`CirBinDis`
Circumbinary disk analyser

Paul Magnus Sørensen-Clark
Jerome Bouvier

July 15, 2015

Contents

Contact:

Paul Magnus Sørensen-Clark: paulmag91@gmail.com

Jerome Bouvier: jerome.bouvier@obs.ujf-grenoble.fr

`https://bitbucket.org/paulmag/circumbinarydisk`

1 Introduction

A small piece of software for receiving an artificial light curve from a simulated density map of a gas disk around a binary star.

This explains certain procedures with bash-commands, which exists in Linux/UNIX and Apple OS. `CirBinDis` should work in Windows as well, but some bash-commads may be different.

2 Installing

2.1 Downloading

The source code is available at this Bitbucket repository:
`https://bitbucket.org/paulmag/circumbinarydisk`
Provided that Git is installed on you computer you can easily get all the files by running the following command inside the folder where you want the repository:
`git clone https://paulmag@bitbucket.org/paulmag/circumbinarydisk.git`
This link will be updated if the location of the repository or the installation process in any other way changes (note that at this moment the repository is private). Should it not work then contact the authors via email.

2.2 Alias

We recommended to make the alias "`cirbindis`" for the command
`python ~/path_to_repository_folder/circumbinarydisk/src/main.py`.
F. ex. place this in your `.bashrc` or `.bash_aliases`:
`alias cirbindis="python /GitHub/circumbinarydisk/src/main.py"`
This alias will be assumed for the rest of this manual.

3 Preparing your data

The format of the input data must be an ASCII/CSV-file with three columns where each line represents a datapoint in space. The two first columns of each line represent the position of a datapoint. (x, y) if using cartesian coordinates and (r, θ) if using polar coordinates. The last column represents the density in this position.

Any units can be used for the input data. How to specify units are covered in the section **Configuring and running** `CirBinDis` .

4 Processing algorithm

`CirBinDis` produces artificial lightcurves by analysing the provided dataset according to given configurations. In this section the process for extracting the lightcurve from the dataset is explained.

4.1 Loading data

TODO

4.2 Cropping

The space covered by the dataset may represent a larger area than the disk you want to analyse. The dataset is cropped to an inner and outer radius such that the shape of the remaining datapoints resembles a donut. The outer radius represents the size of the disk and makes sure that the disk is circular. The inner radius is necessary to avoid treating the stars themselves as dust, and the density of the dust is very low close to the stars anyway.

4.3 Rotating

The coordinates of all datapoints are rotated stepwise with the rotation matrix R_z for $\theta = [0, 360)\,^\circ$.

$$R_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This rotation simulates the physical orbital rotation of the dircumbinary disk. The reason we get a variation in the lightcurve is because when the disk rotates we see the stars through different areas of the disk with different densities.

A rotation also happens around the y-axis due to the inclination angle ϕ. R_y is the rotation matrix which would perform this rotation. However, R_y is *not* used, and the y-rotation is never performed directly. It is implicitly done in a very different way in a following section **Mean density of bins, weighted and integrated**.

$$R_y = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ \sin(\phi) & 0 & \cos(\phi) \end{bmatrix}$$

4.4 Sylinder

A section of the datapoints are cropped out, which represents only the sylinder of gas that is between an observer on Earth and the star. These are the datapoints

that fall within the sylinder whose base area is defined by the stellar surface and which extends from the stellar surface and infinitely along the x-axis in positive direction (the de facto limit is the outer radius of the disk). In other words, the observer's position is assumed to be $(\infty, 0, 0)$. A sylinder like this is made once for *each* azimuthal rotation of all the points. Thus, each sylinder will be a little different from the previous one (if $\mathrm{d}\theta$ is small). If there are two (or even more) stars a sylinder will be created for the line of sight of each star, so there can be two (or even more) sylinders at the same time.

4.5 Binning

4.5.1 Algorithm

Each sylinder is sliced up into n_{steps} bins along the line of sight, where n_{steps} is given by the field `radiussteps` in `input.xml`. $N_{sylinder}$ is the number of datapoints contained within a sylinder. For each bin the mean density is computed. The binning algorithm works like this:

1. Sort all datapoints in sylinder according to x-component.
2. Find $N_{bin} = N_{sylinder}/n_{steps}$.
3. First N_{bin} (sorted) datapoints goes in the first bin, next N_{bin} datapoints go in the second bin, etc.
4. Create corresponding $\mathrm{d}r$ array, where the $\mathrm{d}r$ corresponding the each bin is the difference between the x-component of the first and last datapoint in that bin.

4.5.2 Reasoning

An alternative way this could be done is have a static Δr and check which points fall within $[r, r + \mathrm{d}r]$ for r in $[0, 1, 2, 3 \ldots] \cdot \Delta r$, but this requires a boolean test on the entire sylinder for each radius. It is much faster to sort the datapoints in the sylinder once and then just slice it with indices. There could be even smarter ways to do it, but this has worked well for now. A side effect of this method is that Δr is smaller in areas where there are more datapoints. If the grid of datapoints is spaced denser in central areas where the most interesting features are this is a bonus compared to a static Δr.

4.6 Mean density of bins, weighted and integrated

4.6.1 Mass integral

For each bin a mean density is produced from all the datapoints in that bin. This is done by dividing the total mass of the sylinder with its volume. The mass of a

sylinder can be calculated from the following integral.

$$M_{bin,j} = \int\int\int_{V_{bin,j}} \rho(x,y,z)\ \mathrm{d}x\ \mathrm{d}y\ \mathrm{d}z$$

All our density datapoints are in the xy-plane, so the x and y part of the integral can be evaluated by summing the density of each datapoint $\rho_{0,i}$ multiplied with its respective discrete $\mathrm{d}x_i\ \mathrm{d}y_i$. We will assume that the datapoints in the grid is spaced evenly. This is not necessarily true, but it should be approximately true for most cases, especially if the size of the bin is much smaller than the whole dataset. In this case $\mathrm{d}x_i$ and $\mathrm{d}y_i$ is the same for every datapoint.

Whe have no data for density variation in the z-dicetion. Instead we assume a gaussian decrese of density with increasing distance from the midplane. $\rho(z)$ is the assumed density at a point with altitude z above a point i in the midplane with density ρ_0.

$$\rho(x_i, y_i, z) = \rho_0(x_i, y_i) \cdot \exp\left(-\frac{z^2}{2H^2}\right)$$

Thus the mass integral has a discrete part and an analytical part. The limits of the discrete part is the area of the midplane S_{bin} contained in the sylinder and is such that $N_{bin}\ \mathrm{d}x_i\ \mathrm{d}y_i = S_{bin}$. The limits for z are explained in the following subsection.

$$\begin{aligned} M_{bin} &= \int\int_{S_{bin}} \rho_0(x_i, y_i)\ \mathrm{d}x_i\ \mathrm{d}y_i \cdot \int_{z_{i,a}}^{z_{i,b}} \exp\left(-\frac{z^2}{2H^2}\right)\ \mathrm{d}z \\ &= \sum_i \left(\rho_{0,i}\ \mathrm{d}x_i\ \mathrm{d}y_i \cdot \frac{\sqrt{\pi}}{2}\sqrt{2H^2}\left[\mathrm{erf}\left(\frac{z_{i,b}}{\sqrt{2H^2}}\right) - \mathrm{erf}\left(\frac{z_{i,a}}{\sqrt{2H^2}}\right)\right]\right) \end{aligned}$$

4.6.2 The z-limits

Each density point is given a weight according to where it is in the sylinder. Points closer to the middle of the sylinder gains larger weight because they represent its fulll height and thus a larger volume than points near the edges. This is illustrated in figure 1 on the following page.

$$W_i(y) = \frac{\sqrt{r_{star}^2 - (y_i - y_{star})^2}}{\cos(\phi)}$$

The factor $1/\cos(\phi)$ adjusts the height of the sylinder if it is inclined so that it is always shaped like a circular sylinder.

To get the density inside the entire area of the slice of the sylinder and the variations in density from different altitudes we integrate the density for each point, projected from the bottom (z_a) to the top (z_b) of the sylinder. The distance to integrate is $2W_i$ for each point, centered around z_i.

$$\begin{aligned} z_i &= (x_i - x_{star}) \cdot \tan(\phi) \\ z_{i,a} &= z_i - W_i \\ z_{i,b} &= z_i + W_i \end{aligned}$$

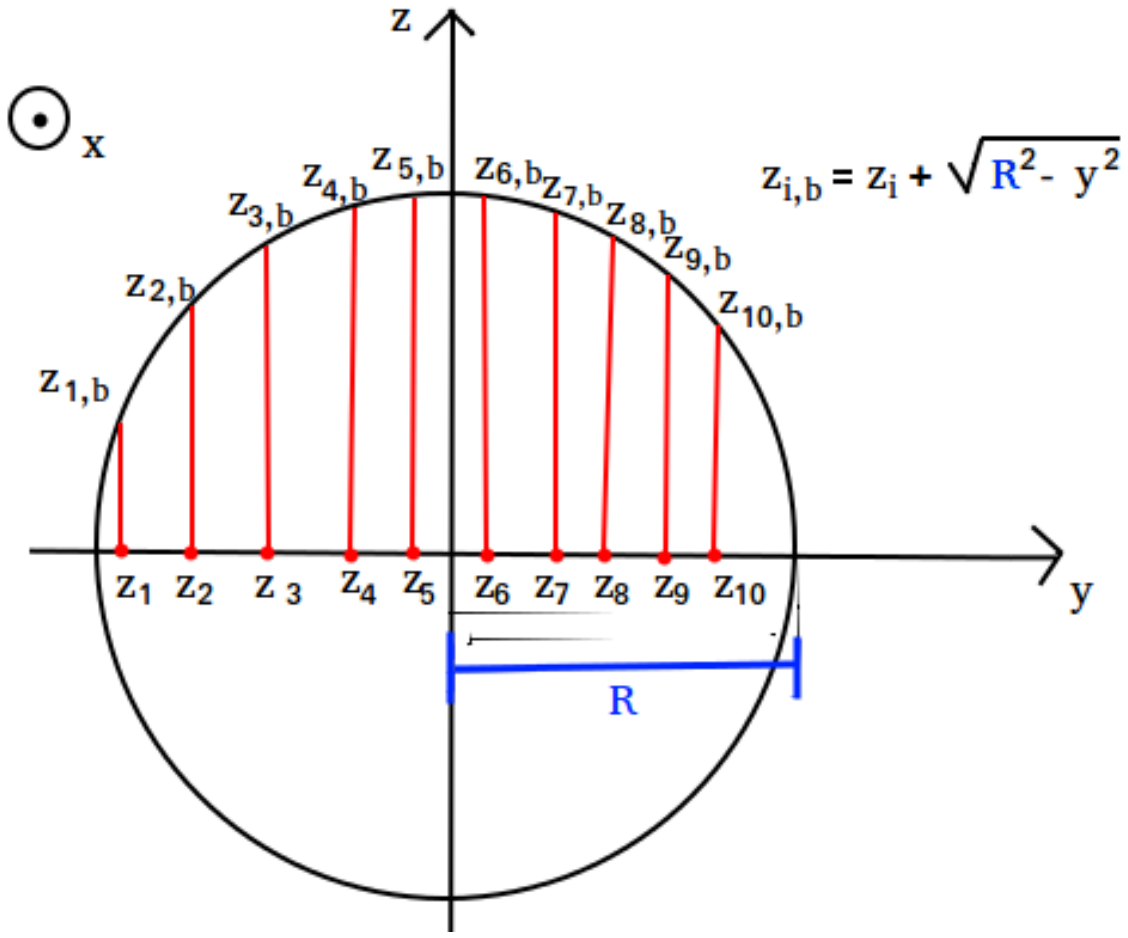

Figure 1: A cross-section of the sylinder. The figure shows how the limits $z_{i,a}$ and $z_{i,b}$ for each datapoint depend on that point's y-component y_i. The position of $z_{i,a}$ in the figure is the same as for $z_{i,b}$, but mirrored to the lower half of the circle.

4.7 Integrating intensity

For each bin j in each sylinder, from the inside to he outside of the disk, the ratio of intensity transferred from one bin to the next is given by the following expression.

$$\tau_j = \kappa \cdot \rho_{bin,j} \cdot \Delta r_j$$
$$I_{j+1} = I_j \exp(-\tau_j)$$

The resulting intensity passed on ny the outermost bin I_{end} is the intensity of the star's radiation that escapes the disk and is observed by the observer on the current line of sight. If there are several stars and thus several sylindres, the total perceived intensity is simply the sum of the $I_{end,k}$ for each sylinder k.

$$I_{total} = \sum_k I_{k,end}$$

4.8 Full algorithm summary

This is how one analysis is performed, and the product is one lightcurve. If providing different values for certain parameters, like different inclination angles or different outer radii then this analysis will be performed once for each different value of each parameter (different inclinations are actually analysed in quasi-parallell for efficiency).

for each θ in $[0, \ldots, 2\pi]$ **do**
  rotate density datapoints angle θ with R_z
  rotate stars angle θ with R_z
  for each star k **do**
    extract sylinder
    bin sylinder
    for each bin j in sylinder **do**
      $\rho_{bin,j} = \dfrac{\int\int\int_{V_{bin,j}} \rho(x,y,z)\ \mathrm{d}x\ \mathrm{d}y\ \mathrm{d}z}{\pi r_{star,k}^2 \Delta r_j}$
      $\tau_j = \kappa \cdot \rho_{bin,j} \cdot \Delta r_j$
      $I_{k,j+1} = I_{k,j} \exp(-\tau_j)$
    end for
  end for
  $I_{\theta,total} = \sum_k I_{k,end}$
end for

5 Configuring and running `CirBinDis`

How to make necessary configurations and then run `CirBinDis` to perform an analysis.

This is the most practical and maybe the most important section, as it explains how to actually use the software.

5.1 Input parameters

The input parameters for each run of `CirBinDis` is configured in an XML file with a predetermined layout. Inside the repository you will find `/xml/input.xml`. Copy this file, save it together with your dataset, and modify the values of the fields as required for your dataset (do not blindly use the default values).

Specifically, this is where you provide the filename of the dataset to analyse. If `input.xml` is in another folder than the dataset you need to write the relative or absolute pathname of the datafile.

You can save your copy of the XML-file with whatever name you wish, which can be useful to link separate XML-files to specific datasets that are in the same folder, or if you have diffenert sets of parameters that you want to reuse on the same dataset.

TODO: Here I plan to more or less copy the explanations that are in `xml/input.xml` already.

unit-mass Dorem

unit-distance ipsum

unit-intensity dolum

5.2 Executing the code

When you have prepared your input XML-file with your dataset, type the command `cirbindis`
(or `python ~/path_to_repository_folder/circumbinarydisk/src/main.py input.xml`) followed by the name of your XML-file in a terminal. F.ex.:

```
cirbindis input.xml
cirbindis dataA.xml
cirbindis dataA_big.xml
cirbindis data/set1.xml
```

The software will run until it has completed the analysis of your dataset with the parameters you specified, or stop and throw an error message if there is a problem with the configuration.

5.3 Output

The output after a `CirBinDis` analysis is a comma-separated-value (csv) file. The output file will be placed in the path you specified in `input.xml`. The filename contains the value of the parameters H, r_{in}, r_{out}, and inclination (ϕ), separated by double underscores "__". A filename can be f.ex. "`H=0.1__r_in=0.75__r_out=3__inc=5.csv`". If you perform several analysis of the same dataset at once, f.ex. by providing several values for r_{out}, then one outfile will be produced for each different value of r_{out}.

The first line of the output is a header containing all the physical and numerical parameters for the simulation. There are two columns. The first column lists rotation angles θ in units of degrees. The second column lists the observed intensities given the respective angles, normalised so the mean intensity is 1. A output file with `azimuthsteps=8` can look like this:

```
#H=0.1, kappa=10, r_star=0.21-0.19, r_in=0.75, r_out=3, dr=0.75,
dtheta=45deg, inc=5deg
0.000000,0.000000
45.000000,0.000000
90.000000,0.000000
135.000000,0.000000
180.000000,7.644186
225.000000,0.000000
270.000000,0.355814
315.000000,0.000000
```

6 Troubleshooting

6.1 Some curves are missing in the plot

6.2 Contact the author

If you cannot find out how to do something and this manual does not explain it, send an email to paulmag91@gmail.com to ask and ask. Do this also if you have suggestions for improvements, as `CirBinDis` is under development.

7 Bibliography

8 Acknowledgments

9 Source code

9.1 `input.xml`

The file for the user to provide input parameters to `CirBinDis` . You can copy and modify it. It is not a part of the source code itself.

```
<?xml version="1.0"?>

<!--==========Explanation of input XML file====================================-->
<!--This is a comment.-->
<!--With this file you can set all the input parameters for analysing a
    circumbinary disk. Make a copy of this file and place it in the directory
```

```
    you run the program from. Then you can just run 'python
    somepathname/main.py'. You can edit all the leaf nodes in this file,
    that is any field between two tags. F.ex. you can replace '0 10 30' with
    '5 15 25' in '<inclination>0 10 30</inclination>'. There are
    descriptions of the meanings of all the elements, and some elements have
    examples under them, where the different examples are separated with
    vertical | bars | like | this.
-->

<input>

    <!--==========Input units==================================================-->
    <!--The units provided here are the units of the input values provided
    in this file, except for kappa. You can choose between the suggested
    units under each field (and other standard units). You can also use a
    quantity of a unit, f.ex. you can measure distance in units of '10.5
    solRad' and mass in units of '1e27 kg'. Output units are always in
    CGS-units and degrees.
    -->
    <unit>
        <mass>solMass</mass>
            <!--Examples:  solMass  |  kg  |  g -->
        <distance>10.921 solRad</distance>
            <!--Examples:  AU  |  solRad  |  km  |  m  |  cm -->
        <intensity>erg s^-1 cm^-2</intensity>
            <!--Examples:  erg s^-1 cm^-2   |   J / (m2 s) -->
        <angle>deg</angle>
            <!--Examples:  rad  |  deg  |  arcmin  |  arcsec -->
    </unit>

    <!--==========Input data====================================================-->
    <!--The pathname to the dataset to be analysed. The pathname must be
        relative to where you run the program from, or it can be an absolute
        path. It can be an ACII file with (x,y,density) columns or a pickle
        file. If it is ASCII you must also specify if the coordinates are
        cartesian (x,y) or polar (r,theta).
    -->
    <datafile>data/data_0.25</datafile>
    <!--A name/identifier for the current dataset. This name will be included
        in the filenames of the output lightcurves and in the title of the
        plots. This field is optional. If left blank the datafilename will be
        used as the default dataname.
    -->
    <dataname></dataname>
        <!--Examples:  data0032  |  cs=0.25 -->
    <!--If you provide a pathname here the cropped version (according to
        "Disk Boundaries) of the input datafile will be saved. This cropped
        version of the data can then be used as the input datafile later. If
        you use "*.p" or "*.pickle" as filename it will be saved as a pickle
        file. This field is optional. If left blank nothing will happen. If
        several pathnames are provided (separated with spaces) several
        copies will be made.
    -->
    <!--
        How to normalize the output data.
        Options are "mean", "max", "stellar", or any combination of these.
    -->
    <normalization>stellar</normalization>
        <!--Examples:  stellar mean  |  max  |  all -->
    <resave_as></resave_as>
        <!--Examples:  data_0.25_cropped3.0.csv  |  data_0.25_cropped3.0.p -->
    <system>cartesian</system>
        <!--Examples:  cartesian  |  polar -->

    <!--==========Output data===================================================-->
    <!--The pathname to the folder to contain the output files. The pathname
        must be relative to where you run the program from, or it can be an
        absolute path. If the folder does not already exist it will be
        created automatically.
    -->
```

```
<outfolder>results</outfolder>

<!--==========Disk boundaries==============================================-->
<!--Define the inner and out radius for where the disk exists. The
    dataset will be cropped to these limits, so it becomes a "donut".
    You can provide several inner and outer radiuses by separating them
    with spaces it you want to analyse different sized disks.
    If radius_in is not provided it will default to just outside the
    position of the stars. If radius_out is not provided it will default
    to the radius that defines the largest circle that can fit within
    the dataset.
-->
<radius_in>1.0</radius_in>
    <!--Examples:  1  |  1.5  |  1 1.25 1.5 -->
<radius_out>3.0</radius_out>
    <!--Examples:  3  |  5  |  4 6  |  3 5 7.5 10 -->

<!--==========Inclinations================================================-->
<!--Which inclinations to analyse the system in. Several inclinations
    can be separated with spaces. If no inclinations is provided a
    default of 0 will be chosen.
-->
<inclination>0 10 30</inclination>
    <!--Examples:  10  |  0 2.5 5  |  0 5 10 20 30 -->

<!--==========Other physical disk parameters==============================-->
<!--Total mass of disk, excluding the stars.
    Total radius of the disk, including the far outer parts that are not
    considered in the analysis.
    Opacity, in units of [cm^2/g].
    Thickness of disk.
-->
<diskmass>0.01</diskmass>
<diskradius>985</diskradius>  <!--50 AU in current units.-->
<kappa>10.0</kappa> <!--[cm^2/g].-->
<H0>0.1</H0>

<!--==========Star parameters==============================================-->
<!--There can be one or more stars with different positions, radiuses
    and intensities. They will be integrated separately and have their
    resulting fluxes added together.
    * If you want to add another star:
      Copy-paste a star element anc change its parameters.
    * If you want to remove a star:
      Delete a star element.
    If no stars are provided a default star will be made at origo with
    radius=radius_in and solar luminosity.
    Provide EITHER cartesian (x,y) OR polar (r,theta) coordinates for
    each star.
-->
<star>
    <position>
        <x>-0.5</x>
        <y>0</y>
        <r></r>
        <theta></theta>
    </position>
    <radius>0.21</radius>
    <intensity>1.1</intensity>
</star>
<star>
    <position>
        <x>0.5</x>
        <y>0</y>
        <r></r>
        <theta></theta>
    </position>
    <radius>0.19</radius>
    <intensity>0.9</intensity>
</star>
```

```
    <!--Example:
        <star>
            <position>
                <x></x>
                <y></y>
                <r>0.5</r>
                <theta>5</theta>
            </position>
            <radius>0.2</radius>
            <intensity>1.0</intensity>
        </star>
    -->

    <!--==========Numerical parameters========================================-->
    <!--How many azimuthal rotation steps to use. This is the resolution of
        the resulting lightcurve.
        dtheta = 360deg / azimuthsteps
    -->
    <azimuthsteps>18</azimuthsteps>
    <!--How many radial steps to use in each line-of-sight integration. This
        defines dr and the accuracy of each fluxpoint in the resulting
        lightcurve.
        dr = (radius_out - radius_in) / radiussteps
    -->
    <radiussteps>10</radiussteps>

</input>
```

9.2 `main.py`

The script that is called when running `CirBinDis` .

```
import sys
import numpy as np
import xmltodict

from DensityMap import DensityMap
from Star import Star
import Functions as func


if __name__ == "__main__":

    # Attempt to read input file and tell the user if the input is wrong:
    try:
        infile = open(sys.argv[1], "r")
    except IndexError:
        print (
            "usage: python %s input_file.xml\n"
            "  Specify the (path)name of an input xml file. "
            % sys.argv[0]
        )
        sys.exit(1)
    except IOError:
        print "No such file: '%s'" % sys.argv[1]
        print (
            "usage: python %s input_file.xml\n"
            "  Specify the (path)name of an input xml file. "
            % sys.argv[0]
        )
        if sys.argv[1] in (
            "help", "Help", "-h", "-H", "--help", "man", "manual"
        ):
            print (
                "  See the manual for details on how to use cirbindis:\n"
                "  https://bytebucket.org/paulmag/circumbinarydisk/raw/"
```

```
                "3945ccbba15ae5fcac439cd5d71dce07b6e35796/report/main.pdf"
            )
        sys.exit(1)
    try:
        input_ = xmltodict.parse(infile)["input"]
    except Exception:
        print (
            "usage: python %s input_file.xml\n"
            "  Specify the (path)name of an input xml file. "
            % sys.argv[0]
        )
        raise
    infile.close()


    for radius_in in func.to_list(input_["radius_in"], float):

        if radius_in is None or np.isnan(radius_in):
            radius_largest = 0.
            for stardict in func.to_list(input_["star"]):
                star = Star(stardict)
                radius = np.linalg.norm(star.position) + star.radius
                if radius > radius_largest:
                    radius_largest = radius
            radius_in = radius_largest

        for radius_out in func.to_list(input_["radius_out"], float):

            dataset = DensityMap(
                filename=input_["datafile"],
                dataname=input_["dataname"],
                coordsystem=input_["system"],
                unit=input_["unit"],
                inclinations=func.to_list(input_["inclination"], float),
                radius_in=radius_in,
                radius_out=radius_out,
                diskmass=float(input_["diskmass"]),
                diskradius=float(input_["diskradius"]),
                H=float(input_["H0"]),
                kappa=float(input_["kappa"]),
            )

            for filename in func.to_list(input_["resave_as"]):
                if filename is None:
                    continue
                elif len(filename) == 0:
                    continue
                dataset.writeto(filename)

            dataset.set_physical_units()

            for star in func.to_list(input_["star"]):
                dataset.add_star(star)

            dataset.make_lightcurve(
                n_angle=int(input_["azimuthsteps"]),
                n_radius=int(input_["radiussteps"]),
                unit=input_["unit"]["angle"],
                save=True,
                savefig=True,
                show=True,
                normalizations=func.to_list(input_["normalization"]),
                outfolder=input_["outfolder"],
            )
```

9.3 DensityMap.py

Contains the class `DensityMap` for making an instance of a dataset representing a circumbinary disk. It contains most methods that can be performed on the data. Also contains a subclass `Sylinder`. Sylinders a sub-sets of a full dataset.

```
import sys
    # For doing meta things like receiving command-line arguments and exiting
    # the program.
import numpy as np
    #  Numerical Python. Contains mathematical functions for performing
    # computations on arrays/matrices fast.
import cPickle as pickle
    # Can be used to save (dump) and load Python objects to files. Much
    # faster than reading and writing ASCII tables.
import time
    # Used to time parts of code to look for bottlenecks.
import re
    # Regular expressions for data reading.
import matplotlib.pyplot as plt
from matplotlib import ticker
    # For plotting results.
from scipy import integrate
from scipy import special
import astropy.units as u
import textwrap
    # String manipulations.

from Star import Star
import Functions as func



class DensityMap:


    def __init__(self,
        data=None,
        filename=None,
        dataname=None,
        coordsystem="cartesian",
        outfolder=None,
        unit=None,
        inclinations=None,
        radius_in=0,
        radius_out=np.inf,
        diskmass=.01,
        diskradius=1000.,
        H=1.,
        kappa=10.
    ):

        self.data_rotated = None
        # If the inclination is a single number, put it in a list:
        try:
            iter(inclinations)
            self.inclinations = inclinations
        except TypeError:
            self.inclinations = [inclinations]
        if dataname is None or dataname == "":
            self.dataname = filename.split("/")[~0]
        else:
            self.dataname = dataname
        print (
            "Loading dataset '%s' from file '%s'..."
            % (self.dataname, filename)
        )
        self.outfolder = outfolder
```

```
        self.unit = unit
        self.stars = []
        self.radius_in = radius_in
        self.radius_out = radius_out
        self.diskmass = diskmass
        self.diskradius = diskradius
        self.H = H
        self.kappa = kappa  # [cm^2 / g]
            # Between 5 and 100 according to Bouvier et al. 1999.

        if data is not None:
            self.data = data
        elif filename is not None:
            self.load(filename)

        if coordsystem == "cartesian":
            pass
        elif coordsystem == "polar":
            x, y = func.pol2cart(self.data[:, 0], self.data[:, 1])
            self.data[:, 0], self.data[:, 1] = x, y
        else:
            raise KeyError("Coordinate system must be 'cartesian' or 'polar'.")


    def add_star(self, d=None, position=None, radius=None, intensity=None):
        """Make a Star instance and store it in a list of stars for the disk.

        d: (dictionairy) Must contain position, radius and intensity and can
            be provided instead of giving these other arguments individually.
        position: (float, array-like) Coordinates of the star.
        radius: (float) Radius of the star.
        intensity: (float) Intensity of the star.
        """
        self.stars.append(Star(
            d=d, position=position, radius=radius, intensity=intensity
        ))


    def load(self,
        filename,
        method=None,
    ):
        """Load a dataset to analyse from a file.

        Assumed to be on the form 'x,y,density' or 'x,y,z,density' which
        represents a point in cartesian space and the density at that point.
        If the z-coordinate is not given it will be assumed to be 0 for
        every point. Resulting data is an array of shape (N, 4), where N is
        the number of data points.

        filename: (string) Full pathname to file containing dataset.
        method: (string) What kind of loading algorithm to use. Can be
            'ascii' or 'pickle', If none is given, will try to automatically
            find out by looking at file ending.
        """

        t_start = time.time()
            # Just to time the loading, in case of large dataset.
        infile = open(filename, "r")

        if method is None:
            if filename.endswith(".p") or filename.endswith(".pickle"):
                method = "pickle"
            else:
                method = "ascii"

        if method == "pickle":
            data = pickle.load(infile)
            mask = (
                (np.linalg.norm(data[:, 0:2], axis=1) >= self.radius_in) *
```

```
                (np.linalg.norm(data[:, 0:2], axis=1) <= self.radius_out)
            )
            self.data = data[np.where(mask)]

        elif method == "ascii":
            data = []
            for line in infile:
                line = re.split("[, ]+", line.strip())
                    # Split on comma/space and strip spaces/newline.
                if len(line) >= 3:
                    line = [float(value) for value in line]
                    if (self.radius_in <=
                        np.linalg.norm(line[0:2]) <=
                        self.radius_out
                    ):
                        data.append(line)
            self.data = np.array(data)

        infile.close()
        t_end = time.time()  # End of timer.
        print "Loading took %f seconds." % (t_end - t_start)


    def set_physical_units(self, mass_total=None, r_total=None):
        """Convert density to physical units related to the total mass and
        size of the disk.
        """

        if mass_total is None:
            mass_total = self.diskmass
        if r_total is None:
            r_total = self.diskradius

        r_in = u.Quantity(self.radius_in, self.unit["distance"])
        print "r_in =", r_in
        r_out = u.Quantity(self.radius_out, self.unit["distance"])
        print "r_out =", r_out
        r_total = u.Quantity(r_total, self.unit["distance"])
        print "r_total =", r_total
        H = u.Quantity(self.H, self.unit["distance"])
        print "H =", H
        mass_total = u.Quantity(mass_total, self.unit["mass"])
        print "mass_total =", mass_total

        mass_central = (
            (r_out**0.5 - r_in**0.5) /
            (r_total**0.5 - r_in**0.5) *
            mass_total
        )
        print "mass_central =", u.Quantity(mass_central, self.unit["mass"])
        rho_central = (
            mass_central / (np.pi * (r_out**2 - r_in**2) * np.sqrt(np.pi/2)*H)
        ).to(u.Unit(self.unit["mass"]) / u.Unit(self.unit["distance"])**3).value
        print "rho_central =", u.Quantity(
            rho_central,
            u.Unit(self.unit["mass"]) / u.Unit(self.unit["distance"])**3,
        )
        rho_central_CGM = (
            mass_central / (np.pi * (r_out**2 - r_in**2) * np.sqrt(np.pi/2)*H)
        ).to(u.Unit("g") / u.Unit("cm")**3).value
        print "rho_central =", u.Quantity(
            rho_central_CGM,
            u.Unit("g") / u.Unit("cm")**3,
        )

        # self.data[:, ~0] /= self.data[:, ~0].mean()
            # Normalize before scaling, or not?
        print "Mean BEFORE:", self.data.mean()
        self.data[:, ~0] *= rho_central
        print "Mean AFTER: ", self.data.mean()
```

```
def writeto(self, filename, method=None, separator=" "):
    """Write self.data to a file for later use.

    Assumed to be on the form 'x,y,z,density' which represents a point in
    cartesian space and the density at that point.

    filename: (string) Full pathname to outfile for writing data.
    method: (string) What kind of writing algorithm to use. Recommended to
        use 'pickle' if it will be loaded by this program later (faster)
        and 'ascii' for any other purpose. If none is given, will try to
        automatically find out by looking at filename ending.
    separator: (string) If method='ascii' this is the separator between the
        values each line. Usually a space or comma. Ignored if
        method='pickle'.
    """

    t_start = time.time()
        # Just to time the writing, in case of large dataset.

    if method is None:
        if filename.endswith(".p") or filename.endswith(".pickle"):
            method = "pickle"
        else:
            method = "ascii"

    if method == "pickle":
        outfile = open(filename, "wb")
        pickle.dump(self.data, outfile)

    elif method == "ascii":
        outfile = open(filename, "w")
        for line in self.data:
            outfile.write("%f%s%f%s%f\n" % (
                line[0], separator,
                line[1], separator,
                line[2],
            ))

    outfile.close()
    t_end = time.time()  # End of timer.
    print "Writing took %f seconds." % (t_end - t_start)


def rotate(self, angle_z=0, unit="deg"):
    """Rotate entire dataset by an angle around any axis.

    The original data is not changed. Rotated version of data stored in
    self.data_rotated. self.stars are also rotated.

    angle_z: (float) Angle to rotate around z-axis. This is the rotational
        axis for the disk. It can be gradually increased to simulate the
        orbital rotation of the system.
    angle_y: (float) Angle to rotate around y-axis. The inclination between
        the disk and the field of view. This angle should always be the same
        for one analysis if the disk is not wobbling.
    angle_x: The x-axis is the line of sight. Rotations around this axis
        would have no effect on the received flux, therefore this angle is
        ignored.
    unit: (string) What unit angles are given in.
        'rad', 'deg', 'arcmin' or 'arcsec'.
    """

    # Transform angles into radians:
    if unit == "rad":
        factor = 1.
    elif unit == "deg":
        factor = np.pi / 180.
    elif unit == "arcmin":
```

```
            factor = np.pi / 180. * 60
        elif unit == "arcsec":
            factor = np.pi / 180. * 3600
        angle_z *= - factor
            # Multiply by negative factor so that the disk is rotated
            # clockwise. Then the line of sight (observer) is rotated
            # counter-clockwise.

        # Make rotation matrix:
        rotation_matrix = np.matrix([
            [ np.cos(angle_z), -np.sin(angle_z)],
            [ np.sin(angle_z),  np.cos(angle_z)],
        ])

        # Rotate the disk:
        coords_in = self.data[:, :~0]
        coords_out = \
            np.asarray(rotation_matrix * coords_in.transpose()).transpose()
        self.data_rotated = np.hstack((coords_out, self.data[:, ~0, None]))

        # Rotate the stars:
        for star in self.stars:
            star.position_rotated = np.asarray(
                rotation_matrix * star.position[:, None]
            ).transpose()[0]


    def distance(self, p1, p2=None):
        """Returns the distances from the (rotated) datapoints to a line.

        p1: (float, array) A point in space which defines a line with p2.
        p2: (float, array) A point in space which defines a line with p1. If
            p2 is not provided it is assumed that the line is parallell to
            the x-axis.

        return: (float, array) The shortest euclidian distances between
            points and the line (p1, p2).
        """

        if p2 is None:
            p2 = p1.copy()
            p2[0] += 1.

        return np.abs(
            np.cross(
                p1 - self.data_rotated[:, :~0],
                p2 - self.data_rotated[:, :~0],
            ) /
            np.linalg.norm(p2 - p1),
        )


    def get_sylinder(self, starno=None, star=None):
        """Slice out a sylinder shape from the data based on a star.

        The sylinder is always oriented along the x-axis. Its size and
        position is determined by a star. It spans from the position of the
        surface of the star until x=inf.

        starno: (int) Index to get star from self.stars. Ignored if star is
            given.
        star: (Star instance) A star to base the sylinder on.

        return: (float, array) The slice of the dataset contained in the
        sylinder.
        """

        if star is None:
            star = self.stars[starno]
```

```
        mask = (
            (self.data_rotated[:, 0] > 0) *
            (self.distance(star.position) <= star.radius)
        )
        data_sylinder = self.data_rotated[np.where(mask)]
        return data_sylinder


    def get_density_profile(self, sigma=1, skip=2000, show=True):
        """Returns and displays the density profile of self.data.
        TODO: Finish this docstring.
        """

        from scipy.ndimage.filters import gaussian_filter1d

        radiuses = np.linalg.norm(self.data[:, 0:2], axis=1)
        indices_sorted = np.argsort(radiuses)
        radiuses = radiuses[indices_sorted][::skip]
        densities = gaussian_filter1d(
            self.data[:, 3][indices_sorted],
            sigma=sigma,
            mode="nearest",
        )[::skip]

        if show:
            plt.plot(radiuses, densities, "b+")
            plt.xlabel("radius")
            plt.ylabel("density")
            plt.show()

        return radiuses, densities


    def make_lightcurve(self,
        inclinations=None,
        H=None,
        n_angle=None,
        dtheta=None,
        theta=None,
        unit="deg",
        n_radius=None,
        dr=None,
        save=False,
        savefig=False,
        show=False,
        normalizations=["stellar"],
        short_title=True,
        outfolder=None,
    ):
        """Makes a lightcurve by calling the other methods for each orientation
        of the dataset. Sort of a main method.

        TODO: Complete docstring.
        """

        # Default setting for theta is a full revolution:
        if theta is None:
            if unit == "rad":
                theta = 2*np.pi
            elif unit == "deg":
                theta = 360.
            elif unit == "arcmin":
                theta = 360. * 60
            elif unit == "arcsec":
                theta = 360. * 3600

        if inclinations is None:
            inclinations = self.inclinations
        # If the inclination is a single number, put it in a list:
        inclinations = func.to_list(inclinations)
```

```
        if H is None:
            H = self.H

        if n_angle is None:
            n_angle = int(round(float(theta) / dtheta))
        elif dtheta is None:
            dtheta = float(theta) / n_angle

        angles = np.linspace(0, theta-dtheta, n_angle)
        lightcurve = np.zeros((len(inclinations), n_angle))

        for i, angle in enumerate(angles):
            print "%f / %f" % (angle, theta)
            self.rotate(
                angle_z=angle,
                unit=unit,
            )
            for k, star in enumerate(self.stars):
                sylinder = Sylinder(
                    star=star,
                    data=self.get_sylinder(star=star),
                    unit=self.unit,
                    radius_in=self.radius_in,
                    radius_out=self.radius_out,
                    kappa=self.kappa
                )
                for j, inclination in enumerate(inclinations):
                    sylinder.space_sylinder(
                        inclination=inclination,
                        unit=unit,
                        H=H,
                        n_steps=n_radius,
                        dr=dr,
                    )
                    lightcurve[j, i] += sylinder.integrate()
        print "%f / %f" % (theta, theta)

        if "all" in normalizations:
            normalizations = ["stellar", "max", "mean"]

        for normalization in normalizations:

            if "stellar" in normalization or "unobscured" in normalization:
                unobscured_flux = 0.
                for star in self.stars:
                    unobscured_flux += star.intensity
                lightcurve /= unobscured_flux
            elif "max" in normalization:
                lightcurve /= lightcurve.max(axis=1)[:, None]
            elif "mean" in normalization:
                lightcurve /= lightcurve.mean(axis=1)[:, None]

            if show or savefig:
                fig = plt.figure(figsize=(12,6))
                fig.gca().get_yaxis().get_major_formatter().set_useOffset(False)
                    # Always use absolute labels and not offsets.
                plt.minorticks_on()  # Turn default minorticks on for y-axis.
                ax = fig.add_subplot(1,1,1)
                ax.set_xlim([0, 360])
                ax.set_xticks(range(0, 360+1, 30))
                ax.xaxis.set_minor_locator(ticker.AutoMinorLocator(3))
                    # A major tick every 30 deg and minor tick every 10 deg.

                # Set a specific tick step (hardcoded switch):
                if False:
                    step = 0.000001
                    stepmin = lightcurve.min()
                    stepmax = lightcurve.max()
                    stepdiff = stepmax - stepmin
```

```
            stepmin -= stepdiff  # Increase the range of the ticks just
            stepmax += stepdiff  # to be sure.
            stepmin = step * round(stepmin / step)  # Round off to the
            stepmax = step * round(stepmax / step)  # nearest step.
            ax.set_yticks(np.linspace(
                stepmin,
                stepmax,
                int(round((stepmax - stepmin) / step)) + 1,
            ))
            # Set a specific label step (hardcoded switch):
            if True:
                # Automatic minor ticks between steps.
                ax.yaxis.set_minor_locator(ticker.AutoMinorLocator(5))
            elif True:
                # Manual label removal of certain steps.
                steplabel = 2 * step
                ylabels = []
                for ytick in ax.get_yticks():
                    if (int(round(ytick/step)) %
                        int(round(steplabel/step)) == 0
                    ):
                        ylabels.append(ytick)
                    else:
                        ylabels.append("")
                ax.set_yticklabels(ylabels)

    for j, inclination in enumerate(inclinations):

        starradius = ""
        starflux = ""
        for star in self.stars:
            starradius += "%g-" % star.radius
            starflux += "%g-" % star.intensity
        starradius = starradius.rstrip("-")
        starflux = starflux.rstrip("-")
        header = (
            "%s, H=%g, kappa=%g, "
            "r_star=%s, flux_star=%s, r_in=%g, r_out=%g, dr=%g, "
            "dtheta=%g%s, inc=%g%s"
            % ( self.dataname,
                H,
                self.kappa,
                starradius,
                starflux,
                self.radius_in,
                self.radius_out,
                (self.radius_out - self.radius_in) / n_radius,
                float(theta) / n_angle,
                unit,
                inclination,
                unit,
            )
        )
        if save:
            outname = (
                "%s__H=%g__"
                "r_in=%g__r_out=%g__"
                "inc=%02g__%snorm"
                % ( self.dataname,
                    H,
                    self.radius_in,
                    self.radius_out,
                    inclination,
                    normalization,
                )
            )
            if outfolder is None:
                outfolder = self.outfolder
            func.make_folder(outfolder)
            outfile = open("%s/%s.csv" % (outfolder, outname), "w")
```

```
                    outfile.write("#" + header + "\n")
                    for angle, flux in zip(angles, lightcurve[j]):
                        outfile.write("%f,%f\n" % (angle, flux))
                    outfile.close()

                if show or savefig:
                    ax.plot(
                        angles,
                        lightcurve[j],
                        label="%2g" % inclinations[j],
                    )

            if show or savefig:
                if short_title:
                    # Only use the name of the data in the title.
                    ax.set_title(self.dataname)
                    ax.set_position([0.10, 0.10, 0.80, 0.83])
                else:
                    # Use all metadata in the title.
                    ax.set_title(
                        "\n".join(textwrap.wrap(header.split(", inc")[0], 70))
                    )
                    ax.set_position([0.10, 0.10, 0.80, 0.80])
                ax.set_xlabel("viewing angle [degree]")
                ax.set_ylabel("flux [%s flux]" % normalization)
                ax.legend(
                    bbox_to_anchor=(1.11, 0.5),
                    title="inc [deg]=",
                    loc="right",
                    borderaxespad=0.,
                )

            if savefig:
                outname = (
                    "%s__H=%g__"
                    "r_in=%g__r_out=%g__"
                    "%snorm"
                    % ( self.dataname,
                        H,
                        self.radius_in,
                        self.radius_out,
                        normalization,
                    )
                )
                if outfolder is None:
                    outfolder = self.outfolder
                func.make_folder(outfolder)
                fig.savefig("%s/%s.png" % (outfolder, outname))

        if show:
            plt.show()


class Sylinder(DensityMap):


    def __init__(self,
        star,
        data,
        unit=None,
        inclinations=None,
        radius_in=0,
        radius_out=np.inf,
        kappa=10.
    ):

        # If the inclination is a single number, put it in a list:
        try:
            iter(inclinations)
```

```
            self.inclinations = inclinations
        except TypeError:
            self.inclinations = [inclinations]
        self.star = star
        self.unit = unit
        self.radius_in = radius_in
        self.radius_out = radius_out
        self.kappa = kappa  # [cm^2 / g]

        self.data = data


    def space_sylinder(self,
        inclination=0,
        unit="deg",
        H=1.,
        n_steps=None,
        dr=None,
    ):
        """Bin this sylinder's datapoints into a set of mean densities.

        The sylinder is first sorted along the x-axis and is then cut along
        the x-axis like a loaf of bread. Each point is integrated
        analytically through its projected density from the bottom to the
        top of the sylinder. The mean density is then computed from each
        slice of the sylinder/bread.

        This is stored temporarily as self.densities and self.drs: (float,
        array), (float, array) A list of mean densities and the
        corresponding list of delta radiuses for each bin. Both are arrays
        of length n_step. The arrays are order FROM inside of disk TO
        oustide of disk.

        inclination: (float) The angle to incline the line of sight on the
            sylinder.
        deg: (string) Unit of the angle.
        H: (float) Thickness of the disk. Necessary for integral.
        n_steps: (int) How many slices to divide the sylinder in. Affects
            accuracy of integral.
        dr: (float) The width of each sylinder section. Ignored if n_steps
            is provided.

        """

        if unit == "rad":
            factor = 1.
        elif unit == "deg":
            factor = np.pi / 180.
        elif unit == "arcmin":
            factor = np.pi / 180. * 60
        elif unit == "arcsec":
            factor = np.pi / 180. * 3600
        inclination *= factor

        if n_steps is None:
            n_steps = int(round((self.radius_out - self.radius_in) / dr))
        dpoints = int(round(self.data.shape[0] / float(n_steps)))
            # How many datapoints to include in each bin.

        densities = np.zeros(n_steps)
        drs = np.zeros(n_steps)

        data = self.data[np.argsort(self.data[:, 0])]

        g = np.sqrt(2) * H  # Constant used several times in calculations.

        for i in xrange(n_steps):
            start = i*dpoints
            if i == n_steps-1:
                # If it is the last step, make sure the last few points are
```

```
                # included (in case there are some rounding problems).
                end = data.shape[0]
                drs[i] = data[end-1, 0] - data[start, 0]
                s = data[start:end].shape[0]
                drs[i] *= (s + 1.) / s
            else:
                end = (i+1)*dpoints
                drs[i] = data[end, 0] - data[start, 0]
            W = np.sqrt(
                self.star.radius**2 -
                (data[start:end, 1] - self.star.position[1])**2
            ) / np.cos(inclination)
            z = (
                (data[start:end, 0] - self.star.position[0]) *
                np.tan(inclination)
            )
            z1 = z - W
            z2 = z + W
            densities[i] = (
                np.sum(
                    # \int_z1^z2 \rho_0 * e^{- z^2 / (2*H^2)} dz
                    g * data[start:end, ~0] * 0.5 * np.sqrt(np.pi) *
                    (special.erf(z2 / g) - special.erf(z1 / g))
                ) / (2. * np.sum(W))
            )

        self.densities = densities
        self.drs = drs


    def integrate(self):
        """Integrates the intensity through the layers of dust.

        Assumes that space_sylinder has just been called and used its results.

        return: (float) Perceived intensity outside the disk
        """

        kappa = self.kappa * u.Unit("cm2/gram").to(
            u.Unit(self.unit["distance"])**2 / u.Unit(self.unit["mass"])
        )
        intensity = self.star.intensity

        for density, dr in zip(self.densities, self.drs):
            tau = kappa * density * dr
            intensity *= np.exp(-tau)

        return intensity * (u.Unit(self.unit["intensity"])).to("erg / (cm2 s)")
```

9.4 Star.py

The simple `Star` class whose intention is to hold the physical parameters of each star.

```
import numpy as np

import Functions as func



class Star:


    def __init__(self, d=None, position=None, radius=None, intensity=None):
        """Make a star instance.
```

```
        d: (dictionairy) Must contain position, radius and intensity and can
            be provided instead of giving these other arguments individually.
        position: (float, array-like) Coordinates of the star.
        radius: (float) Radius of the star.
        intensity: (float) Intensity of the star.
        """

        if d is not None:
            try:
                position = np.array([
                    float(d["position"]["x"]),
                    float(d["position"]["y"]),
                ])
            except KeyError, ValueError:
                position = np.array(func.pol2cart(
                    float(d["position"]["r"]),
                    float(d["position"]["theta"]),
                ))
            radius = float(d["radius"])
            intensity = float(d["intensity"])

        self.position = np.array(position)
        self.radius = radius
        self.intensity = intensity
```

9.5 Functions.py

A file containing some general functions that are used other places in the program.

```
import os
import numpy as np


def to_list(x, dtype=None, separator=" "):
    """Converts any sequence or non-sequence into an array.

    The use of a numpy array is primarely used because then it is easy to
    also convert the type. Except for that it could just as well be a list.

    If x is a string, split the string with the separator.
    If x is a sequence, convert it into an array.
    If x is a non-sequence, put it into a size-1 array.

    X: (anything)  Something to be converted into an array.
    dtype: (type) What type of objects the array contains. F.ex. float. If
        None, numpy will interpret the type itself.
    separator: (string) If x is a string to be split, this is the separator.
        Usually a space or comma.
    """

    if isinstance(x, basestring):  # If x is a string.
        x = np.array(x.split(separator), dtype=dtype)
    elif isinstance(x, dict):  # If x is a dictionairy.
        x = np.array([x], dtype=dtype)
    else:
        try:
            iter(x)  # If x is a sequence.
            x = np.array(x, dtype=dtype)
        except TypeError:  # If x is a non-sequence.
            x = np.array([x], dtype=dtype)

    return x


def cart2pol(x, y):
    """Convert cartesian coordinates into polar coordinates."""
```

```
    r = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)
    return [r, theta]


def pol2cart(r, theta):
    """Convert polar coordinates into cartesian coordinates."""
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return [x, y]


def make_folder(directory, warning=False):
    """Make the directory, but do nothing if it already exists. Optionally
    print a warning if it already exists, if there should be a reason for
    that.
    """
    if not os.path.exists(directory):
        os.makedirs(directory)
    else:
        if warning:
            print "%s already exists." % (directory)
```

9.6 plot.py

A standalone script that can be used to plot the output of `CirBinDis` . You can just as well use f.ex. TOPCAT.

```
import sys
import numpy as np
import matplotlib.pyplot as plt


def load(filename, inclination, separator=","):
    """Load a dataset to plot from a file.

    Assumed to be on the form 'angle,flux'.
    The first 3 lines are the title, xlabel and ylabel.

    filename: (string) or (string, list) Full pathname to file containing
        dataset or a list of pathnames.
    separator: (string) This is the separator between the values each line.
        Usually a space or comma.
    """

    if isinstance(filename, basestring):
        filenames = [filename]
    else:
        try:
            iter(filename)
            filenames = filename
        except TypeError:
            raise TypeError(
                "'filename' must be a string or sequence of strings."
            )

    if isinstance(inclination, basestring):
        inclinations = inclination.split(" ")
    else:
        try:
            iter(inclination)
            inclinations = inclination
        except TypeError:
            raise TypeError(
                "'filename' must be a string or sequence of strings."
            )
```

```
    infiles = [open(filename, "r") for filename in filenames]

    titles = [infile.readline() for infile in infiles]
    xlabels = [infile.readline() for infile in infiles]
    ylabels = [infile.readline() for infile in infiles]
    title = titles[0]
    xlabel = xlabels[0]
    ylabel = ylabels[0]

    i = 0
    for infile in infiles:
        data = []
        for line in infile:
            line = [float(value) for value in line.split(separator)]
            data.append(line)
        angles, fluxes = np.array(data).T
        infile.close()
        plt.plot(angles, fluxes, label="inc="+inclinations[i])
        i += 1

    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.legend()
    plt.show()


if __name__ == "__main__":

    load(filename=sys.argv[2:], inclination=sys.argv[1])
```

9.7 make_testdata.py

A standalone script that can be used to generate artificial datasets that can be analysed by `CirBinDis` . You can use it for testing and for generating data according to any analytical function that you would like to analyse (then you need to change the function `density`).

```
"""This script can be used to make an artificial density map with a smooth
sinusoidal shape. This can be used to test the analysis program.
"""

import numpy as np
import cPickle as pickle


Nx = int(6e2 + 1)
Ny = int(6e2 + 1)
strip_x = np.linspace(-4, 4, Nx)
strip_y = np.linspace(-4, 4, Ny)

x = np.zeros(Nx*Ny)
y = np.zeros(Nx*Ny)
z = np.zeros(Nx*Ny)

for i in xrange(Ny):
    x[i*Nx : (i+1)*Nx] = strip_x
for i in xrange(Nx):
    y[i :: Nx] = strip_y

def density(x, y):
    return (1. + np.sin(2*np.arctan2(y, x)))

def add_density(x, y):
```

```
    return np.vstack((x, y, z, density(x, y))).transpose()

data = add_density(x, y)

outfile = open("../data/testdata.p", "wb")
pickle.dump(data, outfile)
outfile.close()
```