Akamai Polarization Modeling Documentation

Release 1.0

Kyubey

Aug 11, 2018

CONTENTS:

This is the documentation of a simulation program that simulates observations of magnetic morphologies of star-forming regions. Modeling arbitrary and custom 3D cloud shape geometries, integrating along sightlines over user-inputted magnetic fields, density profiles, and polarization functions.

Overall, this codebase would return the user back the values of Stokes parameters for the given object that they inputted. It will also, by default, plot the data to contour maps in the form of heat maps that visually describe the object that the user input into the simulation.

Written apart of a Akamai Workforce Initiative Summer internship.

Written in Python 3.6, this package also relies on the following external Python packages:

- Numpy
- Scipy
- Sympy
- Matplotlib
- Astropy

However, it is suggested to use a standard Anaconda Distribution instead of manually installing the packages in the event of conflicts.

CHAPTER

ONE

QUICKSTART

This is a brief quickstart introduction to this module/code. This highlights the main purposes of the code, providing examples and other references.

It is assumed that the program/code has been installed along with its inherent dependencies. If not, then consult the main page for more information.

The core of this module is the *ObservingRun* object. This object, as the name implies, is the class that acts as the observation run of the simulation.

In order to make the *ObservingRun* object, we first need two other objects: *ProtostarModel* and *Sightline* objects.

1.1 ProtostarModel Creation

First, let us make the *ProtostarModel* object. First off there are five things that we need to provide in order to make such an object (although two are optional).

Please note that the coordinate system for the equations and functions within this section is where the origin of the object is the origin of the coordinate system, with the x-axis on the light of sight, positive going from the observing telescope, to the origin of the object, deeper into space, and thus is equivalent to the r-axis from Earth. The y-z plane is assumed to be analogous to the RA-DEC plane of the sky.

The five things that are required for the *ProtostarModel* is a SkyCoord, a cloud function, a magnetic field function, a density function, and a polarization function.

1.1.1 Coordinates

The first parameter is the coordinates of the object in the sky. This is supposed to simulate actual observations, and thus a real sky coordinate is the accepted term. Consider the example parameters below for an object that is at the spring equinox (i.e. a RA of 00h00m00.00s and a DEC of 00d00m00.00s).

```
import astropy as ap
import astropy.coordinates as ap_coord

# Making the coordinate input, should be an Astropy SkyCoord class.
sky_coordinates = ap_coord.SkyCoord('00h00m00.00s','00d00m00.00s',
                                    frame='icrs')
```

1.1.2 Cloud Equation

The cloud equation must be a function that takes in three dimensions and is the expression of the implicit functional form. In a general sense, the function is programed as $f(x, y, z) \rightarrow$ return, where the expression $f(x, y, z)$ is from the mathematical implicit equation $f(x, y, z) = 0$.

```
# Making a cloud function, a sphere in this case. Note that the units
# are in angular space, and thus the unit of circle is radians.
def cloud_equation(x,y,z):
    radius = 0.01
    return x**2 + y**2 + z**2 - radius**2
```

1.1.3 Magnetic Field

The magnetic field is a little bit more complex, this function is not a scalar function like the density or polarization functions. It is a vector function and thus must return three vectors in cartesian space, still with respect to the origin of the protostar model. Consider the following field that is uniform in one direction. More complex field geometries can be found in *magnetic_field_functions_3d* and can be called from there.

Please note the use of Numpy's structures. Although it is possible to return only integers for the components, using these structures is required if table interpolation is used, and thus usage is suggested for compatibility purposes.

```
import numpy as np

# Making a magnetic field that is uniform in one direction. Consider a
# field that is always 0i + 1j + 2k.
def magnetic_field(x,y,z):
    Bfield_x = np.zeros_like(x)
    Bfield_y = np.ones_like(y)
    Bfield_z = np.full_like(z,2)

    return Bfield_x,Bfield_y,Bfield_z
```

1.1.4 Density Function

The density function is a scalar function in three dimensional space. As such, it should return a density value given a three dimensional input ($f(x, y, z) = d$). Consider the example function of a density profile that drops off as a function of $\frac{1}{r^2}$.

```
import numpy as np

# Making a density function of a 1/r**2 profile.
def density_function(x,y,z):
    density = 1/np.dot([x,y,z],[x,y,z])

    # The above line is a faster implementation of the following.
    # density = 1/np.dot(x**2 + y**2 + z**2)

    return density
```

1.1.5 Polarization Function

The polarization function, like the density function is a scalar function in three dimensional space. As such, it should return a value which corresponds to the level of polarization of the light at that given location in space ($f(x, y, z) = p$). Consider the example function of a polarization profile that drops off as a function of r^2.

```
import numpy as np

# Making a polarization function of a 1/r**2 profile.
def polarization_function(x,y,z):
    polarization = np.sqrt(np.dot([x,y,z],[x,y,z]))

    # The above line is a faster implementation of the following.
    # polarization = np.sqrt(x**2 + y**2 + z**2)

    return polarization
```

1.1.6 Creating the Class

From these parameters, a *ProtostarModel* by the following line.

```
import model_observing as m_obs

# Create the protostar class.
protostar = m_obs.ProtostarModel(sky_coordinates,
                                 cloud_equation,
                                 magnetic_field,
                                 density_function,
                                 polarization_function)
```

Or, all in one block of code.

```
import numpy as np
import astropy as ap
import astropy.coordinates as ap_coord

import model_observing as m_obs

# Making the coordinate input, should be an Astropy SkyCoord class.
sky_coordinates = ap_coord.SkyCoord('00h00m00.00s','00d00m00.00s',frame='icrs')

# Making a cloud function, a sphere in this case. Note that the units
# are in angular space, and thus the unit of circle is radians.
def cloud_equation(x,y,z):
    radius = 0.01
    return x**2 + y**2 + z**2 - radius**2

# Making a magnetic field that is uniform in one direction. Consider a
# field that is always 0i + 1j + 2k.
def magnetic_field(x,y,z):
    Bfield_x = np.zeros_like(x)
    Bfield_y = np.ones_like(y)
    Bfield_z = np.full_like(z,2)

# Making a density function of a 1/r**2 profile.
def density_function(x,y,z):
```

(continues on next page)

(continued from previous page)

```
    density = np.sqrt(np.dot([x,y,z],[x,y,z]))

    # The above line is a faster implementation of the following.
    # density = np.sqrt(x**2 + y**2 + z**2)

    return density

# Making a polarization function of a 1/r**2 profile.
def polarization_function(x,y,z):
    polarization = np.sqrt(np.dot([x,y,z],[x,y,z]))

    # The above line is a faster implementation of the following.
    # polarization = np.sqrt(x**2 + y**2 + z**2)

    return polarization

# Create the protostar class.
protostar = m_obs.ProtostarModel(sky_coordinates,
                                 cloud_equation,
                                 magnetic_field,
                                 density_function,
                                 polarization_function)
```

1.2 Sightline Creation

When the object is made, it would also be helpful to actually look at the object (simulated observations). Thus, there is a `Sightline` class. The purpose of this class is to simulate the telescope's pointing location.

The `Sightline` class takes in two strings for the RA and DEC of the object. They should be in the following format:

- RA: ##h##m##.##s (i.e 12h05m10.00s)
- DEC: ##d##m##.##s (i.e 06d18m10.25s)

Note that the seconds may have more decimals for an accuracy greater than the hundredths place.

If we would like to observe the object that was created in the previous step in _ProtostarModel_Creation, it is best to also point to the object. Therefore, we expect to point to the same location in the sky.

The code below generates a `Sightline` class just to do that.

```
import astropy as ap
import astropy.coordinates as ap_coord

import model_observing as m_obs

# RA of 00h00m00.00s and a DEC of 00d00m00.00s
sightline = m_obs.Sightline('00h00m00.00s','00d00m00.00s')
```

Note that the class can also accept a `SkyCoord` class object. This functionality is done to improve the compatibility with other RA-DEC notations. This alternative method of creating a Sightline is demonstrated below. Because the SkyCoord object contains all of the information needed, the strings that the user would have normally input is ignored in favor for the SkyCoord object.

```
import astropy as ap
import astropy.coordinates as ap_coord
```

(continues on next page)

(continued from previous page)

```
import model_observing as m_obs

# Making the SkyCoord class with a RA of 00h00m00.00s and a
# DEC of 00d00m00.00s
sky_coordinates = ap_coord.SkyCoord('00h00m00.00s','00d00m00.00s',
                                    frame='icrs')

# Creating the Sightline class using the SkyCoord class.
sightline = m_obs.Sightline(None, None, SkyCoord_object=sky_coordinates)
```

1.3 ObservingRun Creation

When both the *ProtostarModel* object and the *Sightline* object is created, an *ObservingRun* object can be made using both of them.

An *ObservingRun* object is an object that simulates the act of doing an observing run with a telescope (as the name implies). Through its member functions, the class allows for the computation of different observation run styles.

To make an ObservingRun object, it can be made as follows.

```
import model_observing as m_obs

# Define the field of view of the observation, in radians as the total
# length of the observation square.
field_of_view = 0.015

observing_run = m_obs.ObservingRun(protostar,sightline,field_of_view)
```

From the *ObservingRun* object, the following observations can be completed from it.

1.3.1 Model Stokes Parameters

Modeling Stokes parameters in plots is the primary function (as of current) of this repository codebase. It can be normally called by executing the member function *Stokes_parameter_contours()*. The most basic execution of this method is as follows.

```
import model_observing as m_obs

# Decide on the resolution of the data observed, this sets the number of
# data points on one axis.
axis_resolution = 30

results = observing_run.Stokes_parameter_contours(
    n_axial_samples=axis_resolution)
```

The value of the returned function is a list of the RA-DEC values for the sampled points along with the Stokes parameters from a sightline at those given RA-DEC values. Refer to the method documentation (see *Stokes_parameter_contours()*) for more information.

It should be the case that the plots of the object desired is created with this method. The method itself should already plot the information for you in heat-map based contours.

Warning: If the values of the intensity is highly similar across the field of view, it may be the case that the colorbar readjustments fatally fails. To prevent this, it is suggested to choose one's field of view such that at least one sightline misses the object.

CHAPTER

TWO

EXAMPLE

This is the example from the ~.quickstart guide function.

The combined code block of that guide is below.

```
import numpy as np
import astropy as ap
import astropy.coordinates as ap_coord

import model_observing as m_obs

# Making the coordinate input, should be an Astropy SkyCoord class.
sky_coordinates = ap_coord.SkyCoord('00h00m00.00s','00d00m00.00s',frame='icrs')

# Making a cloud function, a sphere in this case. Note that the units
# are in angular space, and thus the unit of circle is radians.
def cloud_equation(x,y,z):
    radius = 0.01
    return x**2 + y**2 + z**2 - radius**2

# Making a magnetic field that is uniform in one direction. Consider a
# field that is always 0i + 1j + 2k.
def magnetic_field(x,y,z):
    Bfield_x = np.zeros_like(x)
    Bfield_y = np.ones_like(y)
    Bfield_z = np.full_like(z,2)

    return Bfield_x,Bfield_y,Bfield_z

# Making a density function of a 1/r**2 profile.
def density_function(x,y,z):
    density = 1/np.dot([x,y,z],[x,y,z])

    # The above line is a faster implementation of the following.
    # density = 1/(x**2 + y**2 + z**2)

    return density

# Making a polarization function of a 1/r**2 profile.
def polarization_function(x,y,z):
    polarization = 1/np.dot([x,y,z],[x,y,z])

    # The above line is a faster implementation of the following.
    # polarization = 1/(x**2 + y**2 + z**2)
```

(continues on next page)

(continued from previous page)

```
    return polarization

# Create the protostar class.
protostar = m_obs.ProtostarModel(sky_coordinates,
                                 cloud_equation,
                                 magnetic_field,
                                 density_function,
                                 polarization_function)


# Creating the Sightline class using the SkyCoord class.
sightline = m_obs.Sightline('00h00m00', '00d00m00')


# Define the field of view of the observation, in radians as the total
# length of the observation square.
field_of_view = 0.015

observing_run = m_obs.ObservingRun(protostar,sightline,field_of_view)

# Decide on the resolution of the data observed, this sets the number of
# data points on one axis.
axis_resolution = 30

results = observing_run.Stokes_parameter_contours(
    n_axial_samples=axis_resolution)
```

CHAPTER

THREE

ACTIVE_CODEBASE

3.1 Backend package

3.1.1 Submodules

Backend.Ewertowski_Basu_2013 module

This file is most of the backend functionality that is required for the more important functions within this module that relates to the paper by Bartek Ewertowshi and Shantanu Basu on their work in 2013 on "A Mathematical Model for an Hourglass Magnetic Field".

The functions here are almost explicitly for magnetic field functions.

Backend.Ewertowski_Basu_2013.**Ewer_Basu__B_r**(*r*, *z*, *h*, *k_array*, *disk_radius*)

An implementation of Eq 45 of Ewertiwski & Basu 2013.

This implements equation 45 of Ewertiwski & Basu 2013. The k_array (values of k) determine the number of summation terms that will be computed. The value of r is with respect to the cylindrical cordinate system.

Parameters

- **r** (*array_like*) – The input values of the radial direction for the equation.
- **z** (*array_like*) – The input values of the polar direction for the equation.
- **h** (*float*) – A free parameter as dictated by the paper.
- **k_array** (*array_like*) – The list of k coefficient values for the summation in Eq 45.
- **disk_radius** (*float*) – The radius of the protoplanetary disk. Relevent for the hourglass magnetic field generated by this paper.

Returns **Bfield_r** – The value of the magnetic field in the r-axial direction.

Return type ndarray

Backend.Ewertowski_Basu_2013.**Ewer_Basu__B_z**(*r*, *z*, *h*, *k_array*, *disk_radius*, *uniform_B0*)

An implementation of Eq 46 of Ewertiwski & Basu 2013.

This implements equation 46 of Ewertiwski & Basu 2013. The k_array (values of k) determine the number of summation terms that will be computed. The value of r is with respect to the cylindrical cordinate system.

Parameters

- **r** (*array_like*) – The input values of the radial direction for the equation.
- **z** (*array_like*) – The input values of the polar direction for the equation.
- **h** (*float*) – A free parameter as dictated by the paper.

- **k_array** (*array_like*) – The list of k coefficient values for the summation in Eq 45.
- **disk_radius** (*float*) – The radius of the protoplanetary disk. Relevent for the hourglass magnetic field generated by this paper.
- **uniform_B0** (*float*) – The magnitude of the background magnetic field.

Returns **Bfield_z** – The value of the magnetic field in the z-axial direction.

Return type ndarray

Backend.Ewertowski_Basu_2013.**Ewer_Basu__eigenvalues**(*index_m*, *disk_radius*)

This is the values of the eigenvalues of some integer index m.

This computes the eigenvalues (see Equation 28 of Ewer & Basu 2013) of the normalized Green functions for the hourglass magnetic field solution.

Parameters

- **index_m** (*int*) – This is the index of the eigenvalue, it links up to the index of the positive root of a Bessel function of the first kind of order 1.
- **disk_radius** (*float*) – The radius of the protoplanetary disk. Relevent for the hourglass magnetic field generated by this paper.

Returns **eigenvalue** – The value of the corresponding eigenvalue given the parameters.

Return type float

Backend.Ewertowski_Basu_2013.**bessel_zeros**(*order*, *index*)

Returns a Bessel zero given the function's order and zero index.

This function is a wrapper around Scipy's Bessel function zero generator. For some reason, the Scipy function returns all zeros between the given number and the first zero. This function extracts only the wanted zero given the index.

This is for Bessel functions of the first kind.

Parameters

- **order** (*int*) – The integer order of the Bessel function of the first kind.
- **index** (*int*) – The zero's index that is desired.

Returns **zero** – The value of the zero at the given index of the given Bessel function.

Return type float

Backend.astronomical_coordinates module

This file is mostly the calculations required for corrections to the astrophysical coordinate system, and any calculations required for such is recorded here.

Backend.astronomical_coordinates.**angle_normalization_0_2pi**(*angle*)

Automatically normalize angle value(s) to the range of 0-2pi.

This function relies on modular arithmetic.

Parameters **angle** (*array_like*) – The angles to be converted

Returns **normalized_angles** – The angles after being normalized to be between 0-2pi.

Return type ndarray

Backend.astronomical_coordinates.**auto_ra_wrap_angle**(*object_ra_value*)

Automatically calculate the RA wrap value.

Given an input RA, this function automatically calculates the RA wrap angle to likely be used for an Astropy `SkyCoord` object.

Parameters **object_ra_value** (*array_like*) – The RA value(s) that will determine the wrap angle(s).

Returns **ra_wrap_angle** – The value of the wrap angle that the function best described.

Return type ndarray

Notes

In this function, it assumes that there are only 4 main quadrants. First, the wrap angle is determined to be either at 0 or π depending on the location of the sightline's RA.

Backend.cloud_line_integration module

This module outlines the math side of the integration functions used. The cloud line integration computes a line integral along a sightline path, whereas the bounds of the integration is such that a field is integrated within the cloud.

Backend.cloud_line_integration.**cloud_line_integral**(*field_function*, *cloud_equation*, *view_line_point*, *box_width*, *view_line_deltas=(1*, *0*, *0)*, *n_guesses=100*, *integral_method='scipy'*)

Computs the line integral over a field given bounds of a cloud and path.

This function computes the total summation of the line integrals given a field function that a single sightline passes through, given the boundary that only the section of the line within a cloud would be computed as it is the upper and lower bounds for the integral(s).

Parameters

- **field_function** (*function*) – The function of the field. Must be three dimensional in the form `def f(x,y,z): return a`. Does not accept non-numerical returns.
- **cloud_equation** (*function*) – The implicit equation of the cloud. Must be `f(x)` where `f(x) = 0`.
- **view_line_point** (*array_like*) – Expected in three dimensions. It specifies the point that the viewline is positioned at in cartesian space.
- **box_width** (*float*) – An overestimated value of the size of the cloud along any given axis. Used for finding locations of intersections of the cloud and sightline.
- **view_line_deltas** (*array_like; optional*) – Expected in three dimensions. It specifies the linear coefficient that the sightline travels through space. Defaults to `(1,0, 0)`, a line parallel to the x-axis.
- **n_guesses** (*int; optional*) – An order of magnitude overestimate of the number of intersections between the cloud and the sightline. Defaults to `100`.
- **integral_method** (*string; optional*) – The method of which the integration will be computed. Defaults to Scipy's `scipy.integrate.quad()`.

Returns

- **integrated_value** (*float*) – The integrated value of the given field bounded by the sightline and the shape of the cloud.
- **error** (*float*) – The associated error with the integration.

Backend.cloud_line_integration.**line_integral_boundaries**(*view_line_point*, *cloud_equation*, *box_width*, *view_line_deltas=(1, 0, 0)*, *n_guesses=100*)

Find line integral boundaries given the cloud and sightline.

This function determines the points that intersect the sphere, starting with it entering and exit. It returns the ranges of points that would yield line integral boundaries that integrate within the cloud volume.

By default, the cloud equation should be a function such that it returns a float, `f(x,y,z)`, based on implicit shape making: `f(x,y,z) = 0`. If not, it should be at least a string that contains the python syntax expression of the shape for `f(x,y,z) = 0`, i.e., left-hand side of the equation only.

Parameters

- **view_line_point** (*array_like*) – Expected in three dimensions. It specifies the point that the viewline is positioned at in cartesian space.
- **cloud_equation** (*function*) – The implicit equation of the cloud. Must be `f(x)` where `f(x) = 0`.
- **box_width** (*float*) – An overestimated value of the size of the cloud along any given axis. Used for finding locations of intersections of the cloud and sightline.
- **view_line_deltas** (*array_like*) – Expected in three dimensions. It specifies the linear coefficient that the sightline travels through space. Defaults to `(1,0,0)`, a line parallel to the x-axis.
- **n_guesses** (*int*) – An order of magnitude overestimate of the number of intersections between the cloud and the sightline. Defaults to `100`.

Returns

- **lower_bounds** (*ndarray*) – An array of the lower bound(s) if each integration needed that is within the cloud along the sightline.
- **upper_bounds** (*ndarray*) – An array of the upper bound(s) if each integration needed that is within the cloud along the sightline.

Backend.coordinate_system_transformation module

This file is some side functions that allows for the conversion of cordinate systems.

Backend.coordinate_system_transformation.**cartesian_to_cylindrical_3d**(*x*, *y*, *z*)

Convert cartesian points to cylindrical points.

Convert cartesian coordinate points in 3D to cylindrical coordinate points in 3D. This function uses the notation convention of ISO 80000-2:2009 and its related successors.

Parameters

- **x** (*array_like*) – The x values of the points to be transformed.
- **y** (*array_like*) – The y values of the points to be transformed.
- **z** (*array_like*) – The z values of the points to be transformed.

Returns

- **rho** (*array_like*) – The rho (radial) values of the points after transformation.
- **phi** (*array_like*) – The phi (angular) values of the points after transformation.
- **z** (*array_like*) – The z (height) values of the points after transformation.

Backend.coordinate_system_transformation.**cartesian_to_polar_2d**(*x*, *y*)

Convert cartesian points to polar points.

Convert cartesian coordinate points in 2D to polar coordinate points in 2D. This function uses the notation convention of ISO 80000-2:2009 and its related successors.

Parameters

- **x** (*array_like*) – The x values of the points to be transformed.
- **y** (*array_like*) – The y values of the points to be transformed.

Returns

- **rho** (*array_like*) – The rho (radial) values of the points after transformation.
- **phi** (*array_like*) – The phi (angular) values of the points after transformation.

Backend.coordinate_system_transformation.**cartesian_to_spherical_3d**(*x*, *y*, *z*)

Convert cartesian points to cylindrical points.

Convert cartesian coordinate points in 3D to cylindrical coordinate points in 3D. This function uses the notation convention of ISO 80000-2:2009 and its related successors.

Parameters

- **x** (*array_like*) – The x values of the points to be transformed.
- **y** (*array_like*) – The y values of the points to be transformed.
- **z** (*array_like*) – The z values of the points to be transformed.

Returns

- **r** (*array_like*) – The rho (radial) values of the points after transformation.
- **theta** (*array_like*) – The theta (azimuthal angle) values of the points after the transformation.
- **phi** (*array_like*) – The phi (polar angle) values of the points after the transformation.

Backend.coordinate_system_transformation.**cylindrical_to_cartesian_3d**(*rho*, *phi*, *z*)

Convert cylindrical points to cartesian points.

Convert cylindrical coordinate points in 3D to cartesian coordinate points in 3D. This function uses the notation convention of ISO 80000-2:2009 and its related successors.

Parameters

- **rho** (*array_like*) – The rho values of the points to be transformed.
- **phi** (*array_like*) – The phi values of the points to be transformed.
- **z** (*array_like*) – The z values of the points to be transformed.

Returns

- **x** (*array_like*) – The x values of the points after transformation.
- **y** (*array_like*) – The y values of the points after transformation.
- **z** (*array_like*) – The z values of the points after transformation.

Backend.coordinate_system_transformation.**cylindrical_to_spherical_3d**(*rho*, *phi*, *z*)

Convert cylindrical points to spherical points.

Convert cylindrical coordinate points in 3D to spherical coordinate points in 3D. This function uses the notation convention of ISO 80000-2:2009 and its related successors.

Parameters

- **rho** (*array_like*) – The rho values of the points to be transformed.
- **phi** (*array_like*) – The phi values of the points to be transformed.
- **z** (*array_like*) – The z values of the points to be transformed.

Returns

- **r** (*array_like*) – The rho (radial) values of the points after transformation.
- **theta** (*array_like*) – The theta (azimuthal angle) values of the points after the transformation.
- **phi** (*array_like*) – The phi (polar angle) values of the points after the transformation.

Backend.coordinate_system_transformation.**polar_to_cartesian_2d**(*rho*, *phi*)

Convert polar points to cartesian points.

Convert polar coordinate points in 2D to cartesian coordinate points in 2D. This function uses the notation convention of ISO 80000-2:2009 and its related successors.

Parameters

- **rho** (*array_like*) – The rho values of the points to be transformed.
- **phi** (*array_like*) – The phi values of the points to be transformed.

Returns

- **x** (*array_like*) – The x values of the points after transformation.
- **y** (*array_like*) – The y values of the points after transformation.

Backend.coordinate_system_transformation.**spherical_to_cartesian_3d**(*r*, *theta*, *phi*)

Convert spherical points to cartesian points.

Convert spherical coordinate points in 3D to cartesian coordinate points in 3D. This function uses the notation convention of ISO 80000-2:2009 and its related successors.

Parameters

- **r** (*array_like*) – The r values of the points to be transformed.
- **theta** (*array_like*) – The theta values of the points to be transformed.
- **phi** (*array_like*) – The phi values of the points to be transformed.

Returns

- **x** (*array_like*) – The x values of the points after transformation.
- **y** (*array_like*) – The y values of the points after transformation.
- **z** (*array_like*) – The z values of the points after transformation.

Backend.coordinate_system_transformation.**spherical_to_cylindrical_3d**(*r*, *theta*, *phi*)

Convert cylindrical points to cartesian points.

Convert cylindrical coordinate points in 3D to cartesian coordinate points in 3D. This function uses the notation convention of ISO 80000-2:2009 and its related successors.

Parameters

- **r** (*array_like*) – The r values of the points to be transformed.
- **theta** (*array_like*) – The theta values of the points to be transformed.
- **phi** (*array_like*) – The phi values of the points to be transformed.

Returns

- **rho** (*array_like*) – The rho values of the points after transformation.
- **phi** (*array_like*) – The phi (angular) values of the points after transformation.
- **z** (*array_like*) – The z values of the points after transformation.

Backend.electromagnetic_field_polarization module

This file deals with the functions regarding polarization and the relationship between the electric and magnetic fields.

Backend.electromagnetic_field_polarization.**Stokes_parameters_from_field**(*E_i*, *E_j*, *percent_polarized=1*, *chi=0*)

Returns Stokes parameters based off non-circularly polarized light.

This function returns the Stokes parameters based off a given electric field vector.

Technically it can handle circularly polarized light, the value that must be given is chi, the angle between the semi-major axis of the polarization ellipse and the line segment connecting between two points on the ellipse and the semi-major and semi-minor axes. See note [1].

Parameters

- **E_i** (*float or array_like*) – The component of the electric field in the i-hat direction.
- **E_j** (*float or array_like*) – The component of the electric field in the j-hat direction.
- **percent_polarized** (*float*) – The percent of the EM wave that is polarized. It should not be too far off the value of 1.
- **chi** (*float or array_like*) – The parameter for circularly polarized light.

Returns

- **I** (*ndarray*) – The first Stokes parameter, equivalent to S_0. The intensity of the light.
- **Q** (*ndarray*) – The second Stokes parameter, equivalent to S_1. The x,y polarization aspect.
- **U** (*ndarray*) – The third Stokes parameter, equivalent to S_2. The a,b (45 deg off set of x,y) polarization aspect.
- **V** (*ndarray*) – The fourth Stokes parameter, equivalent to S_3. The circular polarization aspect.

Notes

[1] This function's notation is based on the following diagram. https://en.wikipedia.org/wiki/File:Polarisation_ellipse2.svg

[2] The equations for Stoke's parameters used is from the following: https://en.wikipedia.org/wiki/Stokes_parameters#Definitions.

Backend.electromagnetic_field_polarization.**angle_from_Stokes_parameters**(*Q*, *U*)

Given Stoke parameters Q,U, return the angle of polarization.

This function returns an angle of polarization in radians based on the values of two stoke parameters. The angle is signed.

Parameters

- **Q** (*array_like*) – The second Stokes parameter, equivalent to S_1. The x,y polarization aspect.
- **U** (*array_like*) – The third Stokes parameter, equivalent to S_2. The a,b (45 deg off set of x,y) polarization aspect.

Returns **angle** – The angle of the polarization corresponding to the given Q and U value pairs. The angle array is parallel to the Q and U array_like inputs.

Return type ndarray

Backend.electromagnetic_field_polarization.**electric_to_magnetic**(*E_i*, *E_j*, *normalize=False*)

Convert a 2D electric field vector set to a magnetic field set.

This function takes the electric field that would normally be seen in a polarization ellipse and converts it to the magnetic field vectors. This function returns a perpendicular vector of the magnetic field, perserving the magnitude.

In two dimensions, there are always two vectors perpendicular to a vector. Or one vector with positive and negative magnitude. In three, there are infinitely many, so it is much harder to give a good vector.

Parameters

- **E_i** (*float or array_like*) – The component of the electric field in the i-hat direction.
- **E_j** (*float or array_like*) – The component of the electric field in the j-hat direction.
- **normalize** (*bool; optional*) – If True, then the returned E field vector components are of a unitary vector. Default is False.

Returns

- **B_i** (*ndarray*) – The component of the magnetic field in the i-hat direction.
- **B_j** (*ndarray*) – The component of the magnetic field in the j-hat direction.

Backend.electromagnetic_field_polarization.**magnetic_to_electric**(*B_i*, *B_j*, *normalize=False*)

Convert a 2D magnetic field vector set to a electric field set.

This function takes the magnetic field that would normally be seen in a polarization ellipse and converts it to the electric field vectors. This function returns a perpendicular vector of the magnetic field, perserving the magnitude.

In two dimensions, there are always two vectors perpendicular to a vector. Or one vector with positive and negative magnitude. In three, there are infinitely many, so it is much harder to give a good vector.

Parameters

- **B_i** (*float or array_like*) – The component of the magnetic field in the i-hat direction.
- **B_j** (*float or array_like*) – The component of the magnetic field in the j-hat direction.
- **normalize** (*bool; optional*) – If true, then the returned E field vector components are of a unitary vector. Default is False.

Returns

- **E_i** (*ndarray*) – The component of the electric field in the i-hat direction.
- **E_j** (*ndarray*) – The component of the electric field in the j-hat direction.

Backend.plotting_customization module

Backend.plotting_customization.**shiftedColorMap**(*cmap*, *start=0*, *midpoint=0.5*, *stop=1.0*, *name='shiftedcmap'*)

Modifies the scaling and zero point of Colormap

Function to offset the "center" of a colormap. Useful for data with a negative min and positive max and you want the middle of the colormap's dynamic range to be at zero.

Adapted from: https://stackoverflow.com/questions/7404116/defining-the-midpoint-of-a-colormap-in-matplotlib

Parameters

- **cmap** (*The matplotlib colormap to be altered*) –
- **start** (*Offset from lowest point in the colormap's range.*) – Defaults to 0.0 (no lower offset). Should be between 0.0 and *midpoint*.
- **midpoint** (*The new center of the colormap. Defaults to*) – 0.5 (no shift). Should be between 0.0 and 1.0. In general, this should be 1 - vmax / (vmax + abs(vmin)) For example if your data range from -15.0 to +5.0 and you want the center of the colormap at 0.0, *midpoint* should be set to 1 - 5/(5 + 15)) or 0.75
- **stop** (*Offset from highest point in the colormap's range.*) – Defaults to 1.0 (no upper offset). Should be between *midpoint* and 1.0.

Returns **newcmap**

Return type The new matplotlib colormap.

Backend.plotting_customization.**zeroedColorMap**(*cmap*, *min_val*, *max_val*, *name='shiftedcmap'*)

Adapts *shiftedColorMap()* such that it is centered at zero.

This function is a thin wrapper around the previous function such that the midpoint is automatically set to zero.

Parameters

- **cmap** (*The matplotlib colormap to be altered*) –
- **start** (*Offset from lowest point in the colormap's range.*) – Defaults to 0.0 (no lower offset). Should be between 0.0 and *midpoint*. should be set to 1 - 5/(5 + 15)) or 0.75

- **stop** (*Offset from highest point in the colormap's range.*) – Defaults to 1.0 (no upper offset). Should be between *midpoint* and 1.0.

Returns **newcmap**

Return type The new matplotlib colormap.

Backend.table_interpolation module

This file contains an entire list of functions that would interpolate a table of values into a numerical approximation of a function. Although it cannot be an analytical function, these functions attempt to return an function which would mimic such analytical functions by extrapolating tables.

Backend.table_interpolation.**funt_interpolate_scalar_table**(*x_array*, *y_array*, *z_array*, *ans_array*, *interp_method*)

Generate functional interpolation of a scalar table.

This function takes a table of x,y,z points which correspond to some scalar answer and attempts to interpolate to generate a function which would allow for the computation of the scalar at any arbitrary x,y,z point.

Parallel array representation of the able is assumed.

Parameters

- **x_array** (*array_like*) – The x values of the scalar table.
- **y_array** (*array_like*) – The y values of the scalar table.
- **z_array** (*array_like*) – The z values of the scalar table.
- **ans_array** (*array_like*) – The scalar answers of the table.

Returns **interpolated_scalar_function** – The numerical approximation to the generalized function.

Return type function

Backend.table_interpolation.**funt_interpolate_vector_table**(*x_array*, *y_array*, *z_array*, *x_ans_array*, *y_ans_array*, *z_ans_array*, *interp_method*)

Generate functional interpolation of a scalar table.

This function takes a table of x,y,z points which correspond to some scalar answer and attempts to interpolate to generate a function which would allow for the computation of the scalar at any arbitrary x,y,z point.

Parallel array representation of the able is assumed.

Parameters

- **x_array** (*array_like*) – The x values of the scalar table.
- **y_array** (*array_like*) – The y values of the scalar table.
- **z_array** (*array_like*) – The z values of the scalar table.
- **x_ans_array** (*array_like*) – The x component of the answer vector.
- **y_ans_array** (*array_like*) – The y component of the answer vector.
- **z_ans_array** (*array_like*) – The z component of the answer vector.

Returns **interpolated_vector_function** – The numerical approximation to the generalized function.

Return type function

3.1.2 Module contents

3.2 Robustness package

3.2.1 Submodules

Robustness.exception module

exception Robustness.exception.**AstronomyError**(*message*)
Bases: Exception

An error to be used if some of the programs executed are trying to do something that is nonsensical in the context of astronomy.

exception Robustness.exception.**InputError**(*message*)
Bases: Exception

An error to be used when the input from a user is incorrect.

exception Robustness.exception.**OutputError**(*message*)
Bases: Exception

An error to be used when the output is grossly unexpected.

exception Robustness.exception.**ShapeError**(*message*)
Bases: Exception

An error to be used when the dimensions or sizes of an array is incorrect.

exception Robustness.exception.**TerminateError**(*message*)
Bases: BaseException

A very serious error that should override the try-except blocks that are written.

Robustness.input_parsing module

The purpose of this function is that inputs are parsed correctly.

Robustness.input_parsing.**user_equation_parse**(*user_eq_input*, *variables*)
Convert input implicit equation into a function.

This function returns a functional form of a user's input expression. Only standard python math functions are to be used, and nothing else. The functional form will be in return f(x), for the user inputs some string for f(x). Variables is a string tuple that contains the list of variables expected in the equation parse.

Parameters

- **user_eq_input** (*string or function*) – This is the wanted string or function to be converted. If it is a function, there is simple verification before passing it back.
- **variables** (*tuple of strings*) – This is the symbols within the equation for parsing as the input (e.g. the `x` and `y` in `f(x,y)`)

Returns function – A callable function that executes the mathematical expression given in the string. The order of parameters from variables are kept.

Return type function

Raises

- *DangerWarning : Warning* – This is used because the eval() function is used in this code.

- *TerminateError : BaseException* – This is done if the verification of the continuation of the program fails.

Notes

This function does use the eval function and excessive precautions are used.

Robustness.validation module

Robustness.validation.**validate_boolean_array**(*boolean_array_input*, *shape=None*, *size=None*, *deep_validate=False*)
The purpose of this function is to validate that the boolean array is valid. The shape and size of the array can be optionally tested.

deep_validate instructs the program to loop over every element array and validate it in turn.

Robustness.validation.**validate_boolean_value**(*boolean_value_input*)
The purpose of this function is to validate that a boolean value input is valid.

Robustness.validation.**validate_float_array**(*float_array_input*, *shape=None*, *size=None*, *deep_validate=False*, *greater_than=None*, *less_than=None*)
The purpose of this function is to validate that the float array is valid. The shape and size of the array can be optionally tested.

deep_validate instructs the program to loop over every element array and validate it in turn.

Robustness.validation.**validate_float_value**(*float_value_input*, *greater_than=None*, *less_than=None*)
The purpose of this function is to validate that a float value is valid. The value, its range (either greater than or less than a number) may also be tested. This function will bark if the value is greater than less_than or less than greater_than.

Robustness.validation.**validate_function_call**(*input_function*, *n_parameters=None*)
Check if the input function is a valid callable function.

Robustness.validation.**validate_int_array**(*int_array_input*, *shape=None*, *size=None*, *deep_validate=False*, *non_zero=None*, *greater_than=None*, *less_than=None*)
The purpose of this function is to validate that the integer array is valid. The shape and size of the array can be optionally tested.

deep_validate instructs the program to loop over every element array and validate it in turn.

Robustness.validation.**validate_int_value**(*int_value_input*, *non_zero=None*, *greater_than=None*, *less_than=None*)
The purpose of this function is to validate that a int value is valid. The value, its range (either greater than or less than a number) may also be tested. This function will bark if the value is greater than less_than or less than greater_than. It can also be tested if it is non-zero (true for non-zero, false for zero passes).

Robustness.validation.**validate_list**(*input_list*, *length=None*)
The purpose of this function is to validate that a list is valid.

Robustness.validation.**validate_string**(*input_string*, *length=None*, *contain_substr=None*)
The purpose of this function is to determine if a string is valid. The length of the string can also be tested. If the string contains a substring can also be tested. Due to oddities of strings, this function may return back false errors.

Robustness.validation.**validate_tuple**(*input_tuple*, *length=None*)
The purpose of this function is to validate that a tuple is valid.

Robustness.warning module

exception Robustness.warning.**AstronomyWarning**(*message*)
Bases: *Robustness.warning.PhysicalityWarning*

A warning to be used when the current program is doing something a bit risky or something that would not make normal sense in astronomical terms.

exception Robustness.warning.**DangerWarning**(*message*)
Bases: Warning

A warning to be used when some input or output is dangerous for the system or program itself and may be disastrous with unexpected inputs.

exception Robustness.warning.**InputWarning**(*message*)
Bases: Warning

A warning to be used when the values of some item is incorrect, but is fixed within the program.

exception Robustness.warning.**OutputWarning**(*message*)
Bases: Warning

A warning to be used when the values of an output may not use all of the inputs given, or that it might become unexpected because of bugs.

exception Robustness.warning.**PhysicalityWarning**(*message*)
Bases: Warning

A warning to be used when the current program is doing something that does not make sense in real life.

exception Robustness.warning.**PhysicsWarning**(*message*)
Bases: *Robustness.warning.PhysicalityWarning*

A warning to be used when the current program is doing something a bit risky or something that would not make normal sense in physics terms.

exception Robustness.warning.**TimeWarning**(*message*)
Bases: ResourceWarning

A warning to be used when some computation or flag might take a long time to compute.

Robustness.warning.**kyubey_warning**(*warn_class*, *message*, *stacklevel=2*, *input_halt=False*)
General warning for the Robustness module/function package. If the warning is serious enough (like a DangerWarning), then force the user to ensure the continuation of the program.

Parameters

- **warn_class** (warning object) – The warning which to submit.
- **message** (*string*) – The warning message.
- **stacklevel** (*int*) – The stack level call that the warning goes back to.
- **input_halt** (*bool; optional*) – If the warning requires user input to continue, this is true. Defaults to false.

Raises *TerminateError* in the event of the input halt failing.

3.2.2 Module contents

3.3 data_systematization module

This file contains the methods needed to convert from a non-standard or accepted data type into a more usable datatype by this module.

class data_systematization.**InterpolationTable**(*x_values*, *y_values*, *z_values*, *classification*, *scalar_ans=None*, *x_vector_ans=None*, *y_vector_ans=None*, *z_vector_ans=None*)

Bases: object

A class representing either a scalar or vector table.

If a lookup table is to be used instead of a function for the model observing method, it is required to standardize the information given by the user's lookup table, as is the purpose of this class.

Parameters

- **x_values** (*array_like*) – The values of the x points for the table. Array must be parallel with y_values and z_values along with the scalar/vector answer.
- **y_values** (*array_like*) – The values of the x points for the table. Array must be parallel with x_values and z_values along with the scalar/vector answer.
- **z_values** (*array_like*) – The values of the x points for the table. Array must be parallel with x_values and y_values along with the scalar/vector answer.
- **classification** (*string*) – The classification of this table, either as a scalar lookup table or a vector lookup table. Should be one of:
  - 'scalar' A scalar based lookup table.
  - 'vector' A vector based lookup table.
- **scalar_ans** (*array_like, {for | classification == 'scalar'}*) – The scalar answers to the (x,y,z) point given by the input values. Must be parallel with x_values, y_values, and z_values. Ignored if classification == 'vector'.
- **x_vector_ans** (*array_like, {for | classification == 'vector'}*) – The x component of the answer vector that exists at the point (x,y,z) given by the input values. Must be parallel with x_values, y_values, and z_values along with other components. Ignored if classification == 'scalar'
- **y_vector_ans** (*array_like, {for | classification == 'vector'}*) – The y component of the answer vector that exists at the point (x,y,z) given by the input values. Must be parallel with x_values, y_values, and z_values along with other components. Ignored if classification == 'scalar'
- **z_vector_ans** (*array_like, {for | classification == 'vector'}*) – The z component of the answer vector that exists at the point (x,y,z) given by the input values. Must be parallel with x_values, y_values, and z_values along with other components. Ignored if classification == 'scalar'

numerical_function() : function {returns | function}

Returns a function which is an interface to a numerical approximation interpolation of the data points given in the lookup table. Automatically detects if it is a scalar function or vector function.

Methods

numerical_function([interp_method])	Generate a numerical function from the lookup table.

__init__(*x_values*, *y_values*, *z_values*, *classification*, *scalar_ans=None*, *x_vector_ans=None*, *y_vector_ans=None*, *z_vector_ans=None*)

A class representing either a scalar or vector table.

If a lookup table is to be used instead of a function for the model observing method, it is required to standardize the information given by the user's lookup table, as is the purpose of this class.

Parameters

- **x_values** (*array_like*) – The values of the x points for the table. Array must be parallel with y_values and z_values along with the scalar/vector answer.
- **y_values** (*array_like*) – The values of the x points for the table. Array must be parallel with x_values and z_values along with the scalar/vector answer.
- **z_values** (*array_like*) – The values of the x points for the table. Array must be parallel with x_values and y_values along with the scalar/vector answer.
- **classification** (*string*) – The classification of this table, either as a scalar lookup table or a vector lookup table. Should be one of:
  - 'scalar' A scalar based lookup table.
  - 'vector' A vector based lookup table.
- **scalar_ans** (*array_like, {for | classification == 'scalar'}*) – The scalar answers to the (x,y,z) point given by the input values. Must be parallel with x_values, y_values, and z_values. Ignored if classification == 'vector'.
- **x_vector_ans** (*array_like, {for | classification == 'vector'}*) – The x component of the answer vector that exists at the point (x,y,z) given by the input values. Must be parallel with x_values, y_values, and z_values along with other components. Ignored if classification == 'scalar'
- **y_vector_ans** (*array_like, {for | classification == 'vector'}*) – The y component of the answer vector that exists at the point (x,y,z) given by the input values. Must be parallel with x_values, y_values, and z_values along with other components. Ignored if classification == 'scalar'
- **z_vector_ans** (*array_like, {for | classification == 'vector'}*) – The z component of the answer vector that exists at the point (x,y,z) given by the input values. Must be parallel with x_values, y_values, and z_values along with other components. Ignored if classification == 'scalar'

numerical_function(*interp_method='linear'*)

Generate a numerical function from the lookup table.

This function creates a functional interface of the data from a lookup table. It interpolates values that are not in the table to return what Scipy thinks is the best value.

Parameters **interp_method** (*string, optional*) – The method of interpolation to be used. Must be one of the following strings:

- 'nearest'
- 'linear', default
- 'cubic'

Returns **numeric_function** – The numerical interpolation function that attempts to best replicate the table.

Return type function

3.4 density_field_functions module

This document contains a large amount of the implemented density field functions for ease of access.

3.5 magnetic_field_functions_2d module

This document highlights 2d versions of magnetic field functions. Thus, these functions take in 2 parameters, the x,y values or rho,phi values, and return the u,v vector components of the given magnetic field

magnetic_field_functions_2d.**circular_magnetic_field_cart_2d**(*x, y, center=[0, 0], mag_function=<function <lambda>>*)

Compute the cartesian magnetic field vectors of a circular field.

The circular magnetic field is radially symmetric. This function returns the values of the components of a magnetic field given some point(s) x,y. The center of the circular magnetic field can be redefined.

Parameters

- **x** (*array_like*) – The x values of the points for magnetic field computation.
- **y** (*array_like*) – The y values of the points for magnetic field computation.
- **center** (*array_like; optional*) – The center of the circular magnetic field function, passed as an array `[x0,y0]`. Defaults to `[0,0]`
- **mag_function** (function `f(r)`; optional) – The value of the magnitude of the vector field at some radius from the center. Default is `f(r) = 1/r**2`.

Returns

- **Bfield_x** (*ndarray*) – The x component of the magnetic field at the given points. Order is perserved.
- **Bfield_y** (*ndarray*) – The y component of the magnetic field at the given points. Order is perserved.

magnetic_field_functions_2d.**hourglass_magnetic_field_cart_2d**(*r, z, h, k_array, disk_radius, uniform_B0, center=[0, 0]*)

Equation for hourglass magnetic fields given by Ewertowshi & Basu 2013.

This function is the two dimensional version of the equations given by Ewertowshi & Basu 2013. This projects the values of the three dimensional form to the r-z plane in cylindrical coordinates. In practice, r,z can be mapped to x,y. The center can also be changed.

Parameters

- **r** (*array_like*) – The input values of the radial direction for the equation.
- **z** (*array_like*) – The input values of the polar direction for the equation.
- **h** (*float*) – A free parameter as dictated by the paper.

- **k_array** (*array_like*) – The list of k coefficient values for the summation in Eq 45.
- **disk_radius** (*float*) – The radius of the protoplanetary disk. Relevent for the hourglass magnetic field generated by this paper.
- **uniform_B0** (*float*) – The magnitude of the background magnetic field.
- **center** (*array_like; optional*) – The center of the hourglass shaped magnetic field function, passed as an array `[r0,z0]`. Defaults to `[0,0]`

Returns

- **Bfield_r** (*ndarray*) – The value of the magnetic field in the r-axial direction.
- **Bfield_z** (*ndarray*) – The value of the magnetic field in the z-axial direction.

3.6 magnetic_field_functions_3d module

This document highlights 2d versions of magnetic field functions. Thus, these functions take in 2 parameters, the x,y values or rho,phi values, and return the u,v vector components of the given magnetic field

magnetic_field_functions_3d.**circular_magnetic_field_cart_3d**(*x, y, z, center=[0, 0, 0], mag_function=<function <lambda>>, curl_axis='z', **kwargs*)

Compute the cartesian magnetic field vectors of a circular field.

The circular magnetic field is radially symmetric. This function returns the values of the components of a magnetic field given some point(s) x,y,z. The center of the circular magnetic field can be redefined. The axis of symmetry can also be redefined.

Parameters

- **x** (*array_like*) – The x values of the points for magnetic field computation.
- **y** (*array_like*) – The y values of the points for the magnetic field computation
- **z** (*array_like*) – The z values of the points for magnetic field computation.
- **center** (*array_like; optional*) – The center of the circular magnetic field function, passed as an array `[x0,y0]`. Default is `[0,0,0]`
- **mag_function** (function `f(r)`; optional) – The value of the magnitude of the vector field at some radius from the center. Default is `f(r) = 1/r**2`.
- **curl_axis** (*string; optional*) – The specified axis that the magnetic field curls around. Default is the `z` axis.

Returns

- **Bfield_x** (*ndarray*) – The x component of the magnetic field at the given points. Order is perserved.
- **Bfield_y** (*ndarray*) – The y component of the magnetic field at the given points. Order is perserved.
- **Bfield_z** (*ndarray*) – The z component of the magnetic field at the given points. Order is perserved.

magnetic_field_functions_3d.**hourglass_magnetic_field_cart_3d**(*x, y, z, h, k_array, disk_radius, uniform_B0, center=[0, 0, 0], **kwargs*)

Equation for hourglass magnetic fields given by Ewertowshi & Basu 2013.

This function is the three dimensional version of the equations given by Ewertowshi & Basu 2013. This function assumes, as the paper does, that the magnetic field is invariant with respect to phi.

Parameters

- **x** (*array_like*) – The input values of the x direction for the equation.
- **y** (*array_like*) – The input values of the y direction for the equation.
- **z** (*array_like*) – The input values of the z direction for the equation.
- **h** (*float*) – A free parameter as dictated by the paper.
- **k_array** (*array_like*) – The list of k coefficient values for the summation in Eq 45.
- **disk_radius** (*float*) – The radius of the protoplanetary disk. Relevent for the hourglass magnetic field generated by this paper.
- **uniform_B0** (*float*) – The magnitude of the background magnetic field.
- **center** (*array_like; optional*) – The center of the hourglass shaped magnetic field function, passed as an array `[r0,phi0,z0]`. Defaults to `[0,0,0]`

Returns

- **Bfield_x** (*ndarray*) – The value of the magnetic field in the x-axial direction.
- **Bfield_y** (*ndarray*) – The value of the magnetic field in the y-axial direction.
- **Bfield_z** (*ndarray*) – The value of the magnetic field in the z-axial direction.

magnetic_field_functions_3d.**hourglass_magnetic_field_cyln_3d**(*rho, phi, z, h, k_array, disk_radius, uniform_B0, center=[0, 0, 0], **kwargs*)

Equation for hourglass magnetic fields given by Ewertowshi & Basu 2013.

This function is the three dimensional version of the equations given by Ewertowshi & Basu 2013. This function assumes, as the paper does, that the magnetic field is invariant with respect to phi.

Parameters

- **rho** (*array_like*) – The input values of the radial direction for the equation.
- **phi** (*array_like*) – The input values of the polar angle for the equation.
- **z** (*array_like*) – The input values of the polar direction for the equation.
- **h** (*float*) – A free parameter as dictated by the paper.
- **k_array** (*array_like*) – The list of k coefficient values for the summation in Eq 45.
- **disk_radius** (*float*) – The radius of the protoplanetary disk. Relevent for the hourglass magnetic field generated by this paper.
- **uniform_B0** (*float*) – The magnitude of the background magnetic field.

- **center** (*array_like*) – The center of the hourglass shaped magnetic field function, passed as an array `[r0,phi0,z0]`. Defaults to `[0,0,0]`

Returns

- **Bfield_rho** (*ndarray*) – The value of the magnetic field in the rho-axial direction.
- **Bfield_phi** (*ndarray*) – The value of the magnetic field in the phi-axial direction.
- **Bfield_z** (*ndarray*) – The value of the magnetic field in the z-axial direction.

magnetic_field_functions_3d.**linear_combination_magnetic_field**(*field_functions*, *contributions*, ***kwargs*)

This function handles the linear combination of functions.

This function, when given a list of magnetic field functions, their contributions, and the required arguments, will return another function which is based on the linear combination (as determined by contribution) of the given functions. The linear combination of the magnetic fields relies on the superposition principle.

Parameters

- **field_functions** (*array_like*) – The array of magnetic field functions that are to be used. Must be an array_like of callable functions.
- **contributions** (*array_like*) – The float value array of the contribution of each magnetic fields. May either be based on percentage, or a multiplicative factor.
- ****kwargs** (*arguments*) – A dictionary (or explicitly typed out list) of parameters that are required for the other field functions.

magnetic_field_functions_3d.**monopole_magnetic_field_cart_3d**(*x, y, z, center=[0, 0, 0], mag_function=<function <lambda>>, **kwargs*)

This is a monopole magnetic field, extending radially with zero curl.

This function gives a description of a magnetic field extending radially according to some magnitude function (as a function of radius). The point of radiance (the center of the field) can be adjusted as needed.

Note that this is currently an impossible shape for a magnetic field as determined by Maxwell's equations.

Parameters

- **x** (*array_like*) – The x values of the input points.
- **y** (*array_like*) – The y values of the input points.
- **z** (*array_like*) – The z values of the input points.
- **center** (*array_like*) – The center of the magnetic field.
- **mag_function** (*function*) – The magnitude of the magnetic fields as a function of radius.

Returns

- **Bfield_x** (*ndarray*) – The x component of the magnetic field.
- **Bfield_y** (*ndarray*) – The y component of the magnetic field.
- **Bfield_z** (*ndarray*) – The z component of the magnetic field.

magnetic_field_functions_3d.**monopole_magnetic_field_sphr_3d**(*r, theta, phi, center=[0, 0, 0], mag_function=<function <lambda>>, **kwargs*)

This is a monopole magnetic field, extending radially with zero curl.

This function gives a description of a magnetic field extending radially according to some magnitude function (as a function of radius). The point of radiance (the center of the field) can be adjusted as needed.

Note that this is currently an impossible shape for a magnetic field as determined by Maxwell's equations.

Parameters

- **r** (*array_like*) – The radial component of the input points.
- **theta** (*array_like*) – The polar angle component of the input points.
- **phi** (*array_like*) – The azimuthal angle component of the input points.
- **center** (*array_like*) – The center point of the magnetic field.
- **mag_function** (*function*) – The measured magnitude of the magnetic field as a function of `r`.

Returns

- **Bfield_r** (*ndarray*) – The radial component of the magnetic field at the given points.
- **Bfield_theta** (*ndarray*) – The polar angle component of the magnetic field at the given points.
- **Bfeild_phi** (*ndarray*) – The azimuthal angle component of the magnetic field at the given points.

3.7 model_observing module

Model observing, this module is built to simulate actual observing. The object is known, and given sight parameters, the data is given. In particular, these functions actually give the values of terms derived from the object model also provided.

class model_observing.**ObservingRun**(*observe_target*, *sightline*, *field_of_view*)

Bases: object

Execute a mock observing run of an object.

This class is the main model observations of an object. Taking a central sightline and the field of view, it then gives back a set of plots, similar to those that an observer would see after data reduction.

The class itself does the computation in its methods, returning back a heatmap/contour object plot from the observing depending on the method.

`self.`**observe_target**
ProtostarModel object – The model target for simulated observing. Conceptually, the object that the telescope observes.

`self.`**sightline**
Sightline object – The primary sightline that is used for the model observing, conceptually where the telescope is aimed at.

`self.`**field_of_view**
float – The field of view value of the observation, given as the length of the observing chart.

Stokes_parameter_contours() : function {returns | ndarray,ndarray}
Compute the value of Stoke parameters at random sightlines from the primary sightline and plot them. Returns the values that was used to plot.

Methods

Stokes_parameter_contours([plot_parameters, ...])	This function produces a contour plot of the stoke values.

Stokes_parameter_contours (*plot_parameters=True*, *n_axial_samples=25*)
This function produces a contour plot of the stoke values.

This function generates a large number of random sightlines to traceout contour information of the of the fields. From there, is creates and returns a contour plot.

The values of the intensity, I, the two polarization values, Q,U, and the polarization intensity, hypt(Q,U) is plotted.

Parameters

- **plot_parameters** (*bool; optional*) – A boolean value to specify if the user wanted the parameters to be plotted.
- **n_axial_samples** (*int; optional*) – The number of points along one RA or DEC axis to be sampled. The resulting sample is a mesh n**2 between the bounds. Default is 25.

Returns

- **ra_dec_array** (*tuple(ndarray)*) – This is a tuple of the values of the RA and DEC of the random sightlines (arranged in parallel arrays).
- **stokes_parameters** (*tuple(ndarray)*) – This is a tuple of ndarrays of the stoke parameters calculated by the random sightlines.

__init__ (*observe_target*, *sightline*, *field_of_view*)
Doing an observing run.

Create an observing run object, compiling the primary sightline and the field of view.

Parameters

- **observe_target** (*ProtostarModel* object) – This is the object to be observed.
- **sightline** (*Sightline object*) – This is the primary sightline, in essence, where the telescope is pointing in this simulation.
- **field_of_view** (*float*) – The width of the sky segment that is being observed. Must be in radians. Applies to both RA and DEC evenly for a square image. Seen range is ‘‘ (RA,DEC) ± field_of_view/2 ‘‘.

class model_observing.**ProtostarModel** (*coordinates*, *cloud_model*, *magnetic_field_model*, *density_model=None*, *polarization_model=None*, *zeros_guess_count=100*)
Bases: object

This is an object that represents a model of an object in space. It contains all the required functions and parameters associated with one of the objects that would be observed for polarimetry data.

`self.`**`coordinates`**
Astropy `SkyCoord` object – This is the coordinates of the object that this class defines.

`self.`**`cloud_model`**
function – This is an implicit function (or a numerical approximation thereof) of the shape of the protostar cloud.

`self.`**`magnetic_field`**
function – This is an implicit function (or a numerical approximation thereof) of the shape of the magnetic field.

`self.`**`density_model`**
function – This is an implicit function (or a numerical approximation thereof) of the shape of the density model of the cloud.

`self.`**`polarization_model`**
function – This is an implicit function (or a numerical approximation thereof) of the polarization model of the cloud.

`__init__`(*coordinates*, *cloud_model*, *magnetic_field_model*, *density_model=None*, *polarization_model=None*, *zeros_guess_count=100*)
Object form of a model object to be observed.

This is the object representation of an object in the sky. The required terms are present.

Parameters

- **`coordinates`** (Astropy `SkyCoord` object) – These are the coordinates of the observation object. It is up to the user to put as complete information as possible.
- **`cloud_model`** (*`function or string,`*) – An implicit equation of the cloud. The origin of this equation must also be the coordinate specified by self.coordinates. Must be cartesian in the form `f(x,y,z) = 0`, for the function or string is `f(x,y,z)`. The x-axis is always aligned with a telescope as it is the same as a telescope's r-axis.
- **`magnetic_field_model`** (function or *`InterpolationTable`*) – A function that, given a single point in cartesian space, will return the value of the magnitude of the magnetic field's three orthogonal vectors in xyz-space. If an interpolation table is given, a numerical approximation function will be used instead.
- **`density_model`** (function or string, or *`InterpolationTable`*; optional) – A function that, given a point in cartesian space, will return a value pertaining to the density of the gas/dust within at that point. Defaults to uniform. If an interpolation table is given, a numerical approximation function will be used instead.
- **`polarization_model`** (function, string, float or *`InterpolationTable`*; optional) – This is the percent of polarization of the light. Either given as a function (or string representing a function) `f(x,y,z)`, or as a constant float value. Default is uniform value of 1. If an interpolation table is given, a numerical approximation function will be used instead.
- **`zeros_guess_count`** (*`int; optional`*) – This value stipulates how many spread out test points there should be when finding sightline intersection points. A higher number should be used for complex shapes. Defaults at 100.

`class` `model_observing.`**`Sightline`**(*right_ascension*, *declination*, *SkyCoord_object=None*)
Bases: `object`

This is a sightline. It contains the information for a given sightline through space. The sightline is always given by the RA and DEC values.

The notation for the accepted values of RA and DEC is found in the Astropy module's `SkyCoord` class.

`self.coordinates`
Astropy `SkyCoord` object. – This is the sky coordinates of the sightline.

sightline_parameters() *: function (returns | ndarray, ndarray)*
This method returns back both the sightline's center and slopes for an actual geometrical representation of the line. Converting from the equatorial coordinate system to the cartesian coordinate system.

Methods

sightline_parameters()	This function returns the sightline linear parameters.

__init__(*right_ascension*, *declination*, *SkyCoord_object=None*)
Initialization of a sightline.

The creates the sightline's main parameters, the defineing elements of the sightline is the location that it is throughout space. This is a specific wrapper around `SkyCoord`.

Parameters

- **right_ascension** (*string*) – The right ascension value for the sightline. This term must be formatted in the Astropy `SkyCoord` format: `00h00m00.00s`. For the values of the seconds are decimal and may extend to any precision.
- **declination** (*string*) – The declination value for the sightline. This term must be formatted in the Astropy `SkyCoord` format: `±00d00m00.00s`. For the values of the seconds are decimal and may extend to any precision.
- **Skycord_object** (`SkyCoord` object; optional) – It may be easier to also just pass an Astropy `SkyCoord` object in general. The other strings are ignored if it is successful.

sightline_parameters()
This function returns the sightline linear parameters.

The sightline is by definition always parallel to the x-axis of the object to be observed. The plane of the sky is the yz-plane of the object. This function returns first the central defining point, then the deltas for the equation.

Returns

- **sightline_center** (*ndarray*) – This returns a cartsian point based on the approximation that, if the x-axis and the r-axis are the same of cartesian and spherical cordinates, then so too are the yz-plane and the theta-phi plane.
- **sightline_slopes** (*ndarray*) – This returns the slopes of the cartesian point values given by the center. Because of the approximation from above, it is always [1,0,0].

Notes

The coordinates of the sightline in relation to the object are as follows:

- The x-axis of the object is equal to the r-axis of the telescope. Both pointing away from the telescope, deeper into space.
- The y-axis of the object is equal to the RA-axis/phi-axis of the

telescope, westward (as y increases, RA decreases) - The z-axis of the object is equal to the DEC-axis of the telescope. It is also equal to the negative of the theta-axis when it is centered on theta = pi/2. Points north-south of the telescope.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

d

m

r

INDEX

Symbols

A

B

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

Z