Contents

List of Figures

List of Tables

Chapter 1

Introduction

Magrathea is a tool for simulating and fitting millimeter wavelength emission from protoplanetary disks. This involves two largely separate sections: the first handles modeling of emission, and the second Markov Chain Monte Carlo fitting of those models to data. This tutorial will provide information on compiling and running the Magrathea code, and include examples highlighting the most common use cases. While some information on the algorithms used will be provided in the following chapters, detailed explanations of which methods were chosen and why can be found in my thesis, which can be found on the Rice University Library website. [1]

The Magrathea source code is in the form of libraries. Users can include its various sections to write C++ programs of their own for fitting and modeling disk emission.

This document is split into three parts. Chapter 1 provides introductory information, including system requirements and compilation instructions. Chapter 2 will describe in detail how to make model disk images, of both continuum and multi-channel line emission. Chapter 3 will explain how to set up and run a full MCMC fit.

The documentation here assumes a basic level of knowledge about C++ language features. Users should not need to be experts, however, in order to use Magrathea for basic modeling and fitting.

1.1 System Requirements

1.1.1 Operating System

Magrathea is designed to run on most recent Linux and Unix systems. I have personally verified functionality on several Linux systems, including Ubuntu (version 16 and 20), Mint (version 20, based off of Ubuntu 20), CentOS 8, RHEL 6, and RHEL 7. Several MacOS versions have been tested as well, including versions from 10.12 to 10.15. In principle, Magrathea should be able to run on basically any Unix system that you are capable of getting a working C++17 capable compiler on.

[1] I'll include a link here once I've submitted the final version and they've put it up.

It is also possible to use Magrathea from Windows, using the Windows Subsystem for Linux (WSL). This requires updating the operating system to a preview build of Windows 10, and then installing the newer WSL 2.[1] Once installed, Magrathea will run in the Linux VM just like on any other Linux system, though I did notice some intermittent connection issues when connecting in to large fits from Windows.

1.1.2 Hardware Requirements

There are few explicit requirements on the hardware needed. Both Intel and AMD systems work fine (you could probably even get an ARM system working, but I never needed to). However, for full functionality, it is recommended to use a newer processor.

Magrathea will attempt to use the wider SIMD instructions provided by newer systems in order to speed up calculation. Currently, the most recent recent instruction set used is AVX2, which is supported by most Intel and AMD processors made after 2015. Older systems will fall back on older instruction sets when possible, such as SSE4.2, and will use the basic X86 instructions if necessary.[2] It is never necessary to use the vector extensions, but can improve speed by a factor of two to three.

1.1.3 External Libraries

Besides the source code, Magrathea requires several external libraries in order to run properly. For details of what each of these libraries contributes, see the relevant sections of my thesis.

External Dependencies

Code	Source	Purpose
Boost	www.boost.org	Various C++ Libraries
Cap'n Proto	capnproto.org	Serialization
CFITSIO	heasarc.gsfc.nasa.gov/fitsio/	FITS File Handling
FFTW	www.fftw.org	Fast Fourier Transform
libcuckoo	github.com/efficient/libcuckoo	Hashing
Threadpool	github.com/progschj/ThreadPool/	Multithreading
VCL	github.com/vectorclass	Vectorization

Table 1.1: External libraries and their uses for Magrathea.

[1] For details on WSL installation, see https://docs.microsoft.com/en-us/windows/wsl/install-win10.

[2] There is a known issue in how Agner Fog's Vectorclass library handles decomposing certain math functions to smaller register widths. It can be necessary to unpack certain operations, such as exponentials, in order to make them run on very old systems. This seems to be a bug in the vectorclass library, but will likely not come up on any computer made post 2012.

Boost

Boost provides a wide variety of extensions to the C++ standard libraries, from mathematical algorithms to networking. Several Boost libraries have been added as official language features, with more to be included in the future.

This thesis makes use of the Boost HTTP server, which provides a simple way to send newly generated temperature files from the server to worker nodes. Worker nodes can request the temperature files from the HTTP server via libcurl.

Cap'n Proto

Cap'n Proto provides serialization and remote process control for C++ programs, written and maintained by Kenton Varda. Here, Cap'n Proto is used to send commands and information between the server and worker nodes. The server sends work units to worker nodes, as well as commands to wait or shut down, as necessary. The worker nodes send connection notifications to the central server, as well as work requests and periodic "heartbeat" signals.

CFITSIO

CFITSIO provides C libraries for reading and writing to FITS files **cite Penc1999**, one of the primary file formats used in astronomy. Internally, CFITSIO is used to read in observations which come as FITS files, and to output models as FITS files for viewing.

FFTW

FFTW provides C libraries for extremely rapid Fast Fourier Transforms, written by Matteo Frigo and Steven G. Johnson. FFTs are used primarily for calculating the goodness of fit of model images, but can also be used to generate beam convolved models.

libcuckoo

libcuckoo provides an efficient, compact, and thread safe hash table **cite Fan2013**. Internally, this is used to generate unique worker IDs when each worker node is started.

Threadpool

The Threadpool library is a C++ library written by Jakob Progsch, which simplifies multithreading in C++. The threadpool allows the programmer to spawn a threadpool with a given number of threads, which are handled internally, so that the programmer does not need to work directly with UNIX threads. This is used internally in every place where a task has been multithreaded, most importantly in generating model images, but also calculating χ^2 values.

VCL

VCL, Agner Fog's vector class, provides simple C++ classes to handle vectorized programs. The vector class provides convenient wrappers around the system provided vector intrinsics, making it much easier to write simple C++ code which takes full advantage of SIMD instructions. Internally, vectorized instructions are used extensively to optimize the ray tracing, and to calculate the goodness of fit.

1.1.4 Docker

For those attempting truly ambitious fits, it may be necessary to shift calculation from local computing resources to larger, cloud based systems. Most of these, such as those operated by AWS, are most easily used via containers, such as Docker. A Docker image contains an operating system, typically a small Linux distribution, as well as the libraries required to build the code. I'll include a link here to the Docker image I've made for Magrathea once it is more finalized.

1.2 Installation

In order to use Magrathea, it is necessary to first necessary to compile the source libraries. This is done in the typical, if old fashioned, way using `make`:

```
./configure.sh --prefix=[install location]
make
make install
```

To set up, first run the provided configure shell script, which will check for the required external libraries and compiler features. To set the install location to something other than the default directory, use the `--prefix` flag. The configure script will check the usual locations for libraries and compilers, but can be directed to look for specific libraries in specific directories using the `--using-[library]` option. For a complete list of options, use the `--help` or `-h` flag.

If successful, then run `make`, to build the source code to the local directory. Lastly, running `make install` will copy the files to the specified install directory.

1.3 Frequently Asked Questions

Actually, none of these are frequently asked questions, since this software and manual are brand new. However, I suspect some of these might become frequently asked, in the event that anyone ever tries to use Magrathea.

Where can I submit bug reports?

Send bugs and other issues to etweaver@gmail.com. I can't promise to rapidly fix anything, since it will no longer be my job to maintain this code in the future, but I

will try to.

Will the code ever include scattering?

Nope. The entire system is structured on the assumption that radiative transfer can be treated only along lines of sight. Adding scattering would lose that, and with it, most of the multithreading and vectorization. If you need scattering in your models, the tool you need is RADMC 3d, which can be found at ita.uni-heidelberg.de/∼dullemond/software/radmc-3d/.

Why "Magrathea"?

Magrathea was a planet in *The Hitchhiker's Guide to the Galaxy*, where they made planets for people. This is code for modeling planet formation, and it needed a good name, so he you go.

Why is this manual formatted like a thesis?

Because I just finished writing the thesis, and the thesis template is what I had handy.

Chapter 2

Modeling

This chapter will explain in detail how one goes about setting up a program to model a disk. Examples are included for several simple cases, for both continuum and line emission. Each example will output a FITS file containing the resulting model.

All units in Magrathea are assumed to be in cgs. Not everyone is happy about this, but it is the standard for most astronomy, and is used here.

2.1 Setup

This section will go into detail about how one sets up a model disk and makes an image of it. Before getting into examples, it is necessary to include the library, and to consider the primary data structures, the Grid, and the Image.

2.1.1 Compilation

Before we can get into the details of running a model, it is first necessary to set up the program correctly. Because Magrathea is a typical library, this has two parts. First, the correct files need to be included in any of the C++ files where Magrathea is used, and second, the compiler must be told to include the library.

Magrathea is small enough that it is entirely included in one umbrella file, magrathea.h. To use it, simply include it as any other header file by adding

```
#include "magrathea.h"
```

This gives access to all of the modeling classes and functions, as well as various physical constants.

In order to compile, the correct linker command must be added. If Magrathea has been installed in a default location, simply adding `-lmagrathea` will include it. If it is installed to a custom location not included in the search path, add `-L/path/to/magrathea` as well.

2.1.2 Basic Structures: The Grid

With the libraries included, we now have access to the Magrathea Grid class. The purpose of the Grid class is to contain the physical data of the disk, including its temperature and density structures, and to provide and interface for radiative transfer. The Grid is one of the largest and most complex parts of Magrathea, as it connects the structures of the disk to the physical processes involved, and it has many parameters.

The basic constructor for initializing a Grid for continuum imaging is

```
grid(double rmin, double rmax, double tmin, double tmax,
    double starMass, shared_ptr<density_base> dustDens,
    shared_ptr<temperature_base> diskTemp,
    shared_ptr<opacity_base> dustOpac):
```

The parameters fall into two basic groups: First, the dimensions of the Grid, as well as the star mass. Next come the user defined structures for the density, temperature, and the opacity. Don't be intimidated by the `std::shared_ptr`s! This is just a slightly fancier way of the old C style of passing pointers to the objects, which ensures that they are properly deleted, and keeps you from leaking memory everywhere.

Extent

First, the physical extent. The Grid structure is designed around modeling protoplanetary disks, and has a shape which reflects this. There are inner and outer radial boundaries, which are defined spherically, and upper and lower azimuth boundaries, defined by cones. [1] This way, the region around the central star can be excluded, as it is typically very hot and has different physics.

It is important to understand that the inner and outer boundaries of the Grid are not necessarily connected to the physical structure of the disk material being modeled. Why? Because in order to do ray tracing, the rays must have a start and stop location, and it's easiest to this by seeing where they intersect the Grid. The structure meanwhile, is often defined in a continuous way. But because most emission comes from the inner regions, and near the midplane, it is reasonable to terminate the disk once there is little emission.

Some care must be taken to define the Grid boundaries in a sensible way that includes all of the important disk emission. The simple heuristics I tend to use are based around the power law disk with exponential cutoff, which is the basic disk model from the similarity solution. Because the exponential cutoff terminates extremely sharply, it isn't necessary to expand the Grid much beyond this. For a disk with a cutoff at 100 au, I might conservatively put the Grid boundary at 150 au, but if really pressed for speed, might contract it to 120 au. Vertically, you don't need to include

[1] Technically, there are also pmin and pmax variables you can use to set the azimuth boundaries. Personally, I can't think of any reason you would ever want these not to be 0 and 2π, so they default to these. However, if you always dreamed of cutting a slice out of your disk like a pie, this is your chance.

much beyond a few scale heights, since the vertical structure is typically Gaussian. For a flared disk, you could calculate the opening angle that fits the vertical extent at the radial cutoff, but for flat disks this is usually overkill. I tend to just assume an opening angle of 30°, and go from there.

One other important not is that the angles, `tmin` and `tmax` start at 0 being the pole and π being the opposite pole, they are not defined relative to the midplane. So, to specify a symmetric, 30°opening angle, `tmin` would be $5\pi/12$, and `tmax` would be $7\pi/12$.

Star Mass

This, fairly obviously, is the star mass, in grams. For continuum modeling, this has very little effect, but is included for completeness. The star mass will matter much more when we talk about modeling line emission.

2.1.3 Input Structures

In Magrathea, the disk can be quite general. The way this is done is by allowing the user to define their own temperatures, densities, and opacities. If you want a disk where the density structure is square, the temperature a spiral and the opacity dependent on radius, you can do that. However, because there are several very common types of disk, I provide a variety of common density, temperature, and opacity types.

For those wanting their own, all that is required is to define a class with the correct interface and pass a shared pointer into the grid. These user defined classes must inherit from the correct base class, either `density_base`, `temperature_base`, or `opacity_base`.

In this section, I'll describe some of the built in structures available, as well as how to define your own.

Physical Structures

The most complicated section is how to specify the physical structures in the disk. In this simple case, two are needed, a single density and a temperature. For models with line emission, a second, separate, gas density can be specified. That will be detailed in the next section.

Dust structure, in particular, can vary considerably, depending on if you want gaps, rings, spirals, or other complicated features. The best way to do this is to define your own density structure. Your user defined density can be whatever you like, make the disk a hexagon if you want. The only requirement is that it have an overloaded call operator, which takes three spherical coordinates as arguments. This is how the integrator will expect to get the densities of positions in the disk. Here's an example for a classic power law disk with exponential cutoff:

```
struct myDensity : public density_base {
    double Sigma0;
    double rc;
    double h0;
    double P;//density index
    double S;//scale height index

    myDensity():
        Sigma0(0), rc(0), h0(0), P(0), S(0){ }
    myDensity(const newDensity& other):
        Sigma0(other.Sigma0), rc(other.rc), h0(other.h0),
        P(other.P), S(other.S){  }
    myDensity(double Sigma0, double rc, double h0,
    double P, double S):
        Sigma0(Sigma0), rc(rc), h0(h0), P(P), S(S){ }

    myDensity& operator= (const myDensity& other){
        Sigma0=other.Sigma0; rc=other.rc;
        h0=other.h0; P=other.P; S=other.S;
        return *this;
    }

    double surfaceMassDensity(const double r) const{
        return (Sigma0*pow(r/rc, -P)) * exp(-(pow(r/rc, 2-P)));
    }

    double scaleHeight(const double r) const{
        return (h0*pow(r/rc,S));
    }

    double operator()(double r, double theta, double phi) const{
        double r_cyl=r*sin(theta);
        double z=r*cos(theta);
        double h=scaleHeight(r_cyl);

        return(((surfaceMassDensity(r_cyl))/(sqrt(2*pi)*h)) *
            exp(-z*z/(2*h*h)));
    }
};
```

This provides the basic power law disk structure described in Chapter 2 of my thesis:

$$\rho(r,z) = \frac{\Sigma(r)}{\sqrt{2\pi}h(r)} exp\left(\frac{-z^2}{2h^2(r)}\right) \tag{2.1}$$

with

$$\Sigma(r) = \Sigma_0 \times \left(\frac{r}{r_0}\right)^{-p}, \tag{2.2}$$

and where Σ_0 is the surface density normalization at the reference radius r_0, and $h_d(r)$ is the dust scale height, parameterized as

$$h_d(r) = h_0 \left(\frac{r}{r_0}\right)^{S} \tag{2.3}$$

with r_0 and h_0 being the radius normalization and corresponding scale height.

This basic power law disk is included in Magrathea as a default. I'll cover all of the supplied structures in section 2.4. I've used several adaptations of this basic model over the years. The DSHARP based models had additional parameters describing Gaussian gaps which got subtracted out, for example.

On last note on setting up a density: The Grid constructor expects an `shared_ptr<density_base` These can be made using the `make_shared` function, which uses the object constructor. For example:

```
shared_ptr<myDensity> dptr=
    make_shared<myDensity>(1,100*AU,5*AU,1.0,1.25);
```

will make a shared pointer to a power law density.

Additional Notes on User Defined Densities

For continuum modeling purposes, the only requirement on a density is that it inherit from `density_base` and provide an overloaded call operator which takes three spherical coordinates. There are two additional virtual functions which can be overloaded optionally. One `scaleHeight()`, takes a radius value and returns the scale height of the disk at that radius. This is used with an optional ray tracing feature we'll discuss in Section 2.2. There is also a `colDensityAbove()` function, which is used primarily for handling freezeout of molecules, and takes three spherical coordinates. Neither of these functions is needed for basic continuum modeling, though providing a `scaleHeight()` can be useful.

Temperature Structures

The temperature structure operates the same way as the density. A user defined temperature inherits from `temperature_base`, and provides an identical overloaded call operator. For simple models, it is easy enough to define a parametric temperature which is similar in structure to the density above. However, for more complicated cases, it is often necessary to use a temperature model produced by an external code, such as RADMC. These must be read in from a file. The next example shows how to read in from temperature models based on the output of RADMC 3d modeling.

I say based on because this doesn't take a raw `.dat` file from RADMC.[1] RADMC works on a user defined grid, while the ray tracing here steps along the rays, but not in a neat gridded way. That means that to use a grid defined temperature (or density) profile, it's necessary to interpolate. To do that, I provide an interpolation surface, called `interpSurf`, which is designed to work with two dimensional RADMC output.

Internally, the `interpSurf` contains two arrays for the dimensions, i.e. a list of center points in the r and theta directions, and a large set of data. A data point consists of the spherical coordinates, and the value. The `interpSurf` class provides a constructor which takes two `std::ifstream`s, the grid file, and the temperature file:

```
interpSurface(std::istream& gridFile, std::istream& dataFile)
```

This assumes the input files are formatted as follows: The grid file is the standard RADMC `amr_grid.inp`, as described at www.ita.uni-heidelberg.de/~dullemond/software/radmc-3d/manual_radmc3d/inputoutputfiles.html The temperature file is different than the basic RADMC defined output, and assumes four columns, theta, phi, r, and temperature. The `interpSurf` class also provides the `interp2d` function, which takes two coordinates, r and theta, and returns the value at that point:

```
double interp2d(double r, double theta)
```

Using the `interpSurf`, we can now define a simple, table based temperature, with the same interface as the density struct above:

```
struct tableTemp: public temperature_base{
    interpSurface tempSurf;
    tableTemp():tempSurf(){}
    tableTemp(const advancedTemp& other):tempSurf(other.tempSurf){}

    tableTemp(std::ifstream& gridFile, std::ifstream& dataFile):
        tempSurf(gridFile, dataFile) {}

    double operator()(double r, double theta, double phi) const{
        return(tempSurf.interp2d(r,theta));
    }
};
```

I've included these example density and temperature structures because they are simple cases of the ones that I used most frequently over the last few years. Most projects generally only needed some modifications to one or the other, such as adding gaps or a spiral to the density. The `interpSurf` class, is, unfortunately, pretty janky. If there's sufficient interest, I can expand it to handle three dimensional structures, etc.

[1] I originally wrote this to work on processed files because I needed to be able to plot them easily. If you want to work directly with the RADMC output, you can adapt the input code to do so. If not, I'll provide my code that does this formatting in the appendix

Opacity Structures

In principle, opacities function just like temperatures and densities. Users are free to define their own, which inherit from the `opacity_base` class, and provide the correct overloaded call operators. In practice, the opacities tend to be either very simple, in the continuum case, or hideously complex, in the line emission case. Here, I mostly recommend using the ones I've already written.

For continuum opacity, I think most cases will simply rely on an external code. This means reading in from a file, for which I provide the `dustOpacity` struct, whose constructor requires only a string giving the path to a file. The continuum opacity is handled fairly differently from that of CO, which we'll look at later, because I don't think there are convenient functional approximations. Modeling dust opacity requires complicated codes, and is thus best handled by just using a table.

The formatting of the input file is the same as that used by RADMC: Two columns, wavelength in microns, and mass opacity, in cm^2/gm. For example

```
0.06500000  487.61199622
0.07150000  498.78685764
0.07860000  511.38581128
0.08650000  522.97715957
0.09120000  532.03370124
0.09510000  536.52678339
0.10500000  545.26333220
0.11500000  551.29091179
0.12700000  558.95778938
...         ...
```

If you are quite determined, you can implement your own dust opacity, which inherits from the `opacity_base` class. I assume that continuum opacity is just a function of frequency, so the main function you must provide is an overloaded call operator which only takes a double. However, if you want to use this opacity in a model with AVX propagation (which I'll discuss in Section 2.3), you will also need to provide a vectorized analogue which operates on a Vec4d of frequencies and returns a Vec4d of opacities.[1]

2.1.4 Basic Structures: The Image

Now that the Grid is set up, we can think about making images. The `image` class handles relative positions of the disk itself and the array of pixels that form the actual image. This involves, as one might guess, a lot of rather fiddly geometry, and if you break the Euler angle stuff, I'm not fixing it.

[1] Vectorization is done using Agner Fog's excellent Vectorclass library. This provides wrappers around the basic vectorized intrinsics provided by the operating system. The Vec4d is a 256 bit vector which holds four doubles. It's usually enough to define your own function that works on one frequency and then make a vectorized version that replaces the single frequency value with a Vec4d, but is still more advanced than the rest of what I discuss here.

The basic constructor for setting up and image is

```
image(unsigned int vpix, unsigned int hpix,double w, double h,
    std::vector<double> frequencies, astroParams& astroData);
```

The parameters are as follows: vpix and hpix, are the number of pixels, vertically and horizontally, for this image. The number of pixels is, ideally, set by the resolution of the observation you are modeling. Generally, you'd like the beam width to be something like 5 pixels. If you go much higher, you don't gain anything, since we can't resolve it. Unfortunately, for large fits, the time required for this is simply too high, so the resolution has to be reduced to save computation time. I usually ended up with images that were of order 1000 by 1000 pixels.

w and h are the width and height of the image in centimeters. It is important to remember here that the image itself has no field of view. Because the target is so far away, we are (thankfully) safe making the assumption of parallel rays. That means that you want the physical dimensions of the image to be roughly that of the target, plus some safety margins around the edge. For a disk with a major axis of 100 au, you might use a 200 au by 200 au image.

Next comes the frequency list. For continuum images, we typically just do single frequency models, but for line emission, this may be a large set. I'll discuss some of the intricacies of the frequency list when we do line emission later.

Lastly, the astronomical data. This is where we specify things like the Right Ascension, Declination, and other features that describe where the disk is and how it's oriented. The `astroParams` struct is simply a wrapper around this data with the following constructor:

```
astroParams(double RA, double Dec, double dist,
    double inc, double PA)
```

It takes the Right Ascension, Declination, distance to the target, and the inclination and position angle of the target.

Internally, the image class contains this information, and also a large array of pixels. When instantiated, this array is empty, we won't have the actual values until we call the `propagate` function, which I'll go into next.

The Image Data

Internally, the image is just a wrapper around a large array of numbers. This is the actual data, and can be treated as a three dimensional array, the pixel values in x and y, and also the frequency. The main way this data is filled in is by using the `propagate` function, but it is there for the user to work directly with if desired.

The way to access a pixel value is through the `data` struct included in an Image class. This struct provides subscript (that's the brackets) operators so that it can be handled like C arrays. For example:

```
image.data[0][200][100]
```

will return the pixel (100,200) for the first frequency. The order is always frequency, then row, then column, so `data[f][y][x]`. Using this, you can get or set pixel values easily.

2.1.5 Making a Model

To actually generate an image, the image class includes the `propagate` function, with the following signature:

```
void propagate(const grid& g, typename grid::prop_type type,
    const vect& offset, ThreadPool& pool, bool avx,
    unsigned int nsteps, double contractionFactor)
```

The first argument is simply the grid that will be imaged. Next comes the propagation type, which tells the system what to include in the ray tracing. There are several types available:

```
enum prop_type {normal, continuum, attenuated_continuum,
    subtracted_bad, subtracted_real};
```

Most cases will either be `normal`, which includes both continuum and line emission, or `continuum`, which has no line emission. The other cases are left over from my 2018 paper, and provide the various components of the temperature derivation calculations. `attenuated_continuum` does a continuum model, but includes absorption from line emission. `subtracted_bad` does improper continuum subtraction, which I complain about at great length in the paper, where `subtracted_real` handles it properly. It's unlikely that most users will need either of these last two, but they are included, just in case.

`offset` is a `vect`, a simple three vector which can be used to offset the center of the disk. If you don't want any offset, you can simply give it the vect (0,0,0).

Now, what's this `Threadpool`? The `Threadpool` is a handy piece of code written by Jakob Progsch, which I use to handle the multithreading. In the, actually quite ideal, case we have here, the different threads don't need to communicate, and there are no race conditions. We don't need anything as complicated as OpenMP or MPI, simple Unix threads will do. The `Threadpool` provides a handy system which sets up a user specified number of threads, which are used to divide up the work needed. You can set the number of threads to whatever you want, up to the maximum your system can support. An easy way to ask for that maximum, whatever it may be, is to use

```
const size_t nThreadsMax=std::thread::hardware_concurrency();
ThreadPool pool(nThreads);
```

This will initialize a `Threadpool` as large as possible.

Lastly, there are some settings for the ray tracing. `AVX` is a boolean stating whether or not to use the AVX enabled ray tracing. This is immensely useful for large models of line emission, but not generally for continuum. `nsteps` is simply the base number

of steps that each ray will take through the disk. You have a lot of leeway with the number of steps. In a perfect world, you would raise this until changing it no longer changed the results noticeably. That usually is unmanageable for most large projects, but I find that 100-300 steps usually gives results which are good enough.

And finally there's the contraction factor. This is a continuum specific bit of optimization based around the idea that the continuum emission is generally all optically thin, and therefore peaks along the disk midplane. Since we have the rare bit of information about where the emission comes from, we can do a sort of bootleg adaptive step size. The contraction factor simply shrinks the step size by the factor you specify when within 2.5 (dust) scale heights of the midplane. This way, you can have far fewer steps, but still get a reasonable result.

2.1.6 FITS File Handling

Astronomers often rely on FITS files to store or examine observations. Magrathea therefore provides a simple interface for handling fits files, both as inputs and outputs.

The most common case is outputting a model as a FITS file. This is done using the provided `printToFits()` function in the Image class, which takes a string representing the file name. `printToFits()` functions identically for single and multichannel images, with the important caveat that the basic ALMA format header assumes that channels are evenly spaced. If you do a multifrequency model with uneven channels, the output file will have an incorrect header.

This produces a FITS file, with a header generated based on the parameters of the image. The header style roughly matches that of a basic ALMA observation.

I've also provided ways of importing FITS files as images. This is designed around ALMA data, and so the files must have properly constructed headers, including all the CRPIX, CRVAL, and CDELT values. The `fitsExtract` function has two different signatures:

```
image fitsExtract(string filename, vector<double> frequencies)
```

and

```
image fitsExtract(string filename, vector<double> frequencies,
    double distance)
```

depending on whether or not you know the distance to the source. The frequency list is not simply inferred from the header for the reason I mentioned above, it may not be regular. The distance is important because the Image class has a defined physical extent, but an actual observation is only in terms of angle subtended on the sky. In order to properly generate a physically meaningful image, a distance is necessary. However, you can still upload the data and work with it even without this information.

2.2 Continuum Models

We now have all of the basic pieces in place to make continuum models. This section will include a few examples, starting with the very basic power law disk above, and going through some more complicated ones. I'll also show a couple of cases of common pitfalls and how to avoid them.

2.2.1 The Power Law Disk

This is the real workhorse of the disk modeling world. The power law disk is about as simple as you can get while still remaining physically motivated. Let's write an example which sets up a power law disk and makes an image. What follows is included in the examples folder as PowerLawDisk.cpp, in its entirety:[1]

```
#include "magrathea/magrathea.h"
using namespace std;
int main(int argc, char* argv[]){
    const size_t nThreadsMax=thread::hardware_concurrency();
    ThreadPool pool(nThreadsMax);
    astroParams diskData(246.6,-24.72,3.18e20,pi/4,pi/4);
    shared_ptr<powerLawDisk> dptr=make_shared<powerLawDisk>
        (1,100*AU,5*AU,1.0,1.25);
    ifstream gridfile("sampleData/amr_grid.inp");
    ifstream datafile("sampleData/dust_temperature_phi0.ascii");
    shared_ptr<fileTemp> tptr=make_shared<fileTemp>
        (gridfile,datafile);
    shared_ptr<dustOpacity> doptr=make_shared<dustOpacity>
        ("sampleData/dustopac.txt");

    grid g(0.1*AU,150*AU,75*pi/180,105*pi/180, 2.18*mSun,
        dptr, tptr, doptr);
    image img(500, 500,200*AU, 200*AU, {2.30538e11}, diskData);
    vect offset(0,0,0);

    img.propagate(g, grid::continuum, offset, pool, false, 100, 5);
    img.printToFits("PowerLawDisk.fits");

    return 0;
}
```

That's it! Let's look closely at a few of the lines. First we set up our Threadpool, to be as large as possible. Then we make an `astroParams`, located, if I remember correctly at DoAr 25's actual coordinates. It's put at a distance of 100pc, and at an inclination and position angle of $pi/4$. Then we set up the disk itself. Looking at

[1] With one change: I switched to using the standard namespace here so that the lines are shorter and a little clearer.

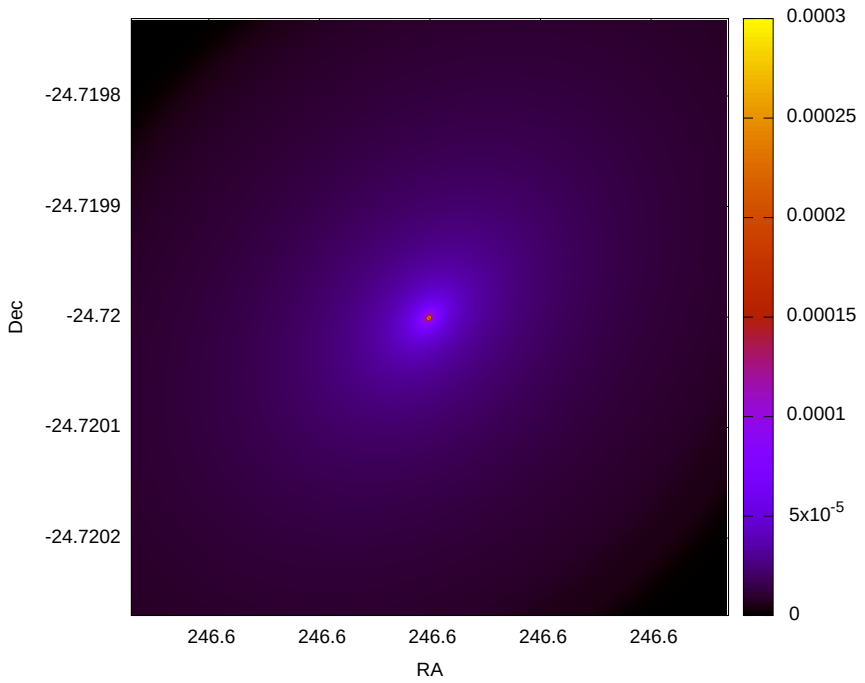

Figure 2.1: A linear scale plot of the power law disk model.

the constructor for `dptr`, we can see that the surface mass density is 1 gm/cm^2, at a reference distance of 100 au. The scale height at the reference distance is 5 au, and the power law indices are 1, radially, and 1.25 for the scale height. This is a pretty standard flared disk. The temperature is read in from a file, and then we set up the grid. The grid extends from 0.1 to 150 au, and has a 30°opening angle. The star has a mass of $2.18M_{\odot}$. Lastly, the image itself. The image is 500x500 pixels, and has a width and height of 200 au. We image at one frequency, 230.538 GHz. Finally, we make the image, with rays that take 100 steps each, but increased by a factor of 5 near the midplane. The resulting model is then printed to `PowerLawDisk.fits`.

You can compile this program with `clang++ PowerLawDisk.cpp -std=c++17 -lmagrathea -o PowerLawDisk`, and run it with `./PowerLawDisk`. On my laptop with 8 cores, it took 8.491 seconds to run. And the output is PowerLawDisk.fits, which looks like this:

There are a couple of things to note immediately about Figure 2.1. Most important are the units. Those who spend a lot of time looking at processed ALMA images may wonder about the color scale, given that this disk is located 100pc away. The important thing to remember is that there is still a very fundamental difference between this model and an ALMA image: There is no beam convolution here. Instead of the typical Janskys per beam, here we have Jy per pixel, which is typically a much smaller space than the beam size. Although the total flux is the same, to usefully compare point-by-point to an ALMA image, you will need to first convolve with a beam model. I'll discuss this at the end of the chapter.

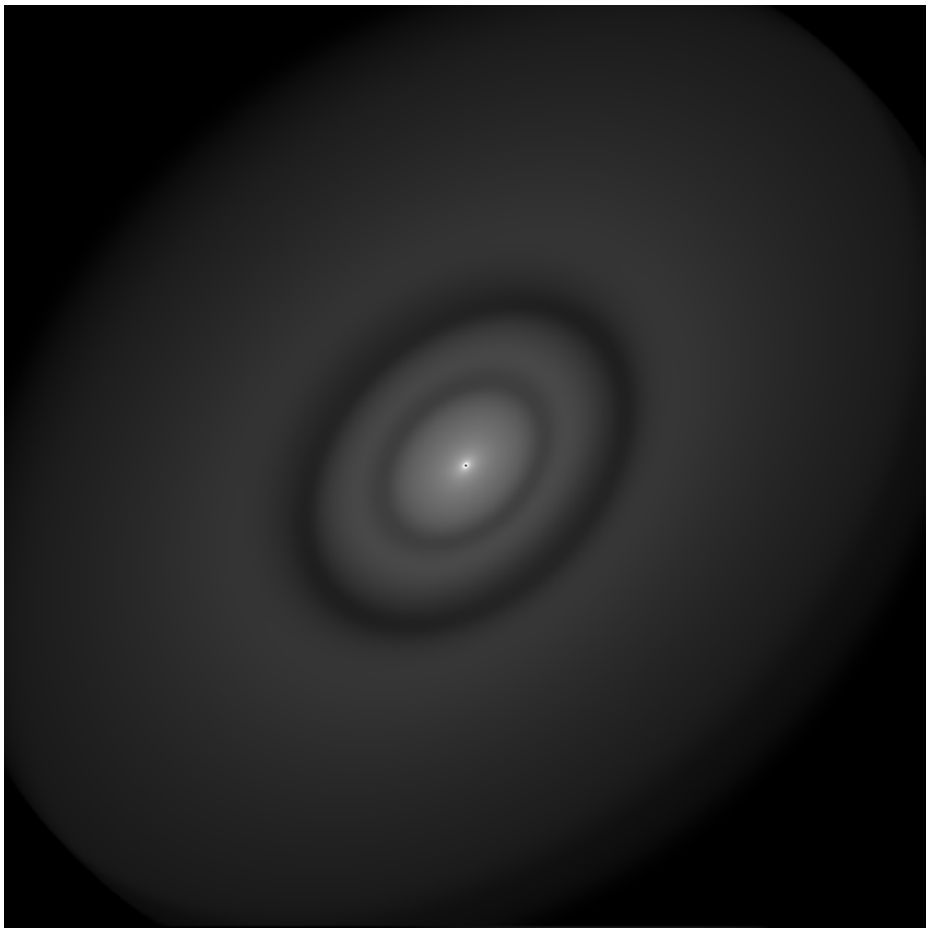

Figure 2.2: The gap disk model. I find an asinh color stretch makes these look good.

This sort of model often looks a little different from what you expect if you're only used to real observations. For one thing, there's no beam smearing, so we have resolution on the scale of the pixel size, which is usually much higher than you actually get. Sometimes this can lead to odd effects in the innermost region of the disk, where emission is much higher than anywhere else. If you convolve this with a realistic beam, that effect will largely go away. Similarly, the middlemost pixel will always go right through the inner cutoff of the grid if there's no offset, and have no emission. There is also no noise in these models, so everything will seem very smooth and perfectly symmetric. I haven't added any kind of noise, that turns out to be pretty complicated, but you can do it with CASA using the `simobserve` command.

2.2.2 Disk with Gaps

The basic power law disk is pretty boring to look at. In this section, I'll make a new density structure which adds gaussian gaps to the disk.

This is another common disk type, where we assume that gaps are carved out of

an otherwise ideal similarity solution disk. To do this, I tend to parameterize the gaps as azimuthally symmetric gaussians, where the parameter varied is "depth", or how much of the surface mass density is subtracted out. There isn't a whole lot of physical motivation to this model, it just happens to be a simple one which roughly fits a lot of gapped disks we observe. Since we have pretty poor understanding of the vertical structure of most disks, we tend to add these sorts of deviation by simply adjusting the surface mass density. Basically, we just change the midplane and assume that the vertical structure works as always.

Each gap needs three parameters, a central position, a depth, and a width. The surface mass density becomes

$$\tilde{\Sigma}(r) = \Sigma(r) \cdot \prod_i \left[1 - \Delta^i \cdot \exp\left(\frac{-(r - r^i)^2}{2\sigma^i}\right)\right], \tag{2.4}$$

where $\Sigma(r)$ is the basic power law disk, given in Equation 2.2. Because the gaps are applied as a removal factor, they are multiplicative, rather than additive. r^i indicates the position of the center of gap i, Δ^i its depth, and σ^i its width.

Equation 2.4 is more general than we really need most of the time, so for this example, I'll implement it for two gaps. The new density structure must include all 6 new gap variables, so the constructors are messier, but otherwise it works like before.

```
#include "magrathea/magrathea.h"
using namespace std;
struct gapDisk: public density_base {
    double Sigma0;
    double rc;
    double h0;
    double P;//density index
    double S;//scale height index
    double p1, p2; //ring positions (au)
    double d1, d2; //ring depths (0 to 1)
    double w1, w2; //ring widths (au)

    gapDisk(): Sigma0(0), rc(0), h0(0), P(0), S(0),
        p1(0), p2(0), d1(0), d2(0), w1(0), w2(0){ }
    gapDisk(const gapDisk& other): Sigma0(other.Sigma0),
        rc(other.rc), h0(other.h0),P(other.P), S(other.S),
        p1(other.p1), p2(other.p2), d1(other.d1), d2(other.d2),
        w1(other.w1), w2(other.w2) {  }
    gapDisk(double Sigma0, double rc, double h0, double P,
        double S, double p1, double p2,
        double d1, double d2, double w1, double w2):
        Sigma0(Sigma0), rc(rc), h0(h0), P(P), S(S),
        p1(p1), p2(p2), d1(d1), d2(d2),w1(w1), w2(w2){  }

    gapDisk& operator= (const gapDisk& other){
```

```
        Sigma0=other.Sigma0; rc=other.rc; h0=other.h0;
        P=other.P; S=other.S; p1=other.p1; p2=other.p2;
        d1=other.d1; d2=other.d2; w1=other.w1; w2=other.w2;
        return *this;
    }

    double surfaceMassDensity(const double r) const{
        return (Sigma0*pow(r/rc, -P)) * exp(-(pow(r/rc, 2-P)));
    }

    double scaleHeight(const double r) const{
        return (h0*pow(r/rc,S));
    }

    double operator()(double r, double theta, double phi) const{
        double r_cyl=r*sin(theta);
        double z=r*cos(theta);
        double h=scaleHeight(r_cyl);
        double gap1=(1-d1*(gaussianNotNorm(r_cyl,p1,w1)));
        double gap2=(1-d2*(gaussianNotNorm(r_cyl,p2,w2)));

        return(((surfaceMassDensity(r_cyl))/(sqrt(2*pi)*h)) *
            exp(-z*z/(2*h*h))*gap1*gap2);
    }
};

int main(int argc, char* argv[]){
    const size_t nThreadsMax=thread::hardware_concurrency();
    ThreadPool pool(nThreadsMax);
    astroParams diskData(246.6,-24.72,3.18e18*100,pi/4,pi/4);
    shared_ptr<gapDisk> dptr=std::make_shared<gapDisk>
        (1,100*AU,5*AU,0.5,1.25,25*AU,50*AU,0.8,0.9,5*AU,10*AU);
    ifstream gridfile("../amr_grid.inp");
    ifstream datafile("../dust_temperature_phi0.ascii");
    shared_ptr<advancedTemp> tptr=make_shared<advancedTemp>
        (gridfile,datafile);
    shared_ptr<dustOpacity> doptr=make_shared<dustOpacity>
        ("../dustopac.txt");

    grid g(0.1*AU,150*AU,75*pi/180,105*pi/180, 2.18*mSun,
        dptr, tptr, doptr);
    image img(500, 500,250*AU, 250*AU, {2.30538e11}, diskData);
    vect offset(0,0,0);

    img.propagate(g, grid::continuum, offset, pool, false, 100, 5);
    img.printToFits("GapDisk.fits");
```

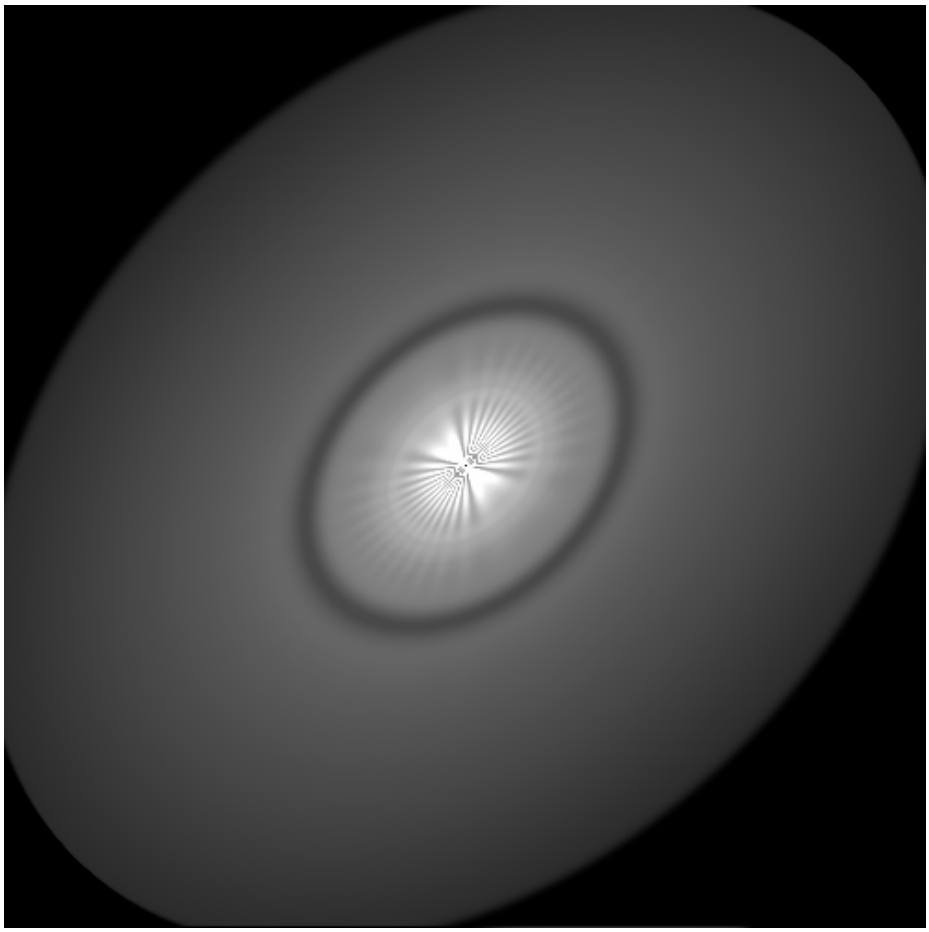

Figure 2.3: This model has too few steps for each line of sight, leading to the odd patterns in the center.

```
    return 0;
}
```

The output, `GapDisk.fits`, is shown in Figure 2.2 I was lazy here, and didn't bother to add axes or a color bar here. On my laptop, it took 7.765 seconds to generate this image. That's faster than the last one, but there are two effects here. One is that I increased the size of the image from 200x200 au to 250x250, so more pixels entirely miss the disk. On the other hand, we're doing a lot more math each time the ray tracer needs to look up a density.

2.2.3 Common Problems

In this section, I'll highlight a couple of common problems I've had when making models like this. Consider this first example: This is a similar model to the gap disk we just made, with a few changes. One is that I've made the vertical structure flatter,

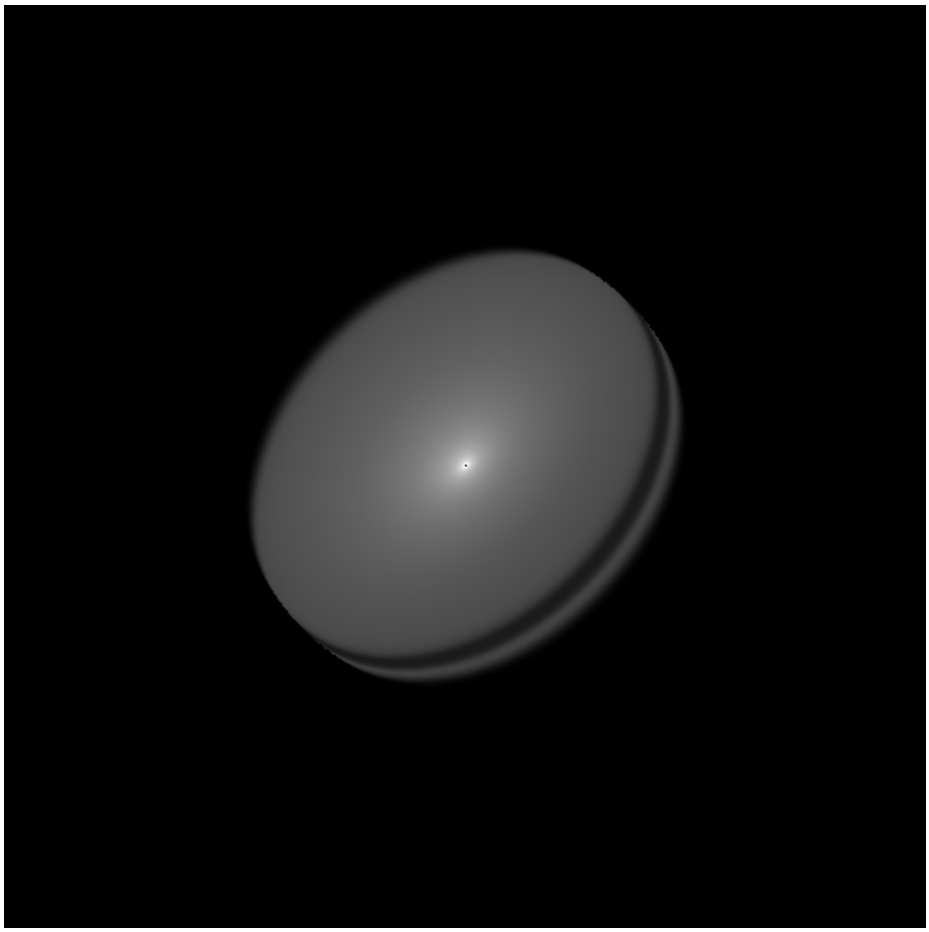

Figure 2.4: This model has been truncated at too low a radius, and we're looking into it from the side.

by a factor of five. The other is that I've reduced the number of steps through the disk by half, and turned off the midplane reduction. The result is that we are now taking far too few steps through the disk to adequately sample the emission.

We know that for a continuum model like this, the emission peaks on the midplane, but with so few steps, it makes a big difference how close the center step is too the midplane. For some lines of sight, it's right about on, and we measure a lot of emission. Some are farther off, and don't see any. This leads to the strange and curiously artistic patterning you see in the middle. Generally, any time you make art with Magrathea, it means something has gone wrong.

Now consider Figure 2.4. This effect occurs when the outer radius of the grid is set to be too short for the density and temperature structures. In this case, the reference point for the density is at 100 au, but the grid itself terminates at 50. The vertical effect you see is us looking into the disk from the side. Basically, we're looking at the vertical temperature structure here. If only we had disks like this in reality! Generally speaking, any time you see a disk that looks like an inverse Oreo cookie,

it's been truncated.

The continuum models tend to be much simpler, so there is less to go interestingly wrong like this. I often make mistakes that lead to there being no emission whatsoever, which can happen in a couple of ways. First, there could simply be an error in how the camera is pointed, and any time the image is all zero, you'll get a warning which includes the camera position and pointing. This also can mean something went wrong and that either the temperature, density, or opacity is zero. You can debug this easily by simply asking from the temperature or density at a random point that you know should have nonzero values.

2.3 Line Emission

Line emission works exactly the same way as the continuum, it simply has more components. For one thing, there can be separate density structures for the gas and the dust, and you must also supply an opacity for the line in question. The line opacities are complicated. You can see in my thesis what goes into rotational CO opacity, or, in the really ugly case, H_2O.

The more significant change is that line emission is frequency dependent in a way that the continuum isn't. Most of the time, you will want to generate multiple frequencies at a time. This means that the output is much larger, and the running time will be much longer.

2.4 Provided Disk Structures

Chapter 3

Fitting