TRIPPy Documentation

Release 1.0

Ian Faust

May 01, 2014

CHAPTER

ONE

SUBPACKAGES

1.1 TRIPPy.plot package

1.1.1 Submodules

1.1.2 TRIPPy.plot.plot module

`TRIPPy.plot.plot.`**`genCartGrid`**(*x0*, *x1*, *x2*, *edges=False*)

`TRIPPy.plot.plot.`**`genCylGrid`**(*x0*, *x1*, *x2*, *edges=False*)

`TRIPPy.plot.plot.`**`genVertsFromPixel`**(*grid*)
reduces the lengths of the dimensions by 1

`TRIPPy.plot.plot.`**`plotLine`**(*vector*, *val=1.0*, *close=False*, *tube_radius=None*, *index=None*, ***kwargs*)
PlotLine creates a single plot object from a singular vector or from a n-dimensional tuple or list.

`TRIPPy.plot.plot.`**`plotSymIso`**(*plasma*, *pts=15*, ***kwargs*)

`TRIPPy.plot.plot.`**`plotTangency`**(*plasma*, *beam*)
nonlinear minimization of r^2 between the plasma center (in R,Z) versus defined beam s vector

`TRIPPy.plot.plot.`**`plotTokamak`**(*tokamak, angle=[0, 6.283185307179586], pts=250, section=None, **kwargs*)

`TRIPPy.plot.plot.`**`plotView`**(*rays*, *pts=None*, ***kwargs*)

`TRIPPy.plot.plot.`**`plotVol`**(*volume*, *pts=15*, ***kwargs*)

`TRIPPy.plot.plot.`**`plotVol2`**(*plasma*, *pts=None*, *lim=False*, ***kwargs*)

1.1.3 Module contents

CHAPTER

TWO

SUBMODULES

CHAPTER THREE

TRIPPY.AXUV MODULE

`TRIPPy.AXUV.`**`AXUV20`**`(`*temp*`)`

`TRIPPy.AXUV.`**`AXUV22`**`(`*temp*`)`

temp

CHAPTER

FOUR

TRIPPY.BPLY MODULE

TRIPPy.BPLY.**BPLY**(*temp*, *place=(1.87, 0, 0.157277)*, *angle=(0, 1.7453263267948966, -0.05305367320510346)*)

TRIPPy.BPLY.**BPLYbeam**(*alcator*)

TRIPPy.BPLY.**calcArea**(*points*)

TRIPPy.BPLY.**calctoTree**(*shot*)

TRIPPy.BPLY.**effectiveHeight**(*surf1*, *surf2*, *plasma*, *t*, *lim1=0.88*, *lim2=0.92*)
calculate the effective height of a view through the scrape-off-layer

TRIPPy.BPLY.**getBeamFluxSpline**(*beam*, *plasma*, *t*, *lim1*, *lim2*, *points=1000*)
generates a spline off of the beampath. Assumes that the change in flux is MONOTONIC

TRIPPy.BPLY.**globalpowerCalc**(*shot*)

TRIPPy.BPLY.**trend**(*tree*, *signal*, *shot*)

TRIPPy.BPLY.**viewPoints**(*surf1*, *surf2*, *plasma*, *t*, *lim1=0.88*, *lim2=0.92*, *fillorder=True*)

TRIPPy.BPLY.**writeGlobal**(*shot*)

TRIPPy.BPLY.**writeToTree**(*inp*)

CHAPTER

FIVE

TRIPPY.CMODTS MODULE

`TRIPPy.CmodTS.`**`Chords`**`(`*plasma*`)`

CHAPTER

SIX

TRIPPY.NSTX MODULE

`TRIPPy.NSTX.`**`diode1`**(*temp*, *place=(2.0, 0, 0.02)*, *angle=(0, 1.5707963267948966, 0)*)

`TRIPPy.NSTX.`**`diode2`**(*temp*, *place=(2.0, 0, -0.02)*, *angle=(0, 1.5707963267948966, 0)*)

`TRIPPy.NSTX.`**`diode3`**(*temp*, *place=(0.8, 0, 1.8)*, *angle=(0, 1.5707963267948966, 0)*)

`TRIPPy.NSTX.`**`diode4`**(*temp*, *place=(1.5, 0, -1.5)*, *angle=(0, 1.5707963267948966, 0)*)

`TRIPPy.NSTX.`**`diode5`**(*center*, *ap=(0.002, 0, 0.08)*)

`TRIPPy.NSTX.`**`diode6`**(*center*, *ap=(0.0225, 0, 0.08)*, *angler=(0, 0, 0)*)

CHAPTER

SEVEN

TRIPPY.XTOMO MODULE

TRIPPy.XTOMO.**XTOMOchip**(*temp*)

TRIPPY.BEAM MODULE

class TRIPPy.beam.**Beam**(*surf1*, *surf2*)

Bases: TRIPPy.geometry.Origin

Generates a Beam vector object assuming macroscopic surfaces

Uses the definition:

$$\vec{x} = \vec{x}_0 + \vec{x}_1$$

Args:

surf1: Surface or Surface-derived object Defines the origin surface, based on the coordinate system of the surface. Center position is accessible through Beam(0). Generated beam contains same origin as from surf1.

surf2: Surface or Surface-derived object Defines the aperature surface, based on the coordinate system of the surface. Both surfaces must be in the same coordinate system.

Returns: Beam: Beam object.

Examples: Accepts all surface or surface-derived object inputs, though all data is stored as a python object.

Generate an y direction Ray in cartesian coords using a Vec from (0,0,1):

```
cen = geometry.Center(flag=True)
ydir = geometry.Vecx((0,1,0))
zpt = geometry.Point((0,0,1),cen)
```

c()

Conversion of vector to opposite coordinate system

Returns: copy of vector object with opposite coordinate system (set with .flag parameter)

r()

return cylindrical coordinate values

Returns: numpy array of cylindrical coordinates in meters and radians

rmin()

Calculates and returns the s value along the norm vector which minimizes the r0() value (the closest position to the origin norm axis)

Returns: numpy array of s values in meters

smin(*point=None*)

Calculates and returns the s value along the norm vector which minimizes the distance from the ray to a point (default is the origin which the ray is defined).

Kwargs: point: Point or Point-derived object, otherwise defaults to ray origin

Returns: numpy array of s values in meters

t (*r*, *z*)
return toroidal coordinate values for given cylindrical coordinates (r,z) in coordinates of the beam origin.

Args: r: scipy-array of floats or float in meters. r is specified in meters.

z: scipy-array of floats or float in meters. z is specified in meters.

Returns: numpy array of cylindrical coordinates in [meters,radians,radians] where it is radius in meters, toroidal angle and then poloidal angle.

tmin (*r*, *z*, *trace=False*)
Calculates and returns the s value along the norm vector which minimizes the distance from the ray to a circle defined by input (r,z).

Args: r: value, iterable or scipy.array, radius in meters

z: value, iterable or scipy.array, z value in meters

Kwargs: trace: bool if set true, the ray is assumed to be traced within a tokamak. A further evaluation reduces the value to one within the bounds of the vacuum vessel/limiter.

Returns: numpy array of s values in meters

x ()
returns array of cartesian coordinate in meters

Returns: numpy array of cartesian coordinates in meters

class TRIPPy.beam.**Ray** (*pt1*, *inp2*)
Bases: TRIPPy.geometry.Point

Generates a ray vector object

Uses the definition:

$$\vec{x} = \vec{x}_0 + \vec{x}_1$$

Args:

pt1: Point or Point-derived object Defines the origin of the Ray, based on the coordinate system of the origin. pt1 position is accessible through Ray(0).

pt2: Point or Vector-derived object Direction of the ray can be defined by a vector object (assumed to be in the space of the pt1 origin) from pt1, or a point, which generates a vector pointing from pt1 to pt2.

Returns: Ray: Ray object.

Examples: Accepts all point and point-derived object inputs, though all data is stored as a python object.

Generate an y direction Ray in cartesian coords using a Vec from (0,0,1):

```
cen = geometry.Center(flag=True)
ydir = geometry.Vecx((0,1,0))
zpt = geometry.Point((0,0,1),cen)
```

r ()
return cylindrical coordinate values

Returns: numpy array of cylindrical coordinates in meters and radians

redefine (*neworigin*)
redefine Ray object or Ray-derived object into new coordinate system

Args: neworigin: Origin or Origin-derived object

rmin()

rmin returns the s value along the norm vector which minimizes the r0() value (the closest position to the origin norm axis)

Returns: numpy array of s values in meters

smin(*point=None*)

Calculates and returns the s value along the norm vector which minimizes the distance from the ray to a point (default is the origin which the ray is defined).

Kwargs: point: Point or Point-derived object, otherwise defaults to ray origin

Returns: numpy array of s values in meters

t(*r*, *z*)

return toroidal coordinate values for given cylindrical coordinates (r,z) in coordinates of the ray origin.

Args: r: scipy-array of floats or float in meters. r is specified in meters.

z: scipy-array of floats or float in meters. z is specified in meters.

Returns: numpy array of cylindrical coordinates in [meters,radians,radians] where it is radius in meters, toroidal angle and then poloidal angle.

tmin(*r*, *z*, *trace=False*)

Calculates and returns the s value along the norm vector which minimizes the distance from the ray to a circle defined by input (r,z).

Args: r: value, iterable or scipy.array, radius in meters

z: value, iterable or scipy.array, z value in meters

Kwargs: trace: bool if set true, the ray is assumed to be traced within a tokamak. A further evaluation reduces the value to one within the bounds of the vacuum vessel/limiter.

Returns: numpy array of s values in meters

x()

returns array of cartesian coordinate in meters

Returns: numpy array of cartesian coordinates in meters

TRIPPy.beam.**multiBeam**(*surf1*, *surf2*, *split=None*)

Generate a tuple of Beam objects from tuples of surface objects

Args:

surf1: tuple of Surfaces or a Surface object Beam origin surfaces, based on the coordinate system of the surfaces. Center position is accessible through Beam(0), Beam.x()[...,0] or Beam.r()[...,0] (last two options create numpy arrays, the first generats a geometry.Vec object).

surf2: tuple of Surfaces or a Surface object Direction of the ray can be defined by a vector object (assumed to be in the space of the pt1 origin) from pt1, or a point, which generates a vector pointing from pt1 to pt2.

Returns: output: tuple of beam objects.

Examples: Accepts all surface or surface-derived object inputs, though all data is stored as a python object.

Generate an y direction Ray in cartesian coords using a Vec from (0,0,1):

```
cen = geometry.Center(flag=True)
ydir = geometry.Vecx((0,1,0))
zpt = geometry.Point((0,0,1),cen)
```

`TRIPPy.beam.`**`volWeightBeam`**(*beam*, *rgrid*, *zgrid*, *trace=True*, *ds=0.002*, *toroidal=None*, ***kwargs*)

Generate a tuple of Beam objects from tuples of surface objects

Args:

beam: tuple of Surfaces or a Surface object Beam origin surfaces, based on the coordinate system of the surfaces. Center position is accessible through Beam(0), Beam.x()[...,0] or Beam.r()[...,0] (last two options create numpy arrays, the first generats a geometry.Vec object).

rgrid: tuple of Surfaces or a Surface object Direction of the ray can be defined by a vector object (assumed to be in the space of the pt1 origin) from pt1, or a point, which generates a vector pointing from pt1 to pt2.

zgrid: tuple of Surfaces or a Surface object Direction of the ray can be defined by a vector object (assumed to be in the space of the pt1 origin) from pt1, or a point, which generates a vector pointing from pt1 to pt2.

Returns: output: tuple of beam objects.

Examples: Accepts all surface or surface-derived object inputs, though all data is stored as a python object.

Generate an y direction Ray in cartesian coords using a Vec from (0,0,1):

```
cen = geometry.Center(flag=True)
ydir = geometry.Vecx((0,1,0))
zpt = geometry.Point((0,0,1),cen)
```

CHAPTER

NINE

TRIPPY.GEOMETRY MODULE

class `TRIPPy.geometry.`**`Center`**(*flag=True*)

Bases: `TRIPPy.geometry.Origin`

Center object with inherent cartesian backend mathematics.

Creates a new Center instance which can be set to a default coordinate system of cartesian or cylindrical coordinates. All vector mathematics are accomplished in cartesian to simplify computation effort. Cylindrical vectors are converted at last step for return. It defaults to cylindrical coordinates

The Center class which underlies all positional calculation. It is located at (0,0,0) and is inherently a cylindrical coordinate system (unless flag set otherwise). It is from the translation of inherently cylindrical data into toroidal coordinates requires this rosetta stone, it can be dynamically set to becoming an origin given a specification of another origin.

Kwargs:

flag: Boolean. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate a Center in cylindrical coordinates:

cent = Center() #implicitly in cyl. coords.

Generate a Center in cartesian coordinates:

cent = Center(flag=False)

`meri` **= <TRIPPy.geometry.Vec object at 0x3e73c90>**

`norm` **= <TRIPPy.geometry.Vec object at 0x3e73cd0>**

`sagi` **= <TRIPPy.geometry.Vec object at 0x3e73bd0>**

class `TRIPPy.geometry.`**`Origin`**(*x_hat*, *ref=None*, *vec=None*, *angle=None*, *flag=None*)

Bases: `TRIPPy.geometry.Point`

Origin object with inherent cartesian backend mathematics.

Creates a new Origin instance which can be set to a default coordinate system of cartesian or cylindrical coordinates. All vector mathematics are accomplished in cartesian to simplify computation effort. Cylindrical vectors are converted at last step for return.

An Origin is defined by a point and two vectors. The two vectors being: 1st the normal to the surface, principal axis, z-vector or the (0,0,1) vector of the new system defined in the reference system. The second vector along with the first fully defines meridonial ray paths, y-vector or the (0,1,0) vector (.meri). The sagittal ray path, x-vector or the (1,0,0) is defined through a cross product (.sagi). Point position and rotation matricies are stored at instantiation.

These conventions of norm to z, meri to y axis and sagi to x axis are exactly as perscribed in the OSLO optical code, allowing for easier translation from its data into Toroidal systems.

If the angles alpha, beta, and gamma are specified following the eulerian rotation formalism, it is processed in the following manner: alpha is the rotation from the principal axis in the meridonial plane, beta is the rotation about the plane normal to the meridonial ray, or 2nd specified vector, and gamma is the 2nd rotation about the principal axis. This might change based on what is most physically intuitive. These are converted to vectors and stored as attributes.

Args:

x_hat: geometry-derived object or Array-like of size 3 or 3xN. Position in the coordinate system defined by origin, which, if it is a scipy array, follows the input convention of the origin. Specified vector will be converted as necessary.

Kwargs:

ref: Origin or Origin-derived object. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

vec: Tuple of two Vector objects The two vectors describe the normal (or z) axis and the meridonial (or y) axis. Inputs should follow [meri,normal]. If not specified, it assumed that angle is specified.

angle: tuple or array of 3 floats alpha, beta and gamma are eulerian rotation angles which describe the rotation and thus the sagittal and meridonial rays.

flag: Boolean. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. newy = Vecr((1.,scipy.pi,0.)) z = Vecr((0.,0.,1.)) ex = Origin((0.,0.,0.), cent, vec=[newy,z])

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. ex1 = Origin((0.,0.,0.), cent, angle=(scipy.pi/2,0.,0.))

Generate an origin at (1,10,-7) with a cartesian coord system:

cent = Center() #implicitly in cyl. coords. place = Vecx((1.,10.,-7.)) ex2 = Origin(place, cent, angle=(0.,0.,0.), flag=False)

Generate an origin at (1,1,1) with a cartesian coord system:

cent = Center(flag=False) #cartesian coords. ex3 = Origin((1.,1.,1.), cent, angle=(0.,0.,0.))

`arot` (*vec*)

Rotate vector out of coordinates of Origin. (Anti-Rotate)

Inverse of rot() function

Args: vec: Vector or Vector-derived object

`redefine` (*neworigin*)

redefine Origin object or Origin-derived object into new coordinate system

Args: neworigin: Origin or Origin-derived object

`rot` (*vec*)

Rotate input vector objects into coordinates of Origin.

Args: vec: Vector object

spin (*angle*)
Spin vector or vector-derived object around Origin about the cylindrical (0,0,1)/norm vector axis. This function is different from rot.

Args: vec: Vector or Vector-derived object

split (**args*, ***kwargs*)
split coordinate values into seperate objects

Kwargs:

obj: geometry-derived object which to form from data. If not specified, returns a tuple of Origin objects.

Returns: Object tuple size N for N points. Works to arbitrary dimension.

class `TRIPPy.geometry.`**`Point`** (*x_hat*, *ref=None*)
Bases: `TRIPPy.geometry.Vec`

Point object with inherent cartesian backend mathematics.

Creates a new Point instance which is set to a default coordinate system of cartesian or cylindrical coordinates determined from the reference coordinate system. All vector mathematics are accomplished in cartesian to simplify computation effort. Cylindrical vectors are converted at last step for return.

Args:

x_hat: geometry object or 3xN coordinate system values. Input coordinate system value options assume that it matches the coordinate system of the reference origin.

Kwargs:

ref: Origin object. Sets the default coordinate nature

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate a point 1m in x from point U (xdir):

```
xdir = Vec((1.,0.,0.),U) #implicitly in cyl. coords.
```

Generate a cartesian point into direction (2,2,2) from Q:

```
vec2 = Vec((2.,2.,2.), ref=Q)
```

Generate a cylindrical radial vector (vec2) into direction (1,0,1):

```
cent = Center() #implicitly in cyl. coords.
vec2 = Vec(vec1, ref=cent)
```

c ()

redefine (*neworigin*)
redefine Point object or Point-derived object into new coordinate system

Args: neworigin: Origin or Origin-derived object

split (**args*, ***kwargs*)
split coordinate values into seperate objects

Kwargs: obj: geometry-derived object which to form from data

Returns: Object tuple size N for N points. Works to arbitrary dimension.

class `TRIPPy.geometry.`**`Vec`** (*x_hat*, *s*, *flag=False*)
Bases: `object`

Vector object with inherent cartesian backend mathematics.

Creates a new Vec instance which can be set to a default coordinate system of cartesian or cylindrical coordinates. All vector mathematics are accomplished in cartesian to simplify computation effort. Cylindrical vectors are converted at last step for return.

It is highly recommended to utilize the Vecx and Vecr functions which allow for proper data checks in generating vectors.

Args:

x_hat: Array-like of size 3 or 3xN in cartesian. for all i, x_hat[0][i]**2 + x_hat[1][i]**2 + x_hat[2][i]**2 is equal to 1.

s: Array-like of size 1 or shape N. Values of the positions of the 2nd dimension of f. Must be monotonic without duplicates.

Kwargs:

flag: Boolean. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate an x direction unit vector (xdir):

```
xdir = Vec((1.,0.,0.),1.0)
```

Generate a cartesian vector (vec1) into direction (2,2,2):

```
vec1 = Vec(scipy.array([1.,1.,1.])/3.0,scipy.array(2.0))
```

Generate a cartesian vector (vec2) into direction (4,4,4):

```
vec2 = Vec(vec1.unit,vec1.s*2)
```

c()

Conversion of vector to opposite coordinate system

Returns: copy of vector object with opposite coordinate system (set with .flag parameter)

copy()

copy of object

Returns: copy of object

point(*ref*)

returns point based off of vector

Args: ref: reference origin

Returns: Point object

r()

return cylindrical coordinate values

Returns: numpy array of cylindrical coordinates in meters and radians

r0()

returns cylindrical coordinate along first dimension

Returns: numpy array of cylindrical coordinates in meters

r1()

returns cylindrical coordinate along second dimension

Returns: numpy array of cylindrical coordinates in radians

r2()
returns cylindrical coordinate along third dimension

Returns: numpy array of cylindrical coordinates in meters

spin(*angle*)
Spin vector object about the cylindrical (0,0,1)/norm vector axis. This function is different from rot.

Args: angle: Singule element float or scipy array.

t(*r*, *z*)
return toroidal coordinate values for given cylindrical coordinates (r,z).

Args:

r: scipy-array of floats or float in meters. r is specified in meters.

z: scipy-array of floats or float in meters. z is specified in meters.

Returns: numpy array of cylindrical coordinates in [meters,radians,radians] where it is radius in meters, toroidal angle and then poloidal angle.

t0(*r*, *z*)
returns toroidal distance given cylindrical coordinates (r,z).

Args:

r: scipy-array of floats or float in meters. r is specified in meters.

z: scipy-array of floats or float in meters. z is specified in meters.

Returns: numpy array of cylindrical coordinates in meters

t2(*r*, *z*)
returns poloidal angle given cylindrical coordinates (r,z)

Args:

r: scipy-array of floats or float in meters. r is specified in meters.

z: scipy-array of floats or float in meters. z is specified in meters.

Returns: numpy array of cylindrical coordinates in meters

x()
return cartesian coordinate values

Returns: numpy array of cartesian coordinates in meters

x0()
returns cartesian coordinate along first dimension

Returns: numpy array of cartesian coordinates in meters

x1()
returns cartesian coordinate along second dimension

Returns: numpy array of cartesian coordinates in meters

x2()
returns cartesian coordinate along third dimension

Returns: numpy array of cartesian coordinates in meters

TRIPPy.geometry.**Vecr**(*x_hat*, *s=None*)
Generates a cylindrical coordinate vector object

Uses the definition:

$$\vec{x} = \texttt{xhat}[0]\hat{r} + \texttt{xhat}[1]\hat{\theta} + \texttt{xhat}[2]\hat{z}$$

Capable of storing multiple directions and lengths as a single vector, but highly discouraged (from POLS).

Args:

x_hat: Array-like, 3 or 3xN. 3 dimensional cylindrical vector input, which is stores the direction and magnitude as seperate values. All values of theta will be aliased to $(\pi, \pi]$

Kwargs:

s: Array-like or scalar float. Vector magnitude in meters. When specified, it is assumed that x_hat is representative of direction only and is of unit length. Saves in computation as length calculation avoided.

Returns: Vec: Vector object.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate an y direction unit vector in cylindrical coords (ydir):

```
ydir = Vecr((1.,scipy.pi/2,0.))
```

Generate a cartesian vector (vec1) into direction (1,pi/3,-4):

```
vec1 = Vecr(scipy.array([1.,scipy.pi/3,-4.]))
```

Generate a cartesian vector (vec2) into direction (6,0,8):

```
vec2 = Vecr(scipy.array([3.,0.,4.])/5.0,s=scipy.array(10.0))
```

Generate a cartesian vector (vec3) into direction (.3,0,.4):

```
vec3 = Vecr(vec2.r()/vec2.s,s=scipy.array(.1))
```

`TRIPPy.geometry.`**`Vecx`**(*x_hat*, *s=None*)

Generates a cartesian coordinate vector object

Uses the definition:

$$\vec{x} = \texttt{xhat}[0]\hat{x} + \texttt{xhat}[1]\hat{y} + \texttt{xhat}[2]\hat{z}$$

Capable of storing multiple directions and lengths as a single vector, but highly discouraged (from POLS).

Args:

x_hat: Array-like, 3 or 3xN. 3 dimensional cartesian vector input, which is stores the direction and magnitude as seperate values.

Kwargs:

s: Array-like or scalar float. Vector magnitude in meters. When specified, it is assumed that x_hat is representative of direction only and is of unit length. Saves in computation as length calculation is avoided.

Returns: Vec: Vector object.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate an x direction unit vector (xdir):

```
xdir = Vecx((1.,0.,0.))
```

Generate a cartesian vector (vec1) into direction (1,3,-4):

```
vec1 = Vecx(scipy.array([1.,3.,-4.]))
```

Generate a cartesian vector (vec2) into direction (2,2,2):

```
vec2 = Vecx(scipy.array([1.,1.,1.])/3.0,s=scipy.array(2.0))
```

Generate a cartesian vector (vec3) into direction (3,3,3):

```
vec3 = Vecx(vec2.unit,s=scipy.array(3.0))
```

`TRIPPy.geometry.`**`angle`**(*Vec1*, *Vec2*)

Returns angle between two vectors.

Args: Vec1: Vec Object

Vec2: Vec Object

Returns: angle in radians $[0, \pi]$ seperating the two vectors on the plane defined by the two.

`TRIPPy.geometry.`**`cross`**(*Vec1*, *Vec2*)

Returns angle between two vectors.

Args: Vec1: Vec Object

Vec2: Vec Object

Returns: Vec object of the vector cross product of the two vectors. It is in the coordinate system of the first argument.

`TRIPPy.geometry.`**`fill`**(*funtype*, *x0*, *x1*, *x2*, **args*, ***kwargs*)

Recursive function to generate TRIPPy Objects

Args: funtype: Object type to replicate

x0: coordinate of 1st direction

x1: coordinate of 2md direction

x2: coordinate of 3rd direction

Returns: Tuple or object of type funtype.

`TRIPPy.geometry.`**`pts2Vec`**(*pt1*, *pt2*)

Returns angle between two vectors.

Args: pt1: geometry Object with reference origin

pt2: geometry Object with reference origin

Returns: Vector object: Vector points from pt1 to pt2.

`TRIPPy.geometry.`**`unit`**(*x_hat*)

Checks vector input to be of correct array dimensionality.

Numpy array dimensionality can often create unexpected array sizes in multiplications. Unit also forces the input to follow unit vector conventions. It checks the expected cartesian unit vector to be 3xN, and that all elements of x_hat are within the range [-1,1]

Args:

x_hat: Array-like, 3 or 3xN. 3 dimensional cartesian vector input, which is stores the direction and magnitude as seperate values.

Returns: Vec: Vector object.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Check [1.,0.,0.]:

```
xdir = unit([1.,0.,0.])
```

Check [[1.,2.],[0.,0.],[0.,0]]:

```
xdir = unit([[1.,2.],[0.,0.],[0.,0]])
```

Raises:

ValueError: If any of the dimensions of x_hat are not 3xN or not of unit length.

CHAPTER TEN

TRIPPY.INVERT MODULE

`TRIPPy.invert.`**`rhosens`**(*beams*, *plasma*, *time*, *points*, *step=0.001*, *meth='psinorm'*)

optimal to give multiple times

`TRIPPy.invert.`**`sens`**(*beams*, *plasmameth*, *time*, *points*, *step=0.001*)

optimal to give multiple times

TRIPPY.PLASMA MODULE

class `TRIPPy.plasma.`**`Tokamak`**(*equilib*, *flag=True*)

Bases: `TRIPPy.geometry.Center`

`getMachineCrossSection`()

`getVessel`(*idx*, *pts=250*)

`gridWeight`(*beams*, *rgrid=None*, *zgrid=None*, *spacing=0.001*)

this method finds the weighting of the view on a poloidal plane it takes a beam and traces it through the plasma

`inVessel`(*ptin*)

`pnt2RhoTheta`(*point*, *t=0*, *method='psinorm'*, *n=0*, *poloidal_plane=0*)

takes r,theta,z and the plasma, and map it to the toroidal position this will be replaced by a toroidal point generating system based of a seperate vector class

`trace`(*ray*, *limiter=0*)

CHAPTER

TWELVE

TRIPPY.SURFACE MODULE

class `TRIPPy.surface.`**`Circle`**(*x_hat*, *ref*, *radius*, *vec=None*, *angle=None*, *flag=None*)

Bases: `TRIPPy.surface.Ellipse`

Origin object with inherent cartesian backend mathematics.

Creates a new Origin instance which can be set to a default coordinate system of cartesian or cylindrical coordinates. All vector mathematics are accomplished in cartesian to simplify computation effort. Cylindrical vectors are converted at last step for return.

An Origin is defined by a point and two vectors. The two vectors being: 1st the normal to the surface, principal axis, z-vector or the (0,0,1) vector of the new system defined in the reference system. The second vector along with the first fully defines meridonial ray paths, y-vector or the (0,1,0) vector (.meri). The sagittal ray path, x-vector or the (1,0,0) is defined through a cross product (.sagi). Point position and rotation matricies are stored at instantiation.

These conventions of norm to z, meri to y axis and sagi to x axis are exactly as perscribed in the OSLO optical code, allowing for easier translation from its data into Toroidal systems.

If the angles alpha, beta, and gamma are specified following the eulerian rotation formalism, it is processed in the following manner: alpha is the rotation from the principal axis in the meridonial plane, beta is the rotation about the plane normal to the meridonial ray, or 2nd specified vector, and gamma is the 2nd rotation about the principal axis. This might change based on what is most physically intuitive. These are converted to vectors and stored as attributes.

Args: x_hat: geometry-derived object or Array-like of size 3 or 3xN.

Kwargs:

ref: Origin or Origin-derived object. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

vec: Tuple of two Vector objects The two vectors describe the normal (or z) axis and the meridonial (or y) axis. Inputs should follow [meri,normal]. If not specified, it assumed that angle is specified.

angle: tuple or array of 3 floats alpha, beta and gamma are eulerian rotation angles which describe the rotation and thus the sagittal and meridonial rays.

flag: Boolean. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. newy = Vecr((1.,scipy.pi,0.)) z = Vecr((0.,0.,1.)) ex = Origin((0.,0.,0.), cent, vec=[newy,z])

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. ex1 = Origin((0.,0.,0.), cent, angle=(scipy.pi/2,0.,0.))

Generate an origin at (1,10,-7) with a cartesian coord system:

cent = Center() #implicitly in cyl. coords. place = Vecx((1.,10.,-7.)) ex2 = Origin(place, cent, angle=(0.,0.,0.), flag=False)

Generate an origin at (1,1,1) with a cartesian coord system:

cent = Center(flag=False) #cartesian coords. ex3 = Origin((1.,1.,1.), cent, angle=(0.,0.,0.))

area (*radius=None*)

edgetest (*radius=None*)

class TRIPPy.surface.**Ellipse** (*x_hat*, *ref*, *area*, *vec=None*, *angle=None*, *flag=None*)

Bases: TRIPPy.surface.Surf

Origin object with inherent cartesian backend mathematics.

Creates a new Origin instance which can be set to a default coordinate system of cartesian or cylindrical coordinates. All vector mathematics are accomplished in cartesian to simplify computation effort. Cylindrical vectors are converted at last step for return.

An Origin is defined by a point and two vectors. The two vectors being: 1st the normal to the surface, principal axis, z-vector or the (0,0,1) vector of the new system defined in the reference system. The second vector along with the first fully defines meridonial ray paths, y-vector or the (0,1,0) vector (.meri). The sagittal ray path, x-vector or the (1,0,0) is defined through a cross product (.sagi). Point position and rotation matricies are stored at instantiation.

These conventions of norm to z, meri to y axis and sagi to x axis are exactly as perscribed in the OSLO optical code, allowing for easier translation from its data into Toroidal systems.

If the angles alpha, beta, and gamma are specified following the eulerian rotation formalism, it is processed in the following manner: alpha is the rotation from the principal axis in the meridonial plane, beta is the rotation about the plane normal to the meridonial ray, or 2nd specified vector, and gamma is the 2nd rotation about the principal axis. This might change based on what is most physically intuitive. These are converted to vectors and stored as attributes.

Args: x_hat: geometry-derived object or Array-like of size 3 or 3xN.

Kwargs:

ref: Origin or Origin-derived object. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

vec: Tuple of two Vector objects The two vectors describe the normal (or z) axis and the meridonial (or y) axis. Inputs should follow [meri,normal]. If not specified, it assumed that angle is specified.

angle: tuple or array of 3 floats alpha, beta and gamma are eulerian rotation angles which describe the rotation and thus the sagittal and meridonial rays.

flag: Boolean. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. newy = Vecr((1.,scipy.pi,0.)) z = Vecr((0.,0.,1.)) ex = Origin((0.,0.,0.), cent, vec=[newy,z])

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. ex1 = Origin((0.,0.,0.), cent, angle=(scipy.pi/2,0.,0.))

Generate an origin at (1,10,-7) with a cartesian coord system:

cent = Center() #implicitly in cyl. coords. place = Vecx((1.,10.,-7.)) ex2 = Origin(place, cent, angle=(0.,0.,0.), flag=False)

Generate an origin at (1,1,1) with a cartesian coord system:

cent = Center(flag=False) #cartesian coords. ex3 = Origin((1.,1.,1.), cent, angle=(0.,0.,0.))

area (*sagi=None*, *meri=None*)

edge (*angle=[0, 6.283185307179586]*, *pts=250*)

edgetest (*meri*, *sagi*)

class `TRIPPy.surface.`**`Rect`** (*x_hat*, *ref*, *area*, *vec=None*, *angle=None*, *flag=None*)

Bases: `TRIPPy.surface.Surf`

Origin object with inherent cartesian backend mathematics.

Creates a new Origin instance which can be set to a default coordinate system of cartesian or cylindrical coordinates. All vector mathematics are accomplished in cartesian to simplify computation effort. Cylindrical vectors are converted at last step for return.

An Origin is defined by a point and two vectors. The two vectors being: 1st the normal to the surface, principal axis, z-vector or the (0,0,1) vector of the new system defined in the reference system. The second vector along with the first fully defines meridonial ray paths, y-vector or the (0,1,0) vector (.meri). The sagittal ray path, x-vector or the (1,0,0) is defined through a cross product (.sagi). Point position and rotation matricies are stored at instantiation.

These conventions of norm to z, meri to y axis and sagi to x axis are exactly as perscribed in the OSLO optical code, allowing for easier translation from its data into Toroidal systems.

If the angles alpha, beta, and gamma are specified following the eulerian rotation formalism, it is processed in the following manner: alpha is the rotation from the principal axis in the meridonial plane, beta is the rotation about the plane normal to the meridonial ray, or 2nd specified vector, and gamma is the 2nd rotation about the principal axis. This might change based on what is most physically intuitive. These are converted to vectors and stored as attributes.

Args: x_hat: geometry-derived object or Array-like of size 3 or 3xN.

Kwargs:

ref: Origin or Origin-derived object. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

vec: Tuple of two Vector objects The two vectors describe the normal (or z) axis and the meridonial (or y) axis. Inputs should follow [meri,normal]. If not specified, it assumed that angle is specified.

angle: tuple or array of 3 floats alpha, beta and gamma are eulerian rotation angles which describe the rotation and thus the sagittal and meridonial rays.

flag: Boolean. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. newy = Vecr((1.,scipy.pi,0.)) z = Vecr((0.,0.,1.)) ex = Origin((0.,0.,0.), cent, vec=[newy,z])

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. ex1 = Origin((0.,0.,0.), cent, angle=(scipy.pi/2,0.,0.))

Generate an origin at (1,10,-7) with a cartesian coord system:

cent = Center() #implicitly in cyl. coords. place = Vecx((1.,10.,-7.)) ex2 = Origin(place, cent, angle=(0.,0.,0.), flag=False)

Generate an origin at (1,1,1) with a cartesian coord system:

cent = Center(flag=False) #cartesian coords. ex3 = Origin((1.,1.,1.), cent, angle=(0.,0.,0.))

area (*sagi=None*, *meri=None*)

edge ()
return points at the edge of rectangle

edgetest (*sagi*, *meri*)

split (*sagi*, *meri*)
utilizes geometry.grid to change the rectangle into a generalized surface, it is specified with a single set of basis vectors to describe the meridonial, normal, and sagittal planes.

class TRIPPy.surface.**Surf** (*x_hat*, *ref*, *area*, *vec=None*, *angle=None*, *flag=None*)
Bases: TRIPPy.geometry.Origin

Surf object with inherent cartesian backend mathematics.

Creates a new Surf instance which can be set to a default coordinate system of cartesian or cylindrical coordinates. All vector mathematics are accomplished in cartesian to simplify computation effort. Cylindrical vectors are converted at last step for return.

A surface is defined by a point and two vectors. The two vectors being: 1st the normal to the surface, principal axis, z-vector or the (0,0,1) vector of the new system defined in the reference system. The second vector along with the first fully defines meridonial ray paths, y-vector or the (0,1,0) vector (.meri). The sagittal ray path, x-vector or the (1,0,0) is defined through a cross product (.sagi). Center position and rotation matricies are stored at instantiation.

These conventions of norm to z, meri to y axis and sagi to x axis are exactly as perscribed in the OSLO optical code, allowing for easier translation from its data into Toroidal systems.

The surface cross-sectional area is specified and used to modify the sagittal and meridonial vectors. This is stored as lengths of the two vectors, meri and sagi. This object assumes that the surface is purely normal to the input norm vector across the entire surface.

If the angles alpha, beta, and gamma are specified following the eulerian rotation formalism, it is processed in the following manner: alpha is the rotation from the principal axis in the meridonial plane, beta is the rotation about the plane normal to the meridonial ray, or 2nd specified vector, and gamma is the 2nd rotation about the principal axis. This might change based on what is most physically intuitive. These are converted to vectors and stored as attributes.

Args: x_hat: geometry-derived object or Array-like of size 3 or 3xN.

ref: Origin or Origin-derived object. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

area: 2-element tuple of scipy-arrays or values. This sets the cross-sectional area of the view, which follows the convenction [sagi,meri]. Meri is the length in the direction of the optical axis, and sagi is the length in the off axis of the optical and normal axes. Values are in meters.

Kwargs:

vec: Tuple of two Vec objects The two vectors describe the normal (or z) axis and the meridonial (or y) axis. Inputs should follow [meri,normal]. If not specified, it assumed that angle is specified.

angle: tuple or array of 3 floats alpha, beta and gamma are eulerian rotation angles which describe the rotation and thus the sagittal and meridonial rays.

flag: Boolean. Sets the default coordinate nature of the vector to cartesian if False, or cylindrical if True.

Examples: Accepts all array like (tuples included) inputs, though all data is stored in numpy arrays.

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. newy = Vecr((1.,scipy.pi,0.)) z = Vecr((0.,0.,1.)) ex = Origin((0.,0.,0.), cent, vec=[newy,z])

Generate an origin at (0,0,0) with a $\pi/2$ rotation:

cent = Center() #implicitly in cyl. coords. ex1 = Origin((0.,0.,0.), cent, angle=(scipy.pi/2,0.,0.))

Generate an origin at (1,10,-7) with a cartesian coord system:

cent = Center() #implicitly in cyl. coords. place = Vecx((1.,10.,-7.)) ex2 = Origin(place, cent, angle=(0.,0.,0.), flag=False)

Generate an origin at (1,1,1) with a cartesian coord system:

cent = Center(flag=False) #cartesian coords. ex3 = Origin((1.,1.,1.), cent, angle=(0.,0.,0.))

intercept(*ray*)

Solves for intersection point of surface and a ray or Beam

Args:

ray: Ray or Beam object It must be in the same coordinate space as the surface object.

Returns: s: value of s [meters] which intercepts along norm, otherwise an empty tuple (for no intersection).

Examples: Accepts all point and point-derived object inputs, though all data is stored as a python object.

Generate an y direction Ray in cartesian coords using a Vec from (0,0,1):

```
cen = geometry.Center(flag=True)
ydir = geometry.Vecx((0,1,0))
zpt = geometry.Point((0,0,1),cen)
```

CHAPTER THIRTEEN

MODULE CONTENTS

t

R

S

T

U

V

W

X