# An Introduction to *dustmap*

Chris Stark

05/04/15
*First Draft*

## 1. Installing dustmap

1) Download the most recent version of dustmap from www.starkspace.com/code. Unzip and place the files into a directory. I recommend placing them in a directory called "dustmap/" which should be a subdirectory of where you keep all of your IDL routines. Now, we need to make sure that the dustmap subdirectory is in your IDL path. To do this, I recommend:

2) Create the file .idlstartupfile in your home directory and add the following line…

```
!PATH=!PATH+':'+Expand_Path('+/Users/username/Documents/IDL
routines/dustmap')
```

where the "/Users/username/Documents/Documents/IDL routines/dustmap" path should be replaced with the path to your own dustmap directory. Note that if you have a folder that contains all of your IDL routines like I do, you only need to point the path to that directory, not the dustmap subdirectory.

3) In your shell startup file (e.g. ".cshrc"), add the following line:

```
setenv IDL_STARTUP '/Users/username/.idlstartupfile'
```

4) dustmap is a hybrid IDL/C code. You must compile the C portion of the code. If you don't have a c compiler, I recommend getting gcc. Mac users can do this by downloading and installing the free XCode package.

I have made compiling *dustmap* relatively easy by including an IDL procedure "compilec.pro." This will compile the C portion of the code and optimize it for your system. To use this, CD to the dustmap subdirectory. Start IDL. Then type the following:

```
IDL> compilec,'dustmap_c'
```

***WARNING***: for some reason, some versions of IDL only compile C code **once per session**. Thus, before compiling a piece of C code, I always restart my IDL session.

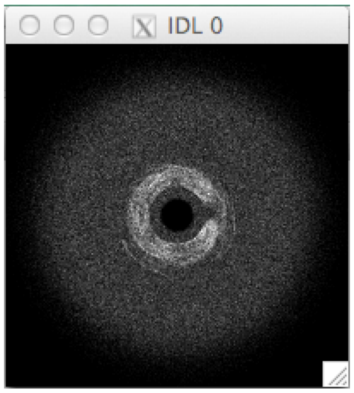
*NOTE*: the "compilec.pro" routine assumes you have gcc installed on your system. If you have another compiler, you will need to edit the contents of compilec.pro such that the correct compiler is called. It should be straightforward.

## 2. Testing dustmap

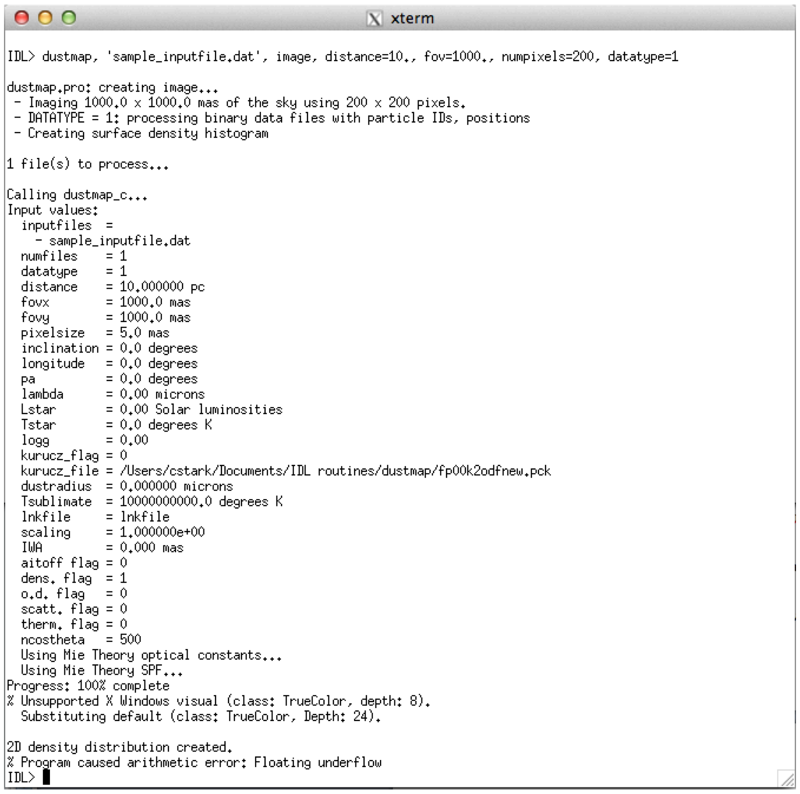
To make sure dustmap is working, CD to the dustmap subdirectory. Start IDL. Run the following command:

```
IDL> dustmap, 'sample_inputfile.dat', image, distance=10.,
fov=1000., numpixels=200, datatype=1
```

You should see an image that looks like the following



and you should see the following output:



```
IDL> dustmap, 'sample_inputfile.dat', image, distance=10., fov=1000., numpixels=200, datatype=1

dustmap.pro: creating image...
 - Imaging 1000.0 x 1000.0 mas of the sky using 200 x 200 pixels.
 - DATATYPE = 1: processing binary data files with particle IDs, positions
 - Creating surface density histogram

1 file(s) to process...

Calling dustmap_c...
Input values:
  inputfiles  =
    - sample_inputfile.dat
  numfiles    = 1
  datatype    = 1
  distance    = 10.000000 pc
  fovx        = 1000.0 mas
  fovy        = 1000.0 mas
  pixelsize   = 5.0 mas
  inclination = 0.0 degrees
  longitude   = 0.0 degrees
  pa          = 0.0 degrees
  lambda      = 0.00 microns
  Lstar       = 0.00 Solar luminosities
  Tstar       = 0.0 degrees K
  logg        = 0.00
  kurucz_flag = 0
  kurucz_file = /Users/cstark/Documents/IDL routines/dustmap/fp00k2odfnew.pck
  dustradius  = 0.000000 microns
  Tsublimate  = 10000000000.0 degrees K
  lnkfile     = lnkfile
  scaling     = 1.000000e+00
  IWA         = 0.000 mas
  aitoff flag = 0
  dens. flag  = 1
  o.d. flag   = 0
  scatt. flag = 0
  therm. flag = 0
  ncostheta   = 500
  Using Mie Theory optical constants...
  Using Mie Theory SPF...
Progress: 100% complete
% Unsupported X Windows visual (class: TrueColor, depth: 8).
  Substituting default (class: TrueColor, Depth: 24).

2D density distribution created.
% Program caused arithmetic error: Floating underflow
IDL> 
```

## 3. Using dustmap

*dustmap* creates histograms, geometric optical depth maps, and scattered light and thermal emission images. *dustmap* was originally intended to display the results of discrete particle (*n*-body) simulations. Thus, you must supply *dustmap* with a file that contains the discrete particle locations. Later we'll discuss the format of those files and how to create your own, but for now, let's just use the included sample data file. Let's run through a few different uses for dustmap.

### *3.1. Creating a dust particle histogram*

The default and most simple use of *dustmap*. Run the following:

```
IDL> file = 'sample_inputfile.dat'
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1
```

The above is the minimum required syntax for *dustmap*. You must supply a file name (string) that contains your discrete data points, a variable name that will contain the output image, a `distance` to the system (in pc), a field of view `fov` (in mas), a number of pixels `numpixels` (or a pixelsize in mas), and you must define the format of the input file via the `datatype` keyword.

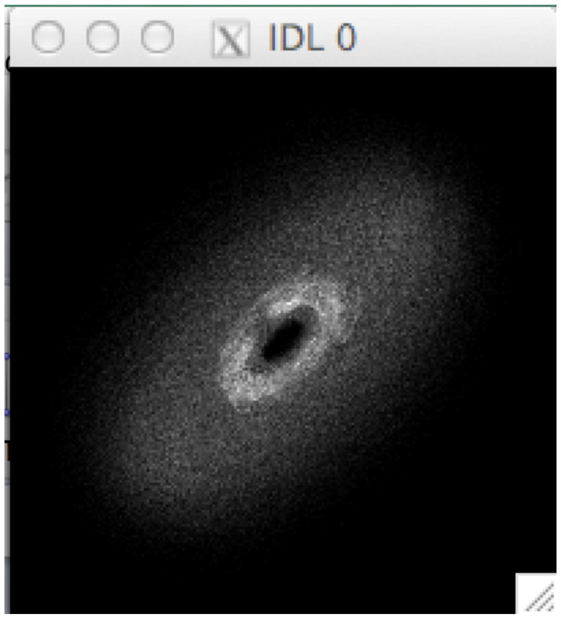
The output `image` will be a histogram of the number of points per pixel. For example, the above code produces the following result:

```
IDL> print,minmax(image)
      0.0000000       56.000000
```

You can change the orientation of the disk as well, via the angle keywords `inclination`, `pa`, and `longitude`, each of which are in degrees. `Inclination` tilts the disk around the image x-axis, `pa` rotates about the axis pointed out of the screen, and `longitude` rotates by the system's *z*-axis. For example,

```
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, inclination=60., longitude=90.,
pa=45.
```

creates the following histogram


IDL 0

## *3.2. Creating a geometric optical depth map*

In this mode, *dustmap* simply calculates the total cross section in each pixel per unit pixel area, i.e. a geometric optical depth. To use this mode, set the `opticaldepth` keyword. You can do this by setting `opticaldepth=1` or by setting `/opticaldepth`. For example, run

```
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, /opticaldepth
```

You'll notice that the above produces an error. To produce an optical depth map, you must supply the grain size `rdust` in microns via

```
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, /opticaldepth, rdust=1.
```

## *3.3. Creating a thermal emission image*

To create a thermal emission image, set the `thermal` keyword. Let's just replace the `opticaldepth` keyword from the previous example with the `thermal` keyword…

```
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, /thermal, rdust=1.
```

You'll notice that you get an error again. *dustmap* will tell you what you forgot to supply that is required to make a thermal emission image. Here is a correct call to *dustmap* to create a thermal emission image

```
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, /thermal, rdust=1., lambda=10.,
Lstar=1., Tstar=5778., composition='astrosil'
```

With the above call, a 10 micron thermal emission image (in Jy per pixel) is created assuming 1 micron astronomical silicate grains around a blackbody star with a temperature of 5778 K and a luminosity equal to 1 solar luminosity. *dustmap* iteratively calculates the temperature of the dust grains by balancing incoming and outgoing energy. By setting `composition='astrosil'`, the code is loading the indices of refraction for astronomical silicates and using Mie theory to calculate Qabs.

*dustmap* can easily and efficiently calculate multi-wavelength data cubes. For example, let's define lambda as a vector via

```
IDL> lambda = findgen(100)*10./99. + 0.3
```

And then call *dustmap*, setting `lambda=lambda`. We'll also set the `fstar=fstar` keyword.

```
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, /thermal, rdust=1.,
lambda=lambda, Lstar=1., Tstar=5778.,
composition='astrosil', fstar=fstar
```

Image is now a data cube with dimensions 200x200x100, where the 3rd index refers to each wavelength. The output variable `fstar` now contains the flux of the star as a function of wavelength. As a sanity check, let's run

```
IDL> plot,lambda,fstar,/xlog,/ylog
```

You should see a blackbody curve equal to the flux from the star in Jy. *dustmap* also can load a Kurucz stellar atmosphere model. To do this, set the `/kurucz` keyword as well as `logg`. For example, let's increase the resolution of lambda on a logarithmic scale and use a Kurucz model. NOTE: currently dustmap only has solar metallicity Kurucz models.

```
IDL> lambda = findgen(1000)/999.*(alog10(100.)-alog10(0.3))
+ alog10(0.3)
IDL> lambda = 10.^lambda
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, /thermal, rdust=1.,
lambda=lambda, Lstar=1., Tstar=5778.,
composition='astrosil', fstar=fstar, /kurucz, logg=4.5
IDL> window,0,xsize=500,ysize=500
IDL> plot,lambda,fstar,/xlog,/ylog
```

You should see absorption features in your stellar spectrum now. It's important to keep in mind that seemingly small changes in the UV can have a non-negligible impact on grain temperature, which can impact the SED of your disk. Speaking of which, let's plot the SED of our disk via

```
IDL> plot,lambda,total(total(image,1),1),/xlog,/ylog
```

You should see something that looks like a broadening blackbody, with a peak near 10 microns. That's what we'd expect because our input disk model has dust located near 1 AU.

Just like `fstar` contains the output stellar flux, we can set `qabs=qabs`, and `qabs` will contain the output values of `qabs`.

## *3.4. Creating a scattered light image*

***WARNING:*** *By default, dustmap sparsely samples the scattering phase functions at scattering angles <10° and >170°. If your disk has a significant amount of dust at these scattering angles, such that the image appearance or total disk flux could be affected, you may need to increase the resolution of the scattering phase function. See discussion below.*

*dustmap* can also create scattered light images. To turn on scattered light, set the `/scattered` keyword. For example,

```
IDL> lambda = 0.55
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, /scattered, rdust=1.,
lambda=lambda, Lstar=1., Tstar=5778.,
composition='astrosil', /kurucz, logg=4.5, inclination=45.
```

creates a V band scattered light image of a disk inclined by 45 degrees. After running the above command, you'll notice that you can only see half of the disk—this is the near side of the disk. You can obviously scale the image to see the full disk. For example,

```
IDL> loadct,39
IDL> tvscl,image[*,*,0]^0.25
```

The near side of the disk is much brighter than the far side because, by default, *dustmap* uses the scattering phase function calculated by Mie theory, which can be significantly forward scattering. In the text output from *dustmap,* you'll see

```
  Using Mie Theory SPF...
```

You can override this scattering phase function and use a Henyey-Greenstein scattering phase function by setting the `/hg` keyword and `g`. For example,

```
IDL> lambda = 0.55
IDL> dustmap, file, image, distance=10., fov=1000.,
numpixels=200, datatype=1, /scattered, rdust=1.,
lambda=lambda, Lstar=1., Tstar=5778.,
composition='astrosil', /kurucz, logg=4.5, inclination=45.,
/hg, g=0.1
```

will produce a modestly forward scattering disk with a Henyey-Greenstein scattering phase function. You should see the following in the *dustmap* output:

```
  Using Henyey Greenstein SPF...
```

Just like setting `fstar=fstar` returns the output stellar flux to the variable `fstar`, and `qabs=qabs` returns the emissivity, we can set `qsca=qsca` to return the scattering efficiency, `costheta=costheta` to return the cosine of the scattering angles used, and `pfunc=pfunc` to return the scattering phase function. You should be able to call the standard IDL size procedure to determine which index corresponds to what. For example, if we have 2 wavelengths,

```
IDL> print,size(costheta,/dim)
        500
IDL> print,size(pfunc,/dim)
           1           2         500
```

So the 2nd index of `pfunc` correspond to wavelength and the 3rd index correspond to scattering angle.

At the beginning of this section there is a warning about how *dustmap* sparsely samples the scattering phase functions at angles <10° and >170°. If your disk has a significant amount of dust at these scattering angles, the image may be inaccurate. To see what angles *dustmap* is using for the phase function, call *dustmap* with the keyword `costheta=costheta`, then run

```
IDL> print,acos(costheta)*180/!pi
```

If the default resolution is not sufficient, you can increase it by setting the `ncostheta` keyword. The default is `ncostheta=500`.

## 3.4. More *dustmap* features

- *Simultaneous scattered light and thermal emission:* simply set both the `/scattered` and `/thermal` keywords to include both scattered light and thermal emission

- *Scaling up the dust disk:* the `scaling` keyword (default `scaling=1`) simply multiplies each grain by this value. May be useful if creating size distributions.

- *Multiple input files:* you can send *dustmap* a vector of input files, e.g. `files = ['file1','file2']`. Two other *dustmap* keywords (`rdust` and `scaling`) can also be vectors, and each entry in those vectors will correspond to a given entry in `files`. For example, if you send *dustmap* the above `files` vector, as well as `rdust=[1.,2.]`, file1 will correspond to 1 micron grains and file2 will correspond to 2 micron grains. To operate *dustmap* as efficiently as possible, make sure to sort files by grain size. This feature is useful if you have many data points and not enough RAM to load them as a single file, or if you want to create size distributions.

- *Mie theory compositions: dustmap* comes with 7 possible compositions to be chosen by the `composition` keyword. The real and imaginary indices of refraction as a function of wavelength are stored in .lnk files in the lnkfile subdirectory. The user may want to supply his/her own composition. To do so, the user must create a .lnk file. To use this file instead of one of the default compositions, set the `lnkfile` keyword to a string containing the full path and name of the desired file.

- *Sublimation temperature: dustmap* can ignore/remove grains whose temperature exceeds the sublimation temperature. To do this, simply set the `tsublimate` keyword equal to the sublimation temperature in degrees K. By default, `tsublimate` is infinite. This only works in scattered light and thermal emission modes, and is not used in the histogram or optical depth modes.

- *Generic optical constants:* by default, *dustmap* assumes you want to use Mie theory. You may wish to specify some other optical constants. To do so, set the `/oc_generic_flag` keyword. You will also be required to set the `qexp` and `albedo` keywords, and will be forced to use a Henyey-Greenstein scattering phase function. For generic optical constants, the code calculates Qabs and Qsca as follows:

  $\text{Qabs} = 1 \qquad \text{for } \lambda < (2\pi\,\texttt{rdust})$

  $\text{Qabs} = (2\pi\,\texttt{rdust}/\lambda)^{\texttt{qexp}} \qquad \text{for } \lambda > (2\pi\,\texttt{rdust})$

  $\text{Qsca} = \texttt{albedo}/(1-\texttt{albedo}) * \text{Qabs}$

- *Alternate viewing geometries:* you can view the model disk from any location—even within the disk. To view a disk from within, it's easiest to set the `/AU` keyword, which tells the code that the `distance` keyword is in units of AU instead of pc. You should probably also set the `/degrees` keyword, which tells the code that the `fov` and `pixelsize` keywords are in units of degrees instead of mas. To make an all-sky map, simply set the `/allsky` keyword and the appropriate `fov` will be chosen. By default, *dustmap* creates a rectangular image. For large fields of view this can create projection problems (especially noticeable in all-sky maps where the poles will be darker). To correct for this, you can tell *dustmap* to create an Aitoff projection via the `/aitoff` keyword.

- *Data types:* to run *dustmap*, you must specify a binary input file containing the points, as well as the `datatype`. The keyword `datatype` specifies the format of the binary input file, and can range from a value of 1 to 4. See dmgetdata.c for the details of each format. Use dustmap_binary_writer.pro to easily make your own binary files. It will tell you the datatype it is using given the data you sent it.

## 4. Creating Binary Input Files

*dustmap* was created to synthesize images from *n*-body dynamical simulations, for which hundreds of millions of points may be necessary. To keep file sizes manageable and reduce the run time of the code, the input files are assumed to be binary files. For details of the binary file formats, please refer to dmgetdata.c. You can make your own binary data files with the proper formatting fairly easily.

### *4.1. Writing a binary file using a known set of points*

Let's say you already have vectors of x, y, and z coordinates. You can use dustmap_binary_writer.pro to convert these points into a binary file. To do so, simply call

```
IDL> dustmap_binary_writer, output_filename, particle_IDs,
x, y, z
```

where `output_filename` is a string of the output file to be created, `particle_IDs` is a vector of length *n* that identifies the "ID" number of each particle, and `x`, `y`, and `z` are vectors of length *n* that contain the 3D coordinates of each particle in AU. It is highly likely that you don't care about which particle created which set of coordinates, so for most applications, the entries in `particle_IDs` don't matter. For example, the following would be a valid call to dustmap_binary_writer.pro:

```
IDL> dustmap_binary_writer, 'test.dat',
intarr(n_elements(x)), x, y, z
```

dustmap_binary_writer.pro will tell you the `datatype` of the output file created, so that when you call *dustmap*, you will know what value to use.

Other available keywords for dustmap_binary_writer.pro are `vx`, `vy`, `vz`, and `intensity`. `vx`, `vy`, and `vz` are the x, y, and z velocities of each particle—data that is useful to dynamical modelers. `intensity` contains a scaling factor for each particle entry, i.e. how many physical particles that one entry represents.

## *4.2. Writing a binary file using an analytic description*

To create a distribution of points from an analytic description, one could randomly distribute points, assign each point an `intensity` based on the analytic density distribution, then write those points to file. Unfortunately, due to the randomization of particle locations, the image will suffer from Poisson noise. On the other hand, this method is relatively easy to code and allows the user to produce images of this set of points at any orientation.

Alternatively, one could greatly reduce the noise of the images by producing a set of particle coordinates "tuned" to the orientation of the disk and the resolution of the image to be produced. Obviously this requires knowing the disk orientation and image resolution prior to making the image. So if you are using *dustmap* to try to determine the disk's orientation, you will have to create a new set of points for each orientation.

I'm working on a tool to assist with this…