`CirBinDis` Circumbinary disk analyser

Paul Magnus Sørensen-Clark
Jerome Bouvier

March 18, 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

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.

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` .

(It is currently possible to have four instead of three. Then the three first columns represent (x, y, z) or (r, θ, z). The use of 3D-datapoints is currently an obsolete feature that will probably be completely removed.)

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.

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

For each bin a mean density is produced from all the datapoints in that bin. Each point is also projected vertically up from the disk (orthogonal to the disk plane) and into the area it represents in a sylinder that is inclined with respect to the disk.

$\rho(z)$ is the assumed density at a point with altitude z above a point in the disk plane with density ρ_0.

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

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 area than points near the edges. (**TODO:** There needs to be a figure of this to illustrate it more cleary.)

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

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.

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

$\exp(-z^2)$ is the gaussian. It's integral has no analytical solution, but it can be approximated by the error function (erf), which is a power series. erf is provided in the Python library SciPy (`scipy.special.erf`) and can handle arrays, which means that this integral can be performed for all the datapoints in the sylinder bin very quickly in a vectorized manner.

$$\begin{aligned} \lambda_i &= \int_{z_{i,a}}^{z_{i,b}} \rho_0 \exp\left(-\frac{z_i^2}{2H^2}\right) \, \mathrm{d}z_i \\ &= \rho_0 \frac{\sqrt{\pi}}{2} \sqrt{2H^2} \left[\mathrm{erf}\left(\frac{z_{i,b}}{\sqrt{2H^2}}\right) - \mathrm{erf}\left(\frac{z_{i,a}}{\sqrt{2H^2}}\right) \right] \\ \rho_{bin} &= \frac{\sum_i \lambda_i}{\sum_i 2W_i} \end{aligned}$$

ρ_{bin} is thus the mean density of one bin of the sylinder, and it has a corresponding Δr. A ρ_{bin}, Δr pair is calculated for each of the N_{bin} bins in each sylinder.

4.7 Integrating

TODO

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. Inside the repository you will find `/xml/input.xml`. Copy this file to the path where you will run `CirBinDis` from (normally the folder where the input datafile is) and change the value of the fields as required (it must still be called `input.xml`). Specifically, this is where you provide the filename of the dataset to analyse (if `input.xml` is in another folder than the input datafile you need to provide the relative path).

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

unit-mass Dorem

unit-distance ipsum

unit-intensity dolum

5.2 Executing the code

When you have prepared `input.xml` and you are located in it's folder, type the command `cirbindis`
(or `python ~/path_to_repository_folder/circumbinarydisk/src/main.py`). Then the software will run until it has completed the analysis of your datafile with all the parameters you specified.

5.3 Output

TODO

6 Bibliography

7 Acknowledgments

8 Source code

8.1 `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_cropped.p</datafile>
    <system>cartesian</system>
        <!--Examples:  cartesian  |  polar -->

    <!--==========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</radius_in>
        <!--Examples:  1  |  1.5  |  1 1.25 1.5 -->
    <radius_out>3</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.
        Opacity, in units of [cm^2/g].
        Thickness of disk.
    -->
    <diskmass>0.01</diskmass>
    <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>
```

8.2 main.py

The script that is called when running `CirBinDis` .

```
import xmltodict
```

```
from DensityMap import DensityMap
import Functions as func


if __name__ == "__main__":

    infile = open("input.xml", "r")
    input_ = xmltodict.parse(infile)["input"]
    infile.close()

    for radius_in in func.to_list(input_["radius_in"], float):
        for radius_out in func.to_list(input_["radius_out"], float):
            dataset = DensityMap(
                filename=input_["datafile"],
                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"]),
                H=float(input_["H0"]),
                kappa=float(input_["kappa"]),
            )
            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"],
                show=True,
            )
```

8.3 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 matplotlib.pyplot as plt
    # For plotting results.
from scipy import integrate
from scipy import special
import astropy.units as u

from Star import Star
import Functions as func



class DensityMap:


    def __init__(self,
        data=None,
        filename=None,
        coordsystem="cartesian",
        unit=None,
        inclinations=None,
        radius_in=0,
        radius_out=np.inf,
        diskmass=.01,
        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]
        self.unit = unit
        self.stars = []
        self.radius_in = radius_in
        self.radius_out = radius_out
        self.diskmass = diskmass
        self.H = H
        self.kappa = kappa  # [cm^2 / g]
            # Between 5 and 100 according to Boubier et al. 1999.

        if data is not None:
            self.data = data

        elif filename is not None:
            self.load(filename)

        # Convert density to physical units related to the total mass and
        # size of the disk:
        self.data[:, ~0] /= self.data[:, ~0].mean()
        self.data[:, ~0] *= (
            self.diskmass / (np.pi * self.radius_out**2 * 2*self.H)
        )

        if coordsystem == "cartesian":
            pass
        elif coordsystem == "polar":
            x, y = func.pol2cart(self.data[:, 0], self.data[:, 1])
        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,
        separator=" ",
    ):
        """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.
        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 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 = line.rstrip().split(separator)
                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 writeto(self, filename, method="pickle", 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.
        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 == "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 set_r0(self, r0=1.49597871e13):
        self.r0=r0  # [centimeters]

    def set_H(self, H0=0.03, H_power=1/4.):
        self.H0 = H0
        self.H_power = H_power

    def get_H(self, r):
        return r * self.H0 * (r / self.r0)**self.H_power  # [centimeters]

    def set_sigma(self, sigma0=1700., sigma_power=-3/2.):
        self.sigma0 = sigma0
        self.sigma_power = sigma_power

    def get_sigma(self, r):
        return self.sigma0 * (r / self.r0)**self.sigma_power  # [g / cm^2]


    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

        # 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,
        show=False,
    ):
        """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)

        lightcurve /= lightcurve.mean(axis=1)[:, None]

        for j, inclination in enumerate(inclinations):

            title = (
                "H=%g, kappa=%g, "
                "r_star=%g, r_in=%g, r_out=%g, dr=%g, "
                "dtheta=%g%s, inclination=%g%s"
                % ( H,
                    self.kappa,
                    #TODO One for each star.
                    self.stars[0].radius,
                    self.radius_in,
                    self.radius_out,
                    (self.radius_out - self.radius_in) / n_radius,
                    float(theta) / n_angle,
                    unit,
                    inclination,
                    unit,
                )
            )
            xlabel = "rotation angle [%s]" % unit
            ylabel = "observed flux"

            if save:
                outfile = open("../results/%s.csv" % title.replace(", ", "__"), "w"
                    )
                outfile.write(title + "\n")
                outfile.write(xlabel + "\n")
                outfile.write(ylabel + "\n")
                for angle, flux in zip(angles, lightcurve[j]):
                    outfile.write("%f,%f\n" % (angle, flux))
                outfile.close()

            if show:
                plt.plot(
                    angles,
                    lightcurve[j],
                    label="inc=%g" % inclinations[j],
                )

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



class Sylinder(DensityMap):
```

```python
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
    """

    intensity = self.star.intensity

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

    return intensity
```

8.4 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
```

8.5 Functions.py

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

```
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 because then it is easy to also convert
        the type. Except for that it couls 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, but 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 list to be splitted, this is the
        separator. Normally a space or comma.
    """

    if isinstance(x, basestring):  # If x is a string.
        x = np.array(x.split(separator), 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]
```

8.6 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])
```

8.7 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()
```