```
'''
Scott Carnahan
A ray tracing library I wrote for 5760
Spring 2019
'''

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from Basilisk.utilities import RigidBodyKinematics as rbk

def quadratic_eqn(t, *args):
    # given the independent variable, and args=(a,b,c)
    # returns at2 + bt + c
    # used for numerical solution of quadratic equation
    # don't use this anymore.
    return args[0] * t**2 + args[1] * t + args[2]

def solve_quadratic(a, b, c, pm):
    # standard fo rm of the quadratic equation
    if pm == 1:
        return (-b + np.sqrt(b**2. - 4. * a * c)) / 2. / a
    elif pm == -1:
        return (-b - np.sqrt(b**2. - 4. * a * c)) / 2. / a

def mullers_quadratic_equation(a, b, c, pm):
    # an alternative form of the quadratic equation that doesn't
    # fail if a == 0.
    # useful if rays are coming straight in at a parabolic mirror
    if pm == 1:
        return (2 * c) / (-b + np.sqrt(b**2 - 4. * a * c))
    elif pm == -1:
        return (2 * c) / (-b - np.sqrt(b**2 - 4. * a * c))

class parabolicMirrorWithHole:
    # a symmetric paraboloid
    # a is the scaling factor. a(x2 + y2) = z
    # outer diam is the mirror size
    #inner diam allows for a centered hole in the mirror
    # S is the surface frame
    # L is the lab frame
    # e212 is the euler 2-1-2 rotation describing the orientation of
the surface wrt the lab frame.
    # L_r_L is the lab-frame-resolved position of the vertex of the
paraboloid
    def __init__(self, a, outer_diam, inner_diam, e212, L_r_L):
        self.a = a # slope
        self.outer_diam = outer_diam
        self.inner_diam = inner_diam
        self.max_z = 10.
        self.min_z = 0.
        self.set_limits()
        self.DCM_SL = rbk.euler2122C(e212)
        self.L_r_L = L_r_L.reshape([3, 1])
        self.S_focus = np.array([0., 0., 1 / 4. / a])
        self.L_focus = np.dot(self.DCM_SL.transpose(), np.array([0., 0
., 1 / 4. / a])) + self.L_r_L.reshape([3, ])
        self.name = 'primary'

    def equation(self, xs, ys):
        # defining equation for a paraboloid
```

```
 59         # useful for meshing the surface to plot
 60         return self.a * (xs **2 + ys**2)
 61 
 62     def focus(self):
 63         # focus at 1/(4a)
 64         f = np.array([0., 0., 1. / 4. / self.a])
 65         return np.dot(self.DCM_SL.transpose(), f)
 66 
 67     def normal(self, L_X):
 68         # given points on the paraboloid, gives unit normal vectors.
 69         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
 70         xs = S_X[0, :]
 71         ys = S_X[1, :]
 72         num = np.shape(ys)
 73         S_N = np.array([2 * self.a * xs, 2 * self.a * ys, -np.ones(
    num)])
 74         S_N_hat = S_N / np.linalg.norm(S_N, axis=0)
 75         L_N_hat = np.dot(self.DCM_SL.transpose(), S_N_hat)
 76         return L_N_hat
 77 
 78     def set_limits(self):
 79         # because everything is done in the surface frame
 80         # I can rule out hits by their z-value
 81         # this sets the mirror max/min z
 82         rad = self.outer_diam / 2.
 83         self.max_z = self.a * rad ** 2
 84         rad = self.inner_diam / 2.
 85         self.min_z = self.a * rad ** 2
 86         return
 87 
 88     def surface_points(self, n=50):
 89         # gives X, Y, Z points for the surface to be scatter-plotted
 90         xs = np.linspace(-self.outer_diam/2, self.outer_diam/2, n).
    tolist()
 91         x_out = []
 92         ys = np.linspace(-self.outer_diam / 2, self.outer_diam /2, n)
    .tolist()
 93         y_out = []
 94         z_out = []
 95         for x in xs:
 96             for y in ys:
 97                 x_out.append(x)
 98                 y_out.append(y)
 99                 z_out.append(self.equation(x, y))
100         x_out = np.array(x_out)
101         y_out = np.array(y_out)
102         z_out = np.array(z_out)
103         x_out = x_out[(z_out <= self.max_z) & (z_out >= self.min_z)]
104         y_out = y_out[(z_out <= self.max_z) & (z_out >= self.min_z)]
105         z_out = z_out[(z_out <= self.max_z) & (z_out >= self.min_z)]
106         X = np.vstack([x_out, y_out, z_out])
107         X = np.dot(self.DCM_SL.transpose(), X)
108         X = X + self.L_r_L
109         return X
110 
111     def surface_mesh(self):
112         # give x, y, z points for the surface to be surface plotted
113         out_rad = self.outer_diam / 2.
114         in_rad = self.inner_diam / 2.
115         rad_points = np.linspace(in_rad, out_rad, 2)
116         theta_points = np.linspace(0., np.pi * 2., 20)
```

```
117         R, T = np.meshgrid(rad_points, theta_points)
118         X, Y = R * np.cos(T), R * np.sin(T)
119         Z = self.equation(X, Y)
120         for i in range(np.shape(Z)[0]): # transpose into lab frame
121             for j in range(np.shape(Z)[1]):
122                 vec = np.array([X[i,j], Y[i,j], Z[i,j]])
123                 vec = np.dot(self.DCM_SL.transpose(), vec)
124                 vec = vec + self.L_r_L.reshape(3,)
125                 X[i,j], Y[i,j], Z[i, j] = vec[0], vec[1], vec[2]
126         return X, Y, Z
127 
128     def intersect_rays(self, L_X_0, L_X_d):
129         # takes in ray starts and direction unit vectors in lab frame
130         num = np.shape(L_X_0)[1]
131         S_X_0 = np.dot(self.DCM_SL, L_X_0 - self.L_r_L)
132         S_X_d = np.dot(self.DCM_SL, L_X_d)
133         x0s = S_X_0[0, :]
134         xds = S_X_d[0, :]
135         y0s = S_X_0[1, :]
136         yds = S_X_d[1, :]
137         z0s = S_X_0[2, :]
138         zds = S_X_d[2, :]
139         A = yds ** 2 + xds ** 2
140         B = 2 * (x0s * xds + y0s * yds) - zds / self.a
141         C = x0s**2 + y0s**2 -z0s / self.a
142         non_nan = ~np.isnan(x0s)
143         ts = mullers_quadratic_equation(A[non_nan], B[non_nan], C[
    non_nan], -1)
144         S_X_1 = S_X_0
145         S_X_1[:, non_nan] = S_X_0[:, non_nan] + ts * S_X_d[:, non_nan
    ]
146         L_X_1 = np.dot(self.DCM_SL.transpose(), S_X_1) + self.L_r_L
147         return L_X_1
148 
149     def miss_rays(self, L_X):
150         # L_X is an intersection point for a parabolloid that is
    infinite with no holes
151         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
152         zs = S_X[2, :]
153         zs[np.isnan(zs)] = -1.
154         temp = S_X.transpose()
155         temp[(zs > self.max_z) | (zs < self.min_z)] = np.array([np.
    nan] * 3)
156         S_X = temp.transpose()
157         L_X = np.dot(self.DCM_SL.transpose(), S_X) + self.L_r_L
158         return L_X
159 
160     def reflect_rays(self, L_X, L_d_i):
161         # takes an intersect point L_X and incoming direction L_d_i
162         # produces an outgoing direction L_d_o
163         # is there a better vectorized way of doing this?
164         num = np.shape(L_X)[1]
165         L_N_hat = self.normal(L_X)
166         L_d_o = []
167         for i in range(num):
168             incoming = L_d_i[:,i]
169             nHat = L_N_hat[:,i]
170             M = np.eye(3) -2 * np.outer(nHat, nHat)
171             L_d_o.append(np.dot(M, incoming))
172         return np.array(L_d_o).transpose()  # just put it back into
    3xN array format
```

```
173
174 class circleOfDeath:
175     # a circular surface that kills rays (back of a mirror perhaps)
176     # defined in the x-y frame with a displacement and rotation
177     def __init__(self, r, L_r_L, e212):
178         self.r = r
179         self.L_r_L = L_r_L.reshape([3, 1])
180         self.DCM_SL = rbk.euler2122C(e212)
181         self.name = "dead_spot"
182
183     def miss_rays(self, L_X_0):
184         return L_X_0
185
186     def surface_points(self, num=10):
187         rs = np.linspace(0, self.r, num)
188         ts = np.linspace(0, 2 * np.pi, num)
189         L_points = []
190         for r in rs:
191             for t in ts:
192                 x = r * np.cos(t)
193                 y = r * np.sin(t)
194                 if x**2 + y**2 > self.r:
195                     L_points.append(np.array([np.nan] * 3))
196                 else:
197                     S_point = np.array([x, y, 0])
198                     L_points.append(np.dot(self.DCM_SL.transpose(),
    S_point) + self.L_r_L.reshape([3, ]))
199         return np.array(L_points).transpose()
200
201     def surface_mesh(self):
202         out_rad = self.r
203         rad_points = np.linspace(0., out_rad, 2)
204         theta_points = np.linspace(0., np.pi * 2., 30)
205         R, T = np.meshgrid(rad_points, theta_points)
206         X, Y = R * np.cos(T), R * np.sin(T)
207         Z = np.zeros(np.shape(X))
208         for i in range(np.shape(Z)[0]):
209             for j in range(np.shape(Z)[1]):
210                 vec = np.array([X[i,j], Y[i,j], Z[i,j]])
211                 vec = np.dot(self.DCM_SL.transpose(), vec)
212                 vec = vec + self.L_r_L.reshape(3,)
213                 X[i,j], Y[i,j], Z[i, j] = vec[0], vec[1], vec[2]
214         return X, Y, Z
215
216     def reflect_rays(self, L_X, L_d):
217         return L_d
218
219     def intersect_rays(self, L_X_0, L_X_d):
220         # takes in ray starts and direction unit vectors in lab frame
221         num = np.shape(L_X_0)[1]
222         S_X_0 = np.dot(self.DCM_SL, L_X_0 - self.L_r_L)
223         S_X_d = np.dot(self.DCM_SL, L_X_d)
224         x0s = S_X_0[0, :]
225         xds = S_X_d[0, :]
226         y0s = S_X_0[1, :]
227         yds = S_X_d[1, :]
228         z0s = S_X_0[2, :]
229         zds = S_X_d[2, :]
230         ts = -z0s / zds
231         xs = x0s + ts * xds
232         ys = y0s + ts * yds
```

```
233         S_X_1 = S_X_0 + ts * S_X_d
234         temp = S_X_1.transpose()
235         temp[xs**2. + ys**2. <= self.r ** 2] = np.array([np.nan] * 3)
236         S_X_1 = temp.transpose()
237         L_X_1 = np.dot(self.DCM_SL.transpose(), S_X_1) + self.L_r_L
238         return L_X_1
239
240 class ConvexHyperbolicMirror:
241     def __init__(self, b, a, diam, L_r_L, e212):
242         # (x2 + y2)/b2 - z2/a2 = -1 and take the positive solution
243         # where b**2 = f**2 - a**2. and f is the focal distance.
244         self.a = a
245         self.b = b
246         self.diam = diam
247         self.max_z = 10.
248         self.min_z = 0.
249         self.L_r_L = L_r_L.reshape([3, 1])
250         self.DCM_SL = rbk.euler2122C(e212)
251         self.set_limits()
252         self.S_focus = np.zeros(3)
253         self.L_focus = np.zeros(3)
254         self.set_focus()
255         self.name = 'secondary'
256
257     def set_focus(self):
258         f = np.sqrt(self.a**2 + self.b**2)
259         self.S_focus = np.array([0., 0., f])
260         self.L_focus = np.dot(self.DCM_SL.transpose(), self.S_focus)
    + self.L_r_L.reshape([3, ])
261         return
262
263     def set_limits(self):
264         rad = self.diam / 2.
265         self.max_z = self.equation(rad, 0)
266         self.min_z = self.a
267         return
268
269     def surface_points(self, n=50):
270         xs = np.linspace(-self.diam / 2., self.diam / 2., n).tolist()
271         x_out = []
272         ys = np.linspace(-self.diam / 2., self.diam / 2., n).tolist()
273         y_out = []
274         z_out = []
275         for x in xs:
276             for y in ys:
277                 x_out.append(x)
278                 y_out.append(y)
279                 z_out.append(self.equation(x, y))
280         x_out = np.array(x_out)
281         y_out = np.array(y_out)
282         z_out = np.array(z_out)
283         x_out = x_out[(z_out <= self.max_z) & (z_out >= self.min_z)]
284         y_out = y_out[(z_out <= self.max_z) & (z_out >= self.min_z)]
285         z_out = z_out[(z_out <= self.max_z) & (z_out >= self.min_z)]
286         X = np.vstack([x_out, y_out, z_out])
287         X = np.dot(self.DCM_SL.transpose(), X)
288         X = X + self.L_r_L
289         return X
290
291     def surface_mesh(self):
292         out_rad = self.diam / 2.
```

```
293         rad_points = np.linspace(0., out_rad, 2)
294         theta_points = np.linspace(0., np.pi * 2., 30)
295         R, T = np.meshgrid(rad_points, theta_points)
296         X, Y = R * np.cos(T), R * np.sin(T)
297         Z = self.equation(X, Y)
298         for i in range(np.shape(Z)[0]):
299             for j in range(np.shape(Z)[1]):
300                 vec = np.array([X[i,j], Y[i,j], Z[i,j]])
301                 vec = np.dot(self.DCM_SL.transpose(), vec)
302                 vec = vec + self.L_r_L.reshape(3,)
303                 X[i,j], Y[i,j], Z[i, j] = vec[0], vec[1], vec[2]
304         return X, Y, Z
305 
306     def equation(self, xs, ys):
307         # defining equation for a circular hyperboloid
308         return np.sqrt(((xs**2 + ys**2)/self.b ** 2 + 1) * self.a**2)
309 
310     def intersect_rays(self, L_X_0, L_X_d):
311         # takes in ray starts and direction unit vectors in lab frame
312         num = np.shape(L_X_0)[1]
313         S_X_0 = np.dot(self.DCM_SL, L_X_0 - self.L_r_L)
314         S_X_d = np.dot(self.DCM_SL, L_X_d)
315         x0s = S_X_0[0, :]
316         xds = S_X_d[0, :]
317         y0s = S_X_0[1, :]
318         yds = S_X_d[1, :]
319         z0s = S_X_0[2, :]
320         zds = S_X_d[2, :]
321         A = (xds**2 + yds**2) / self.b**2 - zds**2 / self.a**2
322         B = 2 * (x0s*xds + yds*y0s) / self.b**2 - 2 * z0s * zds /
    self.a**2
323         C = (x0s**2 + y0s**2) / self.b**2 - z0s**2 / self.a**2 + 1.
324         non_nan = ~np.isnan(x0s)
325         ts = mullers_quadratic_equation(A[non_nan], B[non_nan], C[
    non_nan], 1)
326         S_X_1 = S_X_0
327         S_X_1[:, non_nan] = S_X_0[:, non_nan] + ts * S_X_d[:, non_nan
    ]
328         L_X_1 = np.dot(self.DCM_SL.transpose(), S_X_1) + self.L_r_L
329         return L_X_1
330 
331     def normal(self, L_X):
332         # given points on the hyperbolloid, gives unit normal vectors
    .
333         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
334         xs = S_X[0, :]
335         ys = S_X[1, :]
336         zs = S_X[2, :]
337         num = np.shape(ys)
338         S_N = np.array([2 * xs / self.b ** 2, 2 * ys / self.b ** 2, -
    2 * zs / self.a**2])
339         S_N_hat = S_N / np.linalg.norm(S_N, axis=0)
340         L_N_hat = np.dot(self.DCM_SL.transpose(), S_N_hat)
341         return L_N_hat
342 
343     def reflect_rays(self, L_X, L_d_i):
344         # takes an intersect point L_X and incoming direction L_d_i
345         # produces an outgoing direction L_d_o
346         num = np.shape(L_X)[1]
347         L_N_hat = self.normal(L_X)
348         L_d_o = []
```

```
349         for i in range(num):
350             incoming = L_d_i[:,i]
351             nHat = L_N_hat[:,i]
352             M = np.eye(3) - 2 * np.outer(nHat, nHat)
353             L_d_o.append(np.dot(M, incoming))
354         return np.array(L_d_o).transpose()  # just put it back into
    3xN array format
355 
356     def miss_rays(self, L_X):
357         # L_X is an intersection point for a hyperboloid that is
    infinite with no holes
358         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
359         zs = S_X[2, :]
360         zs[np.isnan(zs)] = 1E10
361         temp = S_X.transpose()
362         temp[zs > self.max_z] = np.array([np.nan] * 3)
363         S_X = temp.transpose()
364         L_X = np.dot(self.DCM_SL.transpose(), S_X) + self.L_r_L
365         return L_X
366 
367 class FlatImagePlane:
368     def __init__(self, w, h, L_r_L, e212):
369         self.w = w
370         self.h = h
371         self.L_r_L = L_r_L.reshape([3, 1])
372         self.DCM_SL = rbk.euler2122C(e212)
373         self.name = "image_plane"
374 
375     def intersect_rays(self, L_X_0, L_X_d):
376         # takes in ray starts and direction unit vectors in lab frame
377         num = np.shape(L_X_0)[1]
378         S_X_0 = np.dot(self.DCM_SL, L_X_0 - self.L_r_L)
379         S_X_d = np.dot(self.DCM_SL, L_X_d)
380         x0s = S_X_0[0, :]
381         xds = S_X_d[0, :]
382         y0s = S_X_0[1, :]
383         yds = S_X_d[1, :]
384         z0s = S_X_0[2, :]
385         zds = S_X_d[2, :]
386         ts = -z0s / zds
387         xs = x0s + ts * xds
388         ys = y0s + ts * yds
389         S_X_1 = S_X_0 + ts * S_X_d
390         temp = S_X_1.transpose()
391         xs[np.isnan(xs)] = 10000.
392         ys[np.isnan(ys)] = 10000.
393         temp[(np.fabs(xs) > self.w / 2.) | (np.fabs(ys) > self.h / 2.
    )] = np.array([np.nan] * 3)
394         S_X_1 = temp.transpose()
395         L_X_1 = np.dot(self.DCM_SL.transpose(), S_X_1) + self.L_r_L
396         return L_X_1
397 
398     def miss_rays(self, L_X):
399         return L_X
400 
401     def reflect_rays(self, L_X, L_d):
402         return L_d
403 
404     def extract_image(self, L_X):
405         # takes in the points that intersected the plane
406         image = np.dot(self.DCM_SL, L_X - self.L_r_L)[0:2, :]
```

```
407         return image
408 
409     def surface_points(self, num=15):
410         xs = np.linspace(-self.w/2, self.w/2)
411         ys = np.linspace(-self.h/2, self.h/2)
412         L_pt_list = []
413         for x in xs:
414             for y in ys:
415                 S_pt = np.array([x, y, 0])
416                 L_pt = (np.dot(self.DCM_SL.transpose(), S_pt) + self.
    L_r_L.reshape([3, ])).transpose()
417                 L_pt_list.append(L_pt)
418         return np.array(L_pt_list).transpose()
419 
420     def surface_mesh(self):
421         X = np.linspace(-self.w / 2., self.w / 2.)
422         Y = np.linspace(-self.h / 2., self.h / 2.)
423         X, Y = np.meshgrid(X, Y)
424         Z = np.zeros(np.shape(X))
425         for i in range(np.shape(Z)[0]):
426             for j in range(np.shape(Z)[1]):
427                 vec = np.array([X[i,j], Y[i,j], Z[i,j]])
428                 vec = np.dot(self.DCM_SL.transpose(), vec)
429                 vec = vec + self.L_r_L.reshape(3,)
430                 X[i,j], Y[i,j], Z[i, j] = vec[0], vec[1], vec[2]
431         return X, Y, Z
432 
433     def plot_image(self, ax, L_x):
434         pts = self.extract_image(L_x)
435         ax.scatter(pts[0, :], pts[1, :], color='black', s=1)
436         max_x = np.max(pts[0, :][~np.isnan(pts[0, :])])
437         min_x = np.min(pts[0, :][~np.isnan(pts[0, :])])
438         max_y = np.max(pts[1, :][~np.isnan(pts[1, :])])
439         min_y = np.min(pts[1, :][~np.isnan(pts[1, :])])
440         ax.set_xlim([min_x, max_x])
441         ax.set_ylim([min_y, max_y])
442         return
443 
444 class SphericalDetector:
445     # a spherical detector with square edges. it's a square with
    spherical curvature.
446     def __init__(self):
447         self.r = 1.
448         self.w = 1.  # width/height of detector
449         self.L_r_L = np.zeros(3)
450         self.DCM_SL = np.eye(3)
451         self.name = "image_plane"
452 
453     def equation(self, xs, ys):
454         # defining equation for a sphere
455         return np.sqrt(self.r**2 - xs**2 - ys**2)
456 
457     def intersect_rays(self, L_X_0, L_X_d):
458         # takes in ray starts and direction unit vectors in lab frame
459         S_X_0 = np.dot(self.DCM_SL, L_X_0 - self.L_r_L)
460         S_X_d = np.dot(self.DCM_SL, L_X_d)
461         x0s = S_X_0[0, :]
462         xds = S_X_d[0, :]
463         y0s = S_X_0[1, :]
464         yds = S_X_d[1, :]
465         z0s = S_X_0[2, :]
```

```
466         zds = S_X_d[2, :]
467         A = xds**2 + yds**2 + zds**2
468         B = 2 * (x0s*xds + yds*y0s + zds*z0s)
469         C = x0s**2 + y0s**2 + z0s**2 - self.r**2
470         non_nan = ~np.isnan(x0s)
471         ts = mullers_quadratic_equation(A[non_nan], B[non_nan], C[
    non_nan], -1)
472         S_X_1 = S_X_0
473         S_X_1[:, non_nan] = S_X_0[:, non_nan] + ts * S_X_d[:, non_nan
    ]
474         L_X_1 = np.dot(self.DCM_SL.transpose(), S_X_1) + self.L_r_L
475         return L_X_1
476 
477     def miss_rays(self, L_X):
478         return L_X
479 
480     def reflect_rays(self, L_X, L_d):
481         return L_d
482 
483     def extract_image(self, L_X):
484         # takes in the points that intersected the detector
485         # spits out angular coordinate on detector
486         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
487         xs = S_X[0, :]
488         ys = S_X[1, :]
489         zs = S_X[2, :]
490         ras = np.arctan(ys / (zs + self.r))
491         decs = np.arctan(xs / (zs + self.r))
492         ra_dec = np.vstack([ras, decs])
493         return ra_dec
494 
495     def surface_mesh(self):
496         X = np.linspace(-self.w / 2., self.w / 2.)
497         Y = np.linspace(-self.w / 2., self.w / 2.)
498         X, Y = np.meshgrid(X, Y)
499         Z = self.equation(X, Y)
500         for i in range(np.shape(Z)[0]):
501             for j in range(np.shape(Z)[1]):
502                 vec = np.array([X[i,j], Y[i,j], Z[i,j]])
503                 vec = np.dot(self.DCM_SL.transpose(), vec)
504                 vec = vec + self.L_r_L.reshape(3,)
505                 X[i,j], Y[i,j], Z[i, j] = vec[0], vec[1], vec[2]
506         return X, Y, Z
507 
508 def prep_rays_for_plot(ray_history):
509     ray_list = []
510     num_rays = np.shape(ray_history[-1])[1]
511     ray_array = np.array(ray_history)
512     for i in range(num_rays):
513         xs = ray_array[:, 0, i]
514         ys = ray_array[:, 1, i]
515         zs = ray_array[:, 2, i]
516         ray_list.append([xs, ys, zs])
517     return ray_list
518 
519 class AngledCircleRayDef:
520     def __init__(self):
521         self.rad = 0.5
522         self.angles = [0.]
523         self.num_circ = 15
524         self.per_circ = 150
```

```
525
526 def make_angled_circle_rays(inputs):
527     rad_inc = inputs.rad / inputs.num_circ
528     theta_inc = np.pi * 2 / inputs.per_circ
529     rays_list = []
530     rays_d_list = []
531     for angle in inputs.angles:
532         rays = []
533         angle = angle / 3600. * np.pi / 180.
534         for i in range(inputs.num_circ):
535             for j in range(inputs.per_circ):
536                 r = rad_inc * i
537                 x, y, z = 0., r * np.cos(theta_inc * j), r * np.sin(
    theta_inc * j)
538                 rays.append(np.array([x, y, z]))
539         rays = np.array(rays).transpose()
540         ray_dirs = np.array([np.array([1, 0, 0])] * np.shape(rays)[1]
    ).transpose()
541         DCM = rbk.euler1232C([0., 0., angle]).transpose()
542         ray_dirs = np.dot(DCM, ray_dirs)
543         rays_list.append(rays)
544         rays_d_list.append(ray_dirs)
545     return rays_list, rays_d_list
546
547 def make_one_edge_ray(rad, angle):
548     x, y, z = 0., rad, 0.,
549     L_X = np.array([x,y,z]).reshape([3, 1])
550     angle = angle/3600. * np.pi/180.
551     dir = np.array([np.cos(angle), -np.sin(angle), 0]).reshape([3, 1]
    )
552     return L_X, dir
553
554 def cassegrain_set_up(inputs):
555     f_1 = inputs.f_num_1 * inputs.d_1
556     M = inputs.f_num_total / inputs.f_num_1
557     alpha = inputs.e / f_1
558     beta = (M - alpha) / (M + 1.)
559     c = 1. - beta
560     d = -beta * f_1  # separation of primary and secondary
561     d_2 = c * inputs.d_1
562     f_2 = (f_1 + inputs.e) / 2.
563     secondary_a = f_2 - (f_1 + d)
564     secondary_b = np.sqrt(f_2 ** 2 - secondary_a ** 2)
565     primary_position = np.array([inputs.primary_x, 0, 0])
566     primary_a = 1. / (4. * f_1)
567     primary_hole_diam = 0.1
568     primary = parabolicMirrorWithHole(primary_a, inputs.d_1,
    primary_hole_diam, inputs.orientation_212, primary_position)
569     secondary_position = np.array([inputs.primary_x + d + secondary_a
    , 0., 0])
570     secondary = ConvexHyperbolicMirror(secondary_b, secondary_a, d_2,
     secondary_position, inputs.orientation_212)
571     dead_spot_position = secondary_position - np.array([secondary.
    max_z, 0., 0.])
572     dead_spot = circleOfDeath(d_2 / 2., dead_spot_position, inputs.
    orientation_212)
573     image_plane_position = primary_position + np.array([inputs.e +
    inputs.focal_plane_offset, 0., 0.])
574     image_plane = FlatImagePlane(inputs.d_1, inputs.d_1,
    image_plane_position, inputs.orientation_212)
575     return [dead_spot, primary, secondary, image_plane]
```

```
576
577 def cassegrain_set_up_spherical_detector(f_num_1, d_1, f_num_tot, e,
    primary_x, image_plane_offset, image_plane_r,
578                                           image_plane_max_r,
    orientation):
579     f_num_1 = 3.
580     d_1 = 1.
581     f_1 = f_num_1 * d_1
582     f_num_tot = 15.
583     M = f_num_tot / f_num_1
584     e = 0.15
585     alpha = e / f_1
586     beta = (M - alpha) / (M + 1.)
587     c = 1. - beta
588     d = -beta * f_1  # separation of primary and secondary
589     d_2 = c * d_1
590     f_2 = (f_1 + e) / 2.
591     secondary_a = f_2 - (f_1 + d)
592     secondary_b = np.sqrt(f_2 ** 2 - secondary_a ** 2)
593     primary_position = np.array([primary_x, 0, 0])
594     primary_a = 1. / (4. * f_1)
595     primary_hole_diam = 0.1
596     primary = parabolicMirrorWithHole(primary_a, d_1,
    primary_hole_diam, orientation, primary_position)
597     secondary_position = np.array([primary_x + d + secondary_a, 0., 0
    ])
598     secondary = ConvexHyperbolicMirror(secondary_b, secondary_a, d_2,
     secondary_position, orientation)
599     dead_spot_position = secondary_position - np.array([secondary.
    max_z, 0., 0.])
600     dead_spot = circleOfDeath(d_2 / 2., dead_spot_position,
    orientation)
601     image_plane_position = primary_position + np.array([e+
    image_plane_offset, 0., 0.])
602     image_plane = SphericalImagePlane(image_plane_r,
    image_plane_max_r, image_plane_position, orientation)
603     return [dead_spot, primary, secondary, image_plane]
604
605 def scatter_surfaces(ax, surfaces):
606     for i, surf in zip(range(len(surfaces)), surfaces):
607         pts = surf.surface_points()
608         ax.scatter(pts[0, :], pts[1, :], pts[2, :], label=surf.name,
    s=1)
609     return
610
611 def plot_surfaces(ax, surfaces):
612     for i, surf in zip(range(len(surfaces)), surfaces):
613         X, Y, Z = surf.surface_mesh()
614         ax.plot_surface(X, Y, Z, alpha=0.4)
615     return
616
617 def plot_rays(ax, ray_hist, al=1.0):
618     ray_hist = np.array(ray_hist)
619     for r in range(np.shape(ray_hist)[2]):
620             ray = ray_hist[:, :, r]
621             ax.plot(ray[:, 0], ray[:, 1], ray[:, 2], color='black',
    alpha=al)
622     return
623
624 def save_3d_plot(surfaces, ray_hist, path='/Users/sqc0815/
    Not_Backed_Up/hwRepo/ASTR5760', alpha=.015):
```

```
625     fig = plt.figure()
626     plt.clf()
627     ax = fig.add_subplot(111, projection='3d')
628     plot_surfaces(ax, surfaces)
629     plot_rays(ax, ray_hist, alpha)
630     plt.title('3-D View of Ray Trace')
631     ax.set_ylabel('y [m]')
632     ax.set_xlabel('x [m]')
633     ax.set_zlabel('z [m]')
634     plt.legend()
635     # ax.set_aspect('equal')
636     #plt.savefig(path + "/figures/3Dview.png")
637     return fig
638 
639 def plot_ray_starts(xs, ys, angle, path='/Users/sqc0815/Not_Backed_Up
    /hwRepo/ASTR5760'):
640     fig = plt.figure()
641     plt.clf()
642     ax = fig.add_subplot(111)
643     ax.scatter(-ys, -xs, s=2)
644     ax.set_ylim([-0.5, 0.5])
645     ax.set_xlim([-0.5, 0.5])
646     ax.set_title('ray starts (inverted)')
647     ax.set_aspect('equal')
648     ax.set_xlabel('x [m]')
649     ax.set_ylabel('y [m]')
650     plt.savefig(path + '/figures/offAxisRayStart' + str(angle) + '.
    png')
651     return fig
652 
653 def plot_flat_image(image_plane, L_X, angle, path='/Users/sqc0815/
    Not_Backed_Up/hwRepo/ASTR5760'):
654     img = plt.figure()
655     plt.clf()
656     ax = img.add_subplot(111)
657     pts = image_plane.extract_image(L_X)
658     ax.scatter(pts[0, :], pts[1, :], label='image', color='black', s=
    1)
659     max_x = np.max(pts[0, :][~np.isnan(pts[0, :])])
660     min_x = np.min(pts[0, :][~np.isnan(pts[0, :])])
661     max_y = np.max(pts[1, :][~np.isnan(pts[1, :])])
662     min_y = np.min(pts[1, :][~np.isnan(pts[1, :])])
663     ax.set_xlim([min_x, max_x])
664     ax.set_ylim([min_y, max_y])
665     ax.set_xlabel('x [m]')
666     ax.set_ylabel('y [m]')
667     ax.set_title('flat plane image at ' + str(angle) + "\'\'")
668     # ax.set_aspect('equal')
669     plt.savefig(path + '/figures/image_' + str(angle) + '.png')
670     return img
671 
672 def rms_image(image_plane, L_X):
673     pts = image_plane.extract_image(L_X)
674     xs = pts[0, :]
675     ys = pts[1, :]
676     xs, ys = xs[~np.isnan(xs) & ~np.isnan(ys)], ys[~np.isnan(xs) & ~
    np.isnan(ys)]
677     av_x, av_y = np.average(xs), np.average(ys)
678     av_pt = np.array([av_x, av_y]).reshape([2, 1])
679     pts = np.vstack([xs, ys])
680     diff = pts - av_pt
```

```
681     l2norm_square = (diff*diff).sum(axis=0)
682     rms = np.sqrt(np.average(l2norm_square))
683     return rms
684 
685 def plot_flat_image_at_dist(image_plane, L_X, angle, offset, path='/
    Users/sqc0815/Not_Backed_Up/hwRepo/ASTR5760'):
686     img = plt.figure()
687     plt.clf()
688     ax = img.add_subplot(111)
689     pts = image_plane.extract_image(L_X) * 1000
690     ax.scatter(pts[0, :], pts[1, :], label='image', color='black', s=
    1)
691     max_x = np.max(pts[0, :][~np.isnan(pts[0, :])])
692     min_x = np.min(pts[0, :][~np.isnan(pts[0, :])])
693     max_y = np.max(pts[1, :][~np.isnan(pts[1, :])])
694     min_y = np.min(pts[1, :][~np.isnan(pts[1, :])])
695     ax.set_xlim([min_x, max_x])
696     ax.set_ylim([min_y, max_y])
697     ax.set_xlabel('x [mm]')
698     ax.set_ylabel('y [mm]')
699     ax.set_title('flat plane image at ' + str(angle) + "\'\'_" + str(
    offset))
700     plt.savefig(path + '/figures/image_' + str(angle) + '_' + str(
    offset) + '.png')
701     return img
702 
703 class CassegrainDefinition:
704     def __init__(self):
705         self.f_num_1 = 3.
706         self.d_1 = 1.
707         self.f_num_total = 15.
708         self.e = 0.15
709         self.primary_x = 3.
710         self.orientation_212 = [-np.pi / 2., 0, 0]
711         self.focal_plane_offset = 0.
712         return
713 
714 class Instrument:
715     def __init__(self):
716         self.surfaces = []
717         self.detector = None
718         return
719 
720     def set_surfaces(self, surfs):
721         self.surfaces = surfs
722         self.detector = surfs[-1]
723         return
724 
725     def set_detector(self, detector):
726         self.detector = detector
727         self.surfaces[-1] = self.detector
728 
729 class RayTraceResults:
730     def __init__(self):
731         self.image = None
732         self.rms = 0.0
733         self.mean_spread = 0.0
734         return
735 
736     def find_image_rms(self):
737         xs, ys = self.image[0], self.image[1]
```

```
738         N = len(xs)
739         mean_x, mean_y = np.nanmean(xs), np.nanmean(ys)
740         self.rms = np.sqrt(np.nansum((xs - mean_x)**2 + (ys - mean_y)
    **2) / N)
741         return
742 
743     def find_image_spread(self):
744         xs, ys = self.image[0], self.image[1]
745         spread_x, spread_y = (np.nanmax(xs) - np.nanmin(xs)), (np.
    nanmax(ys) - np.nanmin(ys))
746         self.mean_spread = np.nanmean([spread_x, spread_y])
747         return
748 
749 class Experiment:
750     def __init__(self):
751         self.instrument = None # the instrument to send rays through
    including detector
752         self.L_ray_pts = None  # current location of rays in lab
    frame, L in [m]
753         self.L_ray_dir = None  # direction of rays in lab frame, L
754         self.ray_hist = []
755         self.name = "experiment"
756         self.L_ray_starts = None
757         self.L_ray_start_dirs = None
758         return
759 
760     def add_instrument(self, inst_in):
761         self.instrument = inst_in
762         return
763 
764     def set_ray_starts(self, ray_starts):
765         self.L_ray_pts = ray_starts
766         self.ray_hist.append(ray_starts)  # will I have to np.copy
    here?
767         self.L_ray_starts = ray_starts  # can reset to this
768         return
769 
770     def set_ray_start_dir(self, ray_dirs):
771         self.L_ray_dir = ray_dirs
772         self.L_ray_start_dirs = ray_dirs  # can reset to this
773         return
774 
775     def reset(self):
776         self.L_ray_pts = self.L_ray_starts
777         self.ray_hist = [self.L_ray_pts]
778         self.L_ray_dir = self.L_ray_start_dirs
779         return
780 
781     def trace_rays(self):
782         for surf in self.instrument.surfaces:
783             self.L_ray_pts = surf.intersect_rays(self.L_ray_pts, self
    .L_ray_dir)
784             self.L_ray_pts = surf.miss_rays(self.L_ray_pts)
785             self.L_ray_dir = surf.reflect_rays(self.L_ray_pts, self.
    L_ray_dir)
786             self.ray_hist.append(self.L_ray_pts)  # will I have to np
    .copy here?
787         return
788 
789     def run(self):
790         self.trace_rays()
```

```
791         result = RayTraceResults()
792         result.image = self.instrument.detector.extract_image(self.
    L_ray_pts)
793         result.find_image_rms()
794         result.find_image_spread()
795         return result
796 
797     def result_plot(self):
798         fig = plt.figure()
799         ax = fig.add_subplot('111')
800         ax.set_title('Results from ' + self.name)
801         self.instrument.detector.plot_image(ax, self.L_ray_pts)
802         ax.set_xlabel('x [m]')
803         ax.set_ylabel('y [m]')
804         return fig
805 
806     def spherical_result_plot(self):
807         fig = plt.figure()
808         ax = fig.add_subplot('111')
809         ax.set_title('Results from ' + self.name)
810         self.instrument.detector.plot_image(ax, self.L_ray_pts)
811         ax.set_xlabel('RA [rad]')
812         ax.set_ylabel('DEC [rad]')
813         return fig
814 
815 
816 class Ray:
817     def __init__(self):
818         self.X = None  # 3 x N position vectors of rays
819         self.d = None  # direction vectors of rays in same frame
820         return
821 
822     def set_pos(self, ray_starts):
823         self.X = ray_starts
824         return
825 
826     def set_dir(self, ray_dirs):
827         self.d = ray_dirs
828         return
829 
830 class RowlandCircle:
831     # a rowland circle diffraction grating
832     # circular curvature, but square cut edges
833     # the surface normal always points in +z direction in local (S)
    frame at the origin
834     # i.e., the surface is grazing tangent to the origin (the sphere
    is not centered on the S origin
835     # the S frame y-axis is parallel to the gratings at the origin
836     # the x-axis is y cross z, then
837     # The sphere is centered at 0, 0, 0
838     def __init__(self):
839         self.r = 1.  # radius of Rowland Circle
840         self.w = 0.25  # it's a square chunk of a circle, this is the
     side length.
841         self.m = 1  # diffraction order. For now, just choose one
    order to look into and do it multiple times to get
842                     # multiple orders
843         self.lam = 1000. / 1E10  # wavelength to consider. Again,
    this is just a hack for now
844         self.line_density = 1000.  # grating line density
845         self.DCM_SL = np.eye(3)  # rotation into the surface frame
```

```
845 from lab frame
846         self.L_r_L = np.zeros(3).reshape([3, 1])  # offset from lab
    origin in lab frame coordinates
847         self.grating_direction_q = np.array([0., -1., 0.])  #
    parallel to gratings
848         self.unprojected_spacing = 1.
849         self.central_normal = np.array([0., 0., 1.]).reshape([3, 1])
850         return
851 
852     def set_radius(self, radius):
853         self.r = radius
854         return
855 
856     def set_line_density(self, density):
857         # lines per mm
858         self.line_density = density  # per mm
859         self.unprojected_spacing = 1. / (1000. * self.line_density)
    # [m / space]
860         return
861 
862     def set_position(self, pos):
863         self.L_r_L = pos
864         return
865 
866     def set_DCM(self, dcm):
867         self.DCM_SL = dcm
868         return
869 
870     def set_width(self, width):
871         self.w = width
872         return
873 
874     def set_order(self, order):
875         self.m = order
876         return
877 
878     def set_wavelength(self, wavelength):
879         self.lam = wavelength / 1E10  # convert angstrom to m
880         return
881 
882     def surface_mesh(self):
883         X = np.linspace(-self.w / 2., self.w / 2.)
884         Y = np.linspace(-self.w / 2., self.w / 2.)
885         X, Y = np.meshgrid(X, Y)
886         Z = self.equation(X, Y)
887         for i in range(np.shape(Z)[0]):
888             for j in range(np.shape(Z)[1]):
889                 vec = np.array([X[i,j], Y[i,j], Z[i,j]])
890                 vec = np.dot(self.DCM_SL.transpose(), vec)
891                 vec = vec + self.L_r_L.reshape(3,)
892                 X[i, j], Y[i, j], Z[i, j] = vec[0], vec[1], vec[2]
893         return X, Y, Z
894 
895     def equation(self, xs, ys):
896         # defining equation for a sphere
897         num = np.shape(xs)[0]
898         return solve_quadratic(np.ones(num), -2. * self.r * np.ones(
    num), xs**2 + ys**2, -1)
899 
900     def intersect_rays(self, L_X_0, L_X_d):
901         # takes in ray starts and direction unit vectors in lab frame
```

```
902         S_X_0 = np.dot(self.DCM_SL, L_X_0 - self.L_r_L)
903         S_X_d = np.dot(self.DCM_SL, L_X_d)
904         x0s = S_X_0[0, :]
905         xds = S_X_d[0, :]
906         y0s = S_X_0[1, :]
907         yds = S_X_d[1, :]
908         z0s = S_X_0[2, :]
909         zds = S_X_d[2, :]
910         A = xds**2 + yds**2 + zds**2
911         B = 2. * (x0s*xds + yds*y0s + zds*z0s - self.r*zds)
912         C = x0s**2 + y0s**2 + z0s**2 - 2 * self.r * z0s
913         non_nan = ~np.isnan(x0s)
914         ts = solve_quadratic(A[non_nan], B[non_nan], C[non_nan], 1)
915         S_X_1 = S_X_0
916         S_X_1[:, non_nan] = S_X_0[:, non_nan] + ts * S_X_d[:, non_nan
    ]
917         L_X_1 = np.dot(self.DCM_SL.transpose(), S_X_1) + self.L_r_L
918         return L_X_1
919
920     def normal(self, L_X):
921         # given points on the sphere, gives unit normal vectors.
922         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
923         xs = S_X[0, :]
924         ys = S_X[1, :]
925         zs = S_X[2, :]
926         S_N = -np.array([2. * xs, 2. * ys, 2. * (zs - self.r)])
927         S_N_hat = S_N / np.linalg.norm(S_N, axis=0)
928         L_N_hat = np.dot(self.DCM_SL.transpose(), S_N_hat)
929         return L_N_hat
930
931     def miss_rays(self, L_X):
932         # based on width of the element. misses if outside x,y bounds
     based on width of grating
933         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
934         xs = np.fabs(S_X[0, :])
935         xs[np.isnan(xs)] = -1.
936         ys = np.fabs(S_X[1, :])
937         ys[np.isnan(ys)] = -1.
938         temp = S_X.transpose()
939         temp[(xs > self.w / 2.) | (ys > self.w / 2.)] = np.array([np.
    nan] * 3)
940         S_X = temp.transpose()
941         L_X = np.dot(self.DCM_SL.transpose(), S_X) + self.L_r_L
942         return L_X
943
944     def reflect_rays(self, L_X, L_d):
945         # following spencer and murty, 1961
946         S_d = np.dot(self.DCM_SL, L_d)
947         r = np.dot(self.DCM_SL, self.normal(L_X))  # normal vectors
    at every intersection point in S frame
948         Ks = r[0, :]
949         Ls = r[1, :]
950         Ms = r[2, :]
951         u = 1. / np.sqrt(1. + Ks**2. / (Ls**2. + Ms**2.))
952         v = -Ks * Ls * u / (Ls**2 + Ms**2)
953         w = -Ks * Ms * u / (Ls**2 + Ms**2)
954         p = np.vstack([u, v, w])
955         d = self.unprojected_spacing / u
956         L = self.m * self.lam / d  # assumes wavelength is defined in
     current medium or probably that we're always in vacuum
957         mu = 1.  # no change in medium
```

```
958         kulvmw = np.array([np.dot(p[:, i], S_d[:, i]) for i in range
    (np.shape(p)[1])])
959         b_prime = (mu**2. - 1. + L**2. - 2. * mu * L * kulvmw)  #
    notice I don't divide by r**2 because I norm my normals
960         a = mu * np.array([np.dot(S_d[:, i], r[:, i]) for i in range
    (np.shape(S_d)[1])])
961         nans = np.isnan(b_prime)
962         doable = np.ones(len(b_prime), dtype=bool)
963         doable[~nans] = b_prime[~nans] <= a[~nans]**2
964         doable[nans] = False
965         G = (solve_quadratic(np.ones(np.sum(doable)), 2. * a[doable]
    , b_prime[doable], 1))
966         S_d[:, doable] = S_d[:, doable] - L[doable] * p[:, doable] +
     G.flatten() * r[:, doable]
967         S_d[:, ~doable] = np.ones([3, np.sum(~doable)]) * np.nan
968         L_d = np.dot(self.DCM_SL.transpose(), S_d)
969         return L_d
970
971 class CylindricalDetector:
972     # a cylindrical (circular extrusion) detector.
973     # The long axis is the x-axis
974     # an axial ray would come in the z^ axis, going in the negative
    x^ direction. a reflected ray would go off in the z^
975     # y is radial. orthogonal to x. orthogonal to z (which is also
    radial).
976     # The cylinder is lifted on the z-axis so that the "vertex" of
    the surface is at the frame origin
977     def __init__(self):
978         self.r = 1.
979         self.h = 1.  # width/height of detector
980         self.sweep = np.pi / 8.
981         self.y_min = -100.
982         self.y_max = 100.
983         self.set_y_limits()
984         self.L_r_L = np.zeros(3)
985         self.DCM_SL = np.eye(3)
986         self.name = "image_plane"
987
988     def set_y_limits(self):
989         self.y_max = self.r * np.sin(self.sweep / 2.)
990         self.y_min = -self.y_max
991         return
992
993     def equation(self, xs, ys):
994         # defining equation for a cylinder
995         # return the z-value of the surface
996         num = np.shape(xs)[0]
997         return solve_quadratic(np.ones(num), -2 * self.r, ys ** 2, -
    1)
998
999     def set_radius(self, r):
1000         self.r = r
1001         return
1002
1003     def set_height(self, h):
1004         self.h = h
1005         return
1006
1007     def set_sweep_angle(self, ang):
1008         self.sweep = ang
1009         return
```

```
1010
1011     def set_position(self, pos):
1012         self.L_r_L = pos
1013         return
1014
1015     def set_DCM(self, dcm):
1016         self.DCM_SL = dcm
1017         return
1018
1019     def normal(self, L_X):
1020         # given points on the cylinder, gives unit normal vectors.
1021         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
1022         ys = S_X[1, :]
1023         zs = S_X[2, :]
1024         num = np.shape(zs)[0]
1025         S_N = -np.array([np.zeros(num), 2 * ys, 2 * (zs - self.r)])
1026         S_N_hat = S_N / np.linalg.norm(S_N, axis=0)
1027         L_N_hat = np.dot(self.DCM_SL.transpose(), S_N_hat)
1028         return L_N_hat
1029
1030     def intersect_rays(self, L_X_0, L_X_d):
1031         # takes in ray starts and direction unit vectors in lab
     frame
1032         S_X_0 = np.dot(self.DCM_SL, L_X_0 - self.L_r_L)
1033         S_X_d = np.dot(self.DCM_SL, L_X_d)
1034         y0s = S_X_0[1, :]
1035         yds = S_X_d[1, :]
1036         z0s = S_X_0[2, :]
1037         zds = S_X_d[2, :]
1038         A = zds**2 + yds**2
1039         B = 2 * z0s * zds - 2 * self.r * zds + 2 * y0s * yds
1040         C = z0s**2 - 2 * self.r * z0s + y0s**2
1041         non_nan = ~np.isnan(y0s)
1042         ts = solve_quadratic(A[non_nan], B[non_nan], C[non_nan], 1)
1043         S_X_1 = S_X_0
1044         S_X_1[:, non_nan] = S_X_0[:, non_nan] + ts * S_X_d[:,
     non_nan]
1045         L_X_1 = np.dot(self.DCM_SL.transpose(), S_X_1) + self.L_r_L
1046         return L_X_1
1047
1048     def miss_rays(self, L_X):
1049         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
1050         xs = S_X[0, :]
1051         ys = S_X[1, :]
1052         xs[np.isnan(xs)] = -1E6
1053         ys[np.isnan(ys)] = -1E6
1054         temp = S_X.transpose()
1055         temp[(ys > self.y_max) | (ys < self.y_min) | (xs < -self.h /
      2.) | (xs > self.h / 2.)] = np.array([np.nan] * 3)
1056         S_X = temp.transpose()
1057         L_X = np.dot(self.DCM_SL.transpose(), S_X) + self.L_r_L
1058         return L_X
1059
1060     def reflect_rays(self, L_X, L_d):
1061         return L_d
1062
1063     def extract_image(self, L_X):
1064         # should change this into a linear and angular  coordinate.
     Right now it does a spherical RA/DEC transformation
1065         S_X = np.dot(self.DCM_SL, L_X - self.L_r_L)
1066         xs = S_X[0, :]
```

```
1067          ys = S_X[1, :]
1068          RA = - np.arcsin(ys / self.r)
1069          ra_h = np.vstack([RA, xs])
1070          return ra_h
1071
1072      def surface_mesh(self):
1073          X = np.linspace(-self.h / 2., self.h / 2.)
1074          angles = np.linspace(-self.sweep / 2., self.sweep / 2.)
1075          Y = np.sin(angles) * self.r
1076          X, Y = np.meshgrid(X, Y)
1077          Z = self.equation(X, Y)
1078          for i in range(np.shape(Z)[0]):
1079              for j in range(np.shape(Z)[1]):
1080                  vec = np.array([X[i,j], Y[i,j], Z[i,j]])
1081                  vec = np.dot(self.DCM_SL.transpose(), vec)
1082                  vec = vec + self.L_r_L.reshape(3,)
1083                  X[i,j], Y[i,j], Z[i, j] = vec[0], vec[1], vec[2]
1084          return X, Y, Z
1085
1086
1087
1088
1089
```