_dust*, *rho_gas*, *Sigma_dust*, *St*, *p2g_crit=1.0*, *St_crit=0.01*, *zeta=0.01*)

Function calculates the dust source term due to planetesimal formation of Darzkowska et al. (2016).

Parameters

- **OmegaK** (*array-like, (Nr,)*) – Keplerian frequency
- **rho_dust** (*array-like, (Nr, Nm)*) – Midplane dust volume density
- **rho_gas** (*array-like, (Nr,)*) – Midplane gas volume density
- **Sigma_dust** (*array-like, (Nr, Nm)*) – Dust surface density
- **St** (*array-like, (Nr, Nm)*) – Stokes numbers
- **p2g_crit** (*float, optional, default: 1.*) – Critical midplane pebbles-to-gas ratio of particles above St_crit above which planetesimal formation is triggered
- **St_crit** (*float, optional, default: 0.01*) – Critical Stokes number above which dust particles contribute to trigger planetesimal formation
- **zeta** (*float, optional, default: 0.1*) – Planetesimal formation efficiency

Returns
S – Dust source terms due to planetesimal formation

Return type
array-like, (Nr, Nm)

miller2021

dustpylib.planetesimals.formation.**miller2021**(*OmegaK*, *rho_dust*, *rho_gas*, *Sigma_dust*, *St*, *d2g_crit=1.0*, *n=0.03*, *zeta=0.1*)

Function calculates the dust source term due to planetesimal formation of Miller et al. (2021).

Parameters

- **OmegaK** (*array-like, (Nr,)*) – Keplerian frequency
- **rho_dust** (*array-like, (Nr, Nm)*) – Midplane dust volume density
- **rho_gas** (*array-like, (Nr,)*) – Midplane gas volume density
- **Sigma_dust** (*array-like, (Nr, Nm)*) – Dust surface density
- **St** (*array-like, (Nr, Nm)*) – Stokes numbers
- **d2g_crit** (*float, optional, default: 1.*) – Critical midplane dust-to-gas ratio above which planetesimal formation is triggered
- **n** (*float, optional, default: 0.03*) – Smoothness parameter of dust-to-gas ratio transition
- **zeta** (*float, optional, default: 0.1*) – Planetesimal formation efficiency

Returns
S – Dust source terms due to planetesimal formation

Return type
array-like, (Nr, Nm)

schoonenberg2018

dustpylib.planetesimals.formation.**schoonenberg2018**(*OmegaK*, *rho_dust*, *rho_gas*, *Sigma_dust*, *St*, *d2g_crit=1.0*, *zeta=0.1*)

Function calculates the dust source term due to planetesimal formation of Schoonenberg et al. (2018).

Parameters

- **OmegaK** (*array-like, (Nr,)*) – Keplerian frequency
- **rho_dust** (*array-like, (Nr, Nm)*) – Midplane dust volume density
- **rho_gas** (*array-like, (Nr,)*) – Midplane gas volume density
- **Sigma_dust** (*array-like, (Nr, Nm)*) – Dust surface density
- **St** (*array-like, (Nr, Nm)*) – Stokes numbers
- **d2g_crit** (*float, optional, default: 1.*) – Critical midplane dust-to-gas ratio above which planetesimal formation is triggered
- **zeta** (*float, optional, default: 0.1*) – Planetesimal formation efficiency

Returns
S – Dust source terms due to planetesimal formation

Return type
array-like, (Nr, Nm)

1.7.8 dustpylib.radtrans Package

This package contains interfaces to radiative transfer codes from DustPy models.

1.7.9 dustpylib.radtrans.radmc3d Package

This package contains an interface to create RADMC-3D input files from DustPy models.

class dustpylib.radtrans.radmc3d.**Model**(*sim*, *ignore_last=True*)

Main model class that can read in DustPy models and can create RADMC-3D input files. Attributes with trailing underscore are imported from DustPy, while the other attributes will be used to create RADMC-3D input files.

read_image :

Reads RADMC-3D image file

read_spectrum :

Reads RADMC_3d spectrum file

write_files :

Writes all required RADMC-3D input files into the specified directory

write_opacity_files :

Writes only the required RADMC-3D opacity into files into the specified directory

Attributes

ac_grid
Particle size bin centers in cm for RADMC-3D model.

ai_grid
Particle size bin interfaces in cm for RADMC-3D model

phic_grid
Azimuthal grid cell centers in rad for RADMC-3D model.

phii_grid
Azimuthal grid cell interfaces in rad for RADMC-3D model.

rc_grid
Radial grid cell centers in cm for RADMC-3D model.

ri_grid
Radial grid cell interfaces in cm for RADMC-3D model.

thetac_grid
Polar grid cell centers in rad for RADMC-3D model.

thetai_grid
Polar grid cell interfaces in rad for RADMC-3D model.

Methods

write_files([datadir, write_opacities, ...])	Function writes all required RADMC-3D input files.
write_opacity_files([datadir, opacity, ...])	Function writes the required opacity files.

H_dust_
Dust scale heights array in cm from DustPy

M_star_
Stellar mass in g

R_star_
Stellar radius in cm

Sigma_dust_
Dust surface density profile in g/cm^2 from DustPy

T_gas_
Temperature profile in K from DustPy

T_star_
Stellar effective temperature in K

a_dust_
Particle size array in cm from DustPy

property ac_grid
Particle size bin centers in cm for RADMC-3D model. Do not set manually. Only use size bin interfaces.

property ai_grid
Particle size bin interfaces in cm for RADMC-3D model

datadir
Directory to store the RADMC-3D input files

lam_grid

Wavelength grid for RADMC-3D in cm

opacity

Opacity model. “birnstiel2018” (default) or “ricci2010”

property phic_grid

Azimuthal grid cell centers in rad for RADMC-3D model. Do not set manually. Only use cell interfaces.

property phii_grid

Azimuthal grid cell interfaces in rad for RADMC-3D model.

radmc3d_options

RADMC-3D options for radmc3d.inp file

property rc_grid

Radial grid cell centers in cm for RADMC-3D model. Do not set manually. Only use cell interfaces.

rc_grid_

Radial grid cell centers from DustPy model

property ri_grid

Radial grid cell interfaces in cm for RADMC-3D model.

ri_grid_

Radial grid cell interfaces from DustPy model

property thetac_grid

Polar grid cell centers in rad for RADMC-3D model. Do not set manually. Only use cell interfaces.

property thetai_grid

Polar grid cell interfaces in rad for RADMC-3D model.

write_files(*datadir=None*, *write_opacities=True*, *opacity=None*, *smooth_opacities=False*)

Function writes all required RADMC-3D input files.

Parameters

- **datadir** (*str, optional, default: None*) – Data directory in which the files are written. None defaults to the datadir attribute of the parent class.
- **write_opacities** (*booelan, optional, default: True*) – If False, does not compute nor write opacity files.
- **opacity** (*str, optional, default: None*) – Opacity model to be used. Either ‘birnstiel2018’ or ‘ricci2010’. None defaults to ‘birnstiel2018’.
- **smooth_opacities** (*bool, optional, default: False*) – Smooth the opacities by averaging over multiple particle sizes. This slows down the computation.

write_opacity_files(*datadir=None*, *opacity=None*, *smooth_opacities=False*)

Function writes the required opacity files.

Parameters

- **datadir** (*str, optional, default: None*) – Data directory in which the files are written. None defaults to the datadir attribute of the parent class.
- **opacity** (*str, optional, default: None*) – Opacity model to be used. Either ‘birnstiel2018’ or ‘ricci2010’. None defaults to ‘birnstiel2018’.

- **smooth_opacities** (*bool, optional, default: False*) – Smooth the opacities by averaging over multiple particle sizes. This slows down the computation.

dustpylib.radtrans.radmc3d.**read_image**(*path*)

This functions reads an image file created by RADMC-3D and returns a dictionary with the image data.

Parameters
path (*str*) – Path to the image data file

Returns
d – Dictionary with the image data

Return type
dict

dustpylib.radtrans.radmc3d.**read_model**(*datadir=''*)

This functions reads the RADMC-3D model files and returns a namespace with the data. It should only be used for models created by DustPyLib. For more complex models use Radmc3dPy.

Parameters
datadir (*str, optional, default: ""*) – The path of the directory with the RADMC-3D input files

Returns
data – Namespace with the model data

Return type
namespace

dustpylib.radtrans.radmc3d.**read_spectrum**(*path*)

This functions reads a spectrum file created by RADMC-3D and returns a dictionary with the SED data.

Parameters
path (*str*) – Path to the spectrum data file

Returns
d – Dictionary with the SED data

Return type
dict

Functions

read_image(path)	This functions reads an image file created by RADMC-3D and returns a dictionary with the image data.
read_model([datadir])	This functions reads the RADMC-3D model files and returns a namespace with the data.
read_spectrum(path)	This functions reads a spectrum file created by RADMC-3D and returns a dictionary with the SED data.

read_image

dustpylib.radtrans.radmc3d.**read_image**(*path*)

This functions reads an image file created by RADMC-3D and returns a dictionary with the image data.

Parameters
path (*str*) – Path to the image data file

Returns
d – Dictionary with the image data

Return type
dict

read_model

dustpylib.radtrans.radmc3d.**read_model**(*datadir=''*)

This functions reads the RADMC-3D model files and returns a namespace with the data. It should only be used for models created by DustPyLib. For more complex models use Radmc3dPy.

Parameters
datadir (*str, optional, default: ""*) – The path of the directory with the RADMC-3D input files

Returns
data – Namespace with the model data

Return type
namespace

read_spectrum

dustpylib.radtrans.radmc3d.**read_spectrum**(*path*)

This functions reads a spectrum file created by RADMC-3D and returns a dictionary with the SED data.

Parameters
path (*str*) – Path to the spectrum data file

Returns
d – Dictionary with the SED data

Return type
dict

Classes

Model(sim[, ignore_last])	Main model class that can read in DustPy models and can create RADMC-3D input files.

Model

class dustpylib.radtrans.radmc3d.**Model**(*sim*, *ignore_last=True*)

Bases: object

Main model class that can read in DustPy models and can create RADMC-3D input files. Attributes with trailing underscore are imported from DustPy, while the other attributes will be used to create RADMC-3D input files.

read_image :

Reads RADMC-3D image file

read_spectrum :

Reads RADMC_3d spectrum file

write_files :

Writes all required RADMC-3D input files into the specified directory

write_opacity_files :

Writes only the required RADMC-3D opacity into files into the specified directory

Attributes

ac_grid
Particle size bin centers in cm for RADMC-3D model.

ai_grid
Particle size bin interfaces in cm for RADMC-3D model

phic_grid
Azimuthal grid cell centers in rad for RADMC-3D model.

phii_grid
Azimuthal grid cell interfaces in rad for RADMC-3D model.

rc_grid
Radial grid cell centers in cm for RADMC-3D model.

ri_grid
Radial grid cell interfaces in cm for RADMC-3D model.

thetac_grid
Polar grid cell centers in rad for RADMC-3D model.

thetai_grid
Polar grid cell interfaces in rad for RADMC-3D model.

Methods

write_files([datadir, write_opacities, ...])	Function writes all required RADMC-3D input files.
write_opacity_files([datadir, opacity, ...])	Function writes the required opacity files.

Attributes Summary

ac_grid	Particle size bin centers in cm for RADMC-3D model.
ai_grid	Particle size bin interfaces in cm for RADMC-3D model
phic_grid	Azimuthal grid cell centers in rad for RADMC-3D model.
phii_grid	Azimuthal grid cell interfaces in rad for RADMC-3D model.
rc_grid	Radial grid cell centers in cm for RADMC-3D model.
ri_grid	Radial grid cell interfaces in cm for RADMC-3D model.
thetac_grid	Polar grid cell centers in rad for RADMC-3D model.
thetai_grid	Polar grid cell interfaces in rad for RADMC-3D model.

Methods Summary

write_files([datadir, write_opacities, ...])	Function writes all required RADMC-3D input files.
write_opacity_files([datadir, opacity, ...])	Function writes the required opacity files.

Attributes Documentation

ac_grid

Particle size bin centers in cm for RADMC-3D model. Do not set manually. Only use size bin interfaces.

ai_grid

Particle size bin interfaces in cm for RADMC-3D model

phic_grid

Azimuthal grid cell centers in rad for RADMC-3D model. Do not set manually. Only use cell interfaces.

phii_grid

Azimuthal grid cell interfaces in rad for RADMC-3D model.

rc_grid

Radial grid cell centers in cm for RADMC-3D model. Do not set manually. Only use cell interfaces.

ri_grid

Radial grid cell interfaces in cm for RADMC-3D model.

thetac_grid

Polar grid cell centers in rad for RADMC-3D model. Do not set manually. Only use cell interfaces.

thetai_grid

Polar grid cell interfaces in rad for RADMC-3D model.

Methods Documentation

write_files(*datadir=None*, *write_opacities=True*, *opacity=None*, *smooth_opacities=False*)

Function writes all required RADMC-3D input files.

Parameters

- **datadir** (*str, optional, default: None*) – Data directory in which the files are written. None defaults to the datadir attribute of the parent class.
- **write_opacities** (*booelan, optional, default: True*) – If False, does not compute nor write opacity files.
- **opacity** (*str, optional, default: None*) – Opacity model to be used. Either 'birnstiel2018' or 'ricci2010'. None defaults to 'birnstiel2018'.
- **smooth_opacities** (*bool, optional, default: False*) – Smooth the opacities by averaging over multiple particle sizes. This slows down the computation.

write_opacity_files(*datadir=None*, *opacity=None*, *smooth_opacities=False*)

Function writes the required opacity files.

Parameters

- **datadir** (*str, optional, default: None*) – Data directory in which the files are written. None defaults to the datadir attribute of the parent class.
- **opacity** (*str, optional, default: None*) – Opacity model to be used. Either 'birnstiel2018' or 'ricci2010'. None defaults to 'birnstiel2018'.
- **smooth_opacities** (*bool, optional, default: False*) – Smooth the opacities by averaging over multiple particle sizes. This slows down the computation.

H_dust_

Dust scale heights array in cm from DustPy

M_star_

Stellar mass in g

R_star_

Stellar radius in cm

Sigma_dust_

Dust surface density profile in g/cm^2 from DustPy

T_gas_

Temperature profile in K from DustPy

T_star_

Stellar effective temperature in K

a_dust_

Particle size array in cm from DustPy

property ac_grid

Particle size bin centers in cm for RADMC-3D model. Do not set manually. Only use size bin interfaces.

property ai_grid

Particle size bin interfaces in cm for RADMC-3D model

datadir

Directory to store the RADMC-3D input files

lam_grid

Wavelength grid for RADMC-3D in cm

opacity

Opacity model. “birnstiel2018” (default) or “ricci2010”

property phic_grid

Azimuthal grid cell centers in rad for RADMC-3D model. Do not set manually. Only use cell interfaces.

property phii_grid

Azimuthal grid cell interfaces in rad for RADMC-3D model.

radmc3d_options

RADMC-3D options for radmc3d.inp file

property rc_grid

Radial grid cell centers in cm for RADMC-3D model. Do not set manually. Only use cell interfaces.

rc_grid_

Radial grid cell centers from DustPy model

property ri_grid

Radial grid cell interfaces in cm for RADMC-3D model.

ri_grid_

Radial grid cell interfaces from DustPy model

property thetac_grid

Polar grid cell centers in rad for RADMC-3D model. Do not set manually. Only use cell interfaces.

property thetai_grid

Polar grid cell interfaces in rad for RADMC-3D model.

write_files(*datadir=None*, *write_opacities=True*, *opacity=None*, *smooth_opacities=False*)

Function writes all required RADMC-3D input files.

Parameters

- **datadir** (*str, optional, default: None*) – Data directory in which the files are written. None defaults to the datadir attribute of the parent class.
- **write_opacities** (*booelan, optional, default: True*) – If False, does not compute nor write opacity files.
- **opacity** (*str, optional, default: None*) – Opacity model to be used. Either ‘birnstiel2018’ or ‘ricci2010’. None defaults to ‘birnstiel2018’.
- **smooth_opacities** (*bool, optional, default: False*) – Smooth the opacities by averaging over multiple particle sizes. This slows down the computation.

write_opacity_files(*datadir=None*, *opacity=None*, *smooth_opacities=False*)

Function writes the required opacity files.

Parameters

- **datadir** (*str, optional, default: None*) – Data directory in which the files are written. None defaults to the datadir attribute of the parent class.

- **opacity** (*str, optional, default: None*) – Opacity model to be used. Either 'birnstiel2018' or 'ricci2010'. None defaults to 'birnstiel2018'.
- **smooth_opacities** (*bool, optional, default: False*) – Smooth the opacities by averaging over multiple particle sizes. This slows down the computation.

1.7.10 dustpylib.substructures Package

This package contains modules to create substructures in `DustPy` simulations.

1.7.11 dustpylib.substructures.gaps Package

This package contains extensions to create gaps in the disk profile.

`dustpylib.substructures.gaps.`**duffell2020**(*r*, *a*, *q*, *h*, *alpha0*)

Function calculates the planetary gap profile according Duffell (2020).

Parameters

- **r** (*array-like, (Nr,)*) – Radial grid
- **a** (*float*) – Semi-major axis of planet
- **q** (*float*) – Planet-star mass ratio
- **h** (*float*) – Aspect ratio at planet location
- **alpha0** (*float*) – Unperturbed alpha viscosity parameter

Returns
f – Pertubation of surface density due to planet

Return type
array-like, (Nr,)

`dustpylib.substructures.gaps.`**kanagawa2017**(*r*, *a*, *q*, *h*, *alpha0*)

Function calculates the planetary gap profile according Kanagawa et al. (2017).

Parameters

- **r** (*array-like, (Nr,)*) – Radial grid
- **a** (*float*) – Semi-major axis of planet
- **q** (*float*) – Planet-star mass ratio
- **h** (*float*) – Aspect ratio at planet location
- **alpha0** (*float*) – Unperturbed alpha viscosity parameter

Returns
f – Pertubation of surface density due to planet

Return type
array-like, (Nr,)

Functions

duffell2020(r, a, q, h, alpha0)	Function calculates the planetary gap profile according Duffell (2020).
kanagawa2017(r, a, q, h, alpha0)	Function calculates the planetary gap profile according Kanagawa et al. (2017).

duffell2020

dustpylib.substructures.gaps.**duffell2020**(*r*, *a*, *q*, *h*, *alpha0*)

Function calculates the planetary gap profile according Duffell (2020).

Parameters

- **r** (*array-like, (Nr,)*) – Radial grid
- **a** (*float*) – Semi-major axis of planet
- **q** (*float*) – Planet-star mass ratio
- **h** (*float*) – Aspect ratio at planet location
- **alpha0** (*float*) – Unperturbed alpha viscosity parameter

Returns

f – Pertubation of surface density due to planet

Return type

array-like, (Nr,)

kanagawa2017

dustpylib.substructures.gaps.**kanagawa2017**(*r*, *a*, *q*, *h*, *alpha0*)

Function calculates the planetary gap profile according Kanagawa et al. (2017).

Parameters

- **r** (*array-like, (Nr,)*) – Radial grid
- **a** (*float*) – Semi-major axis of planet
- **q** (*float*) – Planet-star mass ratio
- **h** (*float*) – Aspect ratio at planet location
- **alpha0** (*float*) – Unperturbed alpha viscosity parameter

Returns

f – Pertubation of surface density due to planet

Return type

array-like, (Nr,)

DustPyLib is a community project. New modules are welcome and can be added via pull requests.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

dustpylib, 49
dustpylib.dynamics, 49
dustpylib.dynamics.backreaction, 49
dustpylib.grid, 51
dustpylib.grid.refinement, 51
dustpylib.planetesimals, 52
dustpylib.planetesimals.formation, 52
dustpylib.radtrans, 55
dustpylib.radtrans.radmc3d, 55
dustpylib.substructures, 64
dustpylib.substructures.gaps, 64

INDEX

A

a_dust_ (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 62
ac_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 61
ac_grid (*dustpylib.radtrans.radmc3d.Model property*), 56, 62
ai_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 61
ai_grid (*dustpylib.radtrans.radmc3d.Model property*), 56, 62

B

BackreactionCoefficients() (*in module dustpylib.dynamics.backreaction*), 49, 50
BackreactionCoefficients_VerticalStructure() (*in module dustpylib.dynamics.backreaction*), 49, 50

D

datadir (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 62
drazkowska2016() (*in module dustpylib.planetesimals.formation*), 52, 54
duffell2020() (*in module dustpylib.substructures.gaps*), 64, 65
dustDiffusivity_Backreaction() (*in module dustpylib.dynamics.backreaction*), 49, 51
dustpylib
    module, 49
dustpylib.dynamics
    module, 49
dustpylib.dynamics.backreaction
    module, 49
dustpylib.grid
    module, 51
dustpylib.grid.refinement
    module, 51
dustpylib.planetesimals
    module, 52
dustpylib.planetesimals.formation
    module, 52
dustpylib.radtrans
    module, 55
dustpylib.radtrans.radmc3d
    module, 55
dustpylib.substructures
    module, 64
dustpylib.substructures.gaps
    module, 64

H

H_dust_ (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 62

K

kanagawa2017() (*in module dustpylib.substructures.gaps*), 64, 65

L

lam_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 63

M

M_star_ (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 62
miller2021() (*in module dustpylib.planetesimals.formation*), 52, 54
Model (*class in dustpylib.radtrans.radmc3d*), 55, 60
module
    dustpylib, 49
    dustpylib.dynamics, 49
    dustpylib.dynamics.backreaction, 49
    dustpylib.grid, 51
    dustpylib.grid.refinement, 51
    dustpylib.planetesimals, 52
    dustpylib.planetesimals.formation, 52
    dustpylib.radtrans, 55
    dustpylib.radtrans.radmc3d, 55
    dustpylib.substructures, 64
    dustpylib.substructures.gaps, 64

O

opacity (*dustpylib.radtrans.radmc3d.Model attribute*), 57, 63

P

phic_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 61
phic_grid (*dustpylib.radtrans.radmc3d.Model property*), 57, 63
phii_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 61
phii_grid (*dustpylib.radtrans.radmc3d.Model property*), 57, 63

R

R_star_ (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 62
radmc3d_options (*dustpylib.radtrans.radmc3d.Model attribute*), 57, 63
rc_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 61
rc_grid (*dustpylib.radtrans.radmc3d.Model property*), 57, 63
rc_grid_ (*dustpylib.radtrans.radmc3d.Model attribute*), 57, 63
read_image() (*in module dustpylib.radtrans.radmc3d*), 58, 59
read_model() (*in module dustpylib.radtrans.radmc3d*), 58, 59
read_spectrum() (*in module dustpylib.radtrans.radmc3d*), 58, 59
refine_radial_local() (*in module dustpylib.grid.refinement*), 51, 52
ri_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 61
ri_grid (*dustpylib.radtrans.radmc3d.Model property*), 57, 63
ri_grid_ (*dustpylib.radtrans.radmc3d.Model attribute*), 57, 63

S

schoonenberg2018() (*in module dustpylib.planetesimals.formation*), 53, 55
setup_backreaction() (*in module dustpylib.dynamics.backreaction*), 50, 51
Sigma_dust_ (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 62

T

T_gas_ (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 62
T_star_ (*dustpylib.radtrans.radmc3d.Model attribute*), 56, 62
thetac_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 61
thetac_grid (*dustpylib.radtrans.radmc3d.Model property*), 57, 63
thetai_grid (*dustpylib.radtrans.radmc3d.Model attribute*), 61
thetai_grid (*dustpylib.radtrans.radmc3d.Model property*), 57, 63

V

vrad_dust_BackreactionVerticalStructure() (*in module dustpylib.dynamics.backreaction*), 51

W

write_files() (*dustpylib.radtrans.radmc3d.Model method*), 57, 62, 63
write_opacity_files() (*dustpylib.radtrans.radmc3d.Model method*), 57, 62, 63