Kraken– Núcleo para trazado de rayos y simulación óptica

|  |
| --- |
|  |

Contenido

**No se encontraron entradas de tabla de contenido.**

Introducción

Krakenes una herramienta para la simulación de sistemas ópticos, consta de una biblioteca desarrollada en el lenguaje de programación Python 3.0 y las librerías Numpy, PyVTK y PyVista lo cual le permite proveer de visualización tridimensional de los elementos ópticos. Esta herramienta se ha enfocado en el paradigma de la programación orientada a objetos, esto al parecer es de forma natural la que nos brinda una mayor simplicidad en la implementación de un sistema como se verá en este documento.

A lo largo de este documento nos referiremos al código de ejemplo por el número de línea, por ejemplo, a continuación, se muestra la línea 0 del código Python de ejemplo.

1. # código Python de ejemplo en el documento

**Prerrequisitos**

La librería ha sido probada con los siguientes paquetes y versiones.

* Python '3.7.4'
* numpy '1.18.5'
* pyvista '0.25.3'
* pyvtk '0.5.18'
* vtk '8.2'
* csv '1.0'
* Colocar la biblioteca SigmaZX.py y el directorio Cat en la misma ruta donde está el código que se desea ejecutar.

Nota: los ejemplos mostrados en este documento también se encuentran en el archivo de la librería así que pueden ser ejecutados directamente para la visualización.

**Clases y atributos**

La biblioteca se ha simplificado al grado de contar únicamente con dos clases de objetos par la definición de un sistema, estos son ***surf*** y ***system***, cuya aplicación se describe a continuación.

Lo más importante antes de comenzar a utilizar la librería es importar SigmaZ\_Setup, Surf y System

1. **from** Kraken**import** SigmaZ\_Setup
2. **from** Kraken**import** surf
3. **from** Kraken**import** system

Sin embargo, guardar los resultados de un trazado de rayos, graficar el sistema en 3 o 2 dimensiones se deben de cargar las herramientas Raykeeper, rayjuggler, pupilcalc, display3D y display2D

1. **from** Kraken**import** raykeeper
2. **from** Kraken**import** rayjuggler
3. **from** Kraken**import** display3D
4. **from** Kraken**import** display2D
5. **from** Kraken**import** pupilcalc

El objeto ***surf***, contiene toda la información relevante de toda interfaz óptica, de esta forma, toda interfaz óptica es un objeto de la clase surf, todas las interfaces, desde el plano objeto hasta el plano imagen contienen atributos de tamaño, forma, material u orientación.

|  |  |
| --- | --- |
| ***Tabla 1. Parámetros de una superficie*** | |
| surf.Name="" | Nombre del elemento, útil únicamente para diagrama 2D o identificación de rayos |
| surf.NamePos=(0,0) | Posición de la nota con el Nombre “Name” en el diagrama 2D |
| surf.Note="None" | Útil para notas del usuario en dicha superficie |
| surf.Rc = 9999999999.0 | Radio de curvatura paraxial en milímetros, |
| surf.Cylinder\_Rxy\_Ratio=1 | Razón entre el radio de curvatura axial y sagital, útil para lentes cilíndricas y toroides |
| surf.Axicon=0 | Simulación de axicones, para valores diferente de cero se genera un axicón con el ángulo introducido. |
| surf.Thickness=0.0 | Separación entre esta superficie y la siguiente superficie. |
| surf.Diameter = 1.0 | Diámetro exterior de la superficie. |
| surf.InDiameter=0.0 | Diámetro interno de la superficie, útil para elementos como un espejo primario con apertura central. |
| surf.k=0.0 | Constante de conicidad para superficies cónicas clásicas, k=0 para esférica, k=-1 para parábola, etc. |
| surf.DespX=0.0 | Desplazamiento de la superficie en el eje X,Y y Z (en milímetros) |
| surf.DespY=0.0 |
| surf.DespZ=0.0 |
| surf.TiltX=0.0 | Giro de la superficie en el eje X,Y y Z (en grados). |
| surf.TiltY=0.0 |
| surf.TiltZ=0.0 |
| surf.Order=0 | Define el orden de las trasformaciones, si el valor es 0 primero se realizan las traslaciones y después las rotaciones, si el valor es 1 entonces primero se realizan las rotaciones y después as traslaciones. |
| surf.AxisMove=1 | Define lo que ocurrirá con el eje óptico después de una transformación de coordenadas, si el valor es cero la transformación únicamente se realiza a la superficie en cuestión, si el valor es 1 entonces la transformación también afecta al eje óptico por lo tanto las demás superficies seguirán a transformación, si el valor es diferente, por ejemplo 2, entonces el eje óptico será afectado al doble, esto último es útil para espejos planos, ver ejemplo. |
| surf.Diff\_Ord=0.0 | Orden de difracción, convierte al elemento en una rejilla de difracción, el valor de radio de curvatura debe de omitirse para que sea una rejilla de difracción plana, puede utilizarse en transmisión o en refracción. |
| surf.Grating\_D=0.0 | Separación (micras) entre las líneas de difracción. |
| surf.Grating\_Angle=0.0 | Ángulo de las líneas de la rejilla en el plano de la superficie es decir al rededor del eje óptico. Esto es útil para simular dispersión cónica. |
| surf.ZNK=np.zeros(36) | Arreglo de tipo Numpy de 36 elementos que corresponden a los coeficientes de los polinomios de Zernike en la nomenclatura de Noll Ref. XXXXX. |
| surf.ShiftX=0 | Desplazamiento de la función del perfil de la superficie en el eje x o y, esto es útil por ejemplo para superficies fuera de eje como parábolas. |
| surf.ShiftY=0 |
| surf.Mask=0 | 0 non masked, 1 apperture, 2 Obstruction. |
| surf.Mask\_Shape=Objeto\_3D | Ejemplo: Objeto\_3D=pv.Disc(center=[0.0,0.0,0.0], inner=0, outer=0.001, normal=(0,0,1),r\_res=3,c\_res=3). |
| surf.ExtraData=np.zeros(200) | Arreglo de datos extra de propósito multiple, ver ejemplo (TAOS). |
| ﻿Surf.Error\_map= ﻿[X,Y,Z, SPACE] | Arga un mapa de error generado con un arreglo del tipo numpy para las coordenadas en X, yY y la altura en Z con un valor de espacio entre X,Y generado de la siguiente manera.  Ejemplo:  ﻿  def ErrorGen():  L=1000.  N=20.  hight=0.001  SPACE=2\*L/N  x = np.arange(-L, L+SPACE, SPACE)  y = np.arange(-L, L+SPACE, SPACE)  gx, gy = np.meshgrid(x, y)  R=np.sqrt((gx\*gx)+(gy\*gy))  arg=np.argwhere(R<L)  Npoints=np.shape(arg)[0]  X=np.zeros(Npoints)  Y=np.zeros(Npoints)    i=0  for [a,b] in arg:  X[i]= gx[a,b]  Y[i]= gy[a,b]    i=i+1    spa = 10000000  Z = hight\*(np.random.randint(-spa,spa,Npoints))/(spa\*2.0)  return [X,Y,Z, SPACE] |
| surf.Drawing=1 | 1 para que el elemento sea dibujado en el graficado 3D, 0 para omitir el dibujo del elemento. |
| surf.Color=[0,0,0] | Definir el color del elemento en el formato [1,1,1] |
| surf.Solid\_3d\_stl="None" | Ruta del objeto solido 3D en formato STL. |

El objeto ***system*** está pensado como un contenedor para todas las interfaces, este objeto contiene implementaciones para el trazado de rayos y para la obtención de distintos parámetros del rayo, los cuales son acumulativos a través de las superficies por las que el rayo atraviesa, las implementaciones públicas de ***system*** son mostradas a continuación.

***Para comprender la manera en que estos elementos son llamados ver: Examp-Doblete-ComandosSystem.py***

|  |  |
| --- | --- |
| ***Tabla 2. Parámetros para la comunicación con system*** | |
| system.Trace(pS, dC, wV) | Trace es la implementación principal del objeto ***system***, esta realiza el trazo de un rayo a través de todas las superficies que encuentre en su camino de forma secuencial. El rayo debe de ser definido por un punto de origen “pS”, los cosenos directores “dC” la longitud de onda “wV”. Ver ejemplos  pS = [1.0, 0.0, 0.0]  dC=[0.0,0.0,1.0]  wV=0.4 |
| system.NsTrace(pS, dC, wV) | Traza rayos de la misma forma que Trace, sin embargo, lo realiza de forma no secuencial, los parámetros del rayo se definen de la misma manera. |
| system.Parax(w) | Devuelve los sigientes calculos paraxiales accesibles tambien desde system:  system.SistemMatrix  system.S\_Matrix  system.N\_Matrix  system.a  system.b  system.c  system.d  system.EFFL  system.PPA  system.PPP  nota: Ver sección **Herramienta Parax** |
| system.disable\_inner | Habilita y deshabilita las aperturas centrales, esto es muy útil, por ejemplo, cuando se desea calcular un trazo de rayos sin el viñeteo de la apertura de un espejo primario. |
| system.enable\_inner | Plano principal posterior |
| system.SURFACE | Devuelve una lista del número de superficies por las que paso el rayo.  Devuelve una lista del nombre de superficies por las que paso el rayo, si no se agregó un nombre a las superficies entonces la lista aparecerá con campos vacíos. La colocación de nombre a las superficies es muy útil, por ejemplo, para identificar si un rayo ha tocado dicha superficie. |
| system.NAME |
| system.GLASS | Devuelve una lista de los materiales por los cuales el rayo ha atravesado. |
| system.XYZ | Devuelve las coordenadas ***[X,Y,Z]*** del rayo desde su origen hasta el plano imagen |
| system.S\_XYZ | Devuelve las coordenadas ***[X,Y,Z]*** del rayo desde su origen y en todas las superficies donde este rayo es originado, es decir, las coordenadas del plano imagen están exentas. |
| system.T\_XYZ | Devuelve las coordenadas ***[X,Y,Z]*** del rayo desde la primera superficie en la que interseca hasta el plano imagen. |
| system.OST\_XYZ | Devuelve las coordenadas ***[X,Y,Z]*** en las que intercepta un rayo a la superficie en el espacio de la interfaz, es decir, las coordenadas con respecto a un sistema de coordenadas en su vértice incluso si este vértice tiene una traslación o rotación. |
| system.DISTANCE | Devuelve una lista con las distancias recorridas por el rayo entre puntos de intersección. |
| system.OP | Devuelve una lista con los caminos ópticos recorridos por el rayo entre puntos de intersección, para esto se considera la dispersión del vidrio y la distancia entre los puntos de intersección. |
| system.TOP | Devuelve el camino óptico total del rayo desde la fuente hasta el último punto de intersección. |
| system.TOP\_S | Devuelve una lista acumulativa del camino óptico del rayo desde la fuente hasta el último punto de intersección. |
| system.ALPHA | Devuelve una lista con los coeficientes de absorción para los materiales en las superficies interceptadas considerando la longitud de onda, estos valores los obtiene del catálogo de materiales. |
| system.BULK\_TRANS | Devuelve una lista con la transmisión a través de todos los recorridos dentro del sistema, para esto se consideran los caminos ópticos y los coeficientes de absorción de los materiales. |
| system.S\_LMN | Devuelve una lista con los cosenos directores [***L,M***,***N]*** de las normales para los puntos de intersección de un rayo a través de todas las interfaces por las que pasa. |
| system.LMN | Devuelve una lista con los cosenos directores [***L,M***,***N]*** de un rayo incidente a través de todas las interfaces por las que atraviesa. |
| system.R\_LMN | Devuelve una lista con los cosenos directores [***L,M***,***N]*** de rayo resultante a través de todas las interfaces por las que atraviesa. |
| system.N0 | Índices de refracción antes y después de cada interfaz por las que el rayo atraviesa. Este índice es calculado con la dispersión del material y la longitud de onda del rayo en cuestión. |
| system.N1 | Devuelve una lista con los cosenos directores [***L,M***,***N]*** de rayo resultante a través de todas las interfaces por las que atraviesa. |
| system.WAV | Longitud de onda de rayo en cuestión, aunque el comando devuelve una lista todos los valores son iguales porque la longitud de onda es constante para un rayo, el tamaño de la lista indica únicamente el número de iteraciones con interfaces del sistema.  Devuelve una lista con los términos, ***L, M*** o ***N*** de los cosenos directores que definen las líneas de la rejilla de difracción sobre el plano. |
| system.G\_LMN |
| system.ORDER | Devuelve una lista con los órdenes de difracción asociados al rayo en cuestión. |
| system.GRATING | Distancia entre líneas de la rejilla de difracción. |
| system.RP | Devuelve una lista con los coeficientes de Fresnel de reflexión y transmisión para polarización S y P |
| system.RS | Distancia entre líneas de la rejilla de difracción. |
| system.TP | Devuelve una lista con los coeficientes de Fresnel de reflexión y transmisión para polarización S y P  Energía total transmitida o reflejada por elemento.  Energía total transmitida o reflejada total. |
| system.TS |
| system.TTBE |
| system.TT |
| system.targ\_surf(int) | Limita el trazado de rayos hasta la superficie definida (int) |
| system.flat\_surf(int) | Se define con el numero entero de la superficie que requiere hacerse plana, -1 para restaurar |
| system.disable\_inner() | Si alguna superficie del sistema tiene un agujero central este se deshabilita. |
| system.enable\_inner() | Si alguna superficie () tiene un agujero central deshabilitado este comando lo rehabilita. |
|  |  |
|  |  |

**Trabajo con la biblioteca**

**Ejemplo: Doblete**

1. # Importación de las bibliotecas
2. **from** SigmaZX **import** SigmaZ\_Setup
3. **from** SigmaZX **import** Surf
4. **from** SigmaZX **import** System
6. **from** SigmaZX **import** Raykeeper
7. **from** SigmaZX **import** Display3D
8. **from** SigmaZX **import** Display2D
10. **import** numpy as np
11. **import** matplotlib.pyplot as plt

Todas las superficies son declaradas por separado, cada una con los parámetros posibles para una superficie están mostrados en la tabla 1.

1. # Declaración de todas las superficies con la clase Surf()
3. P\_Obj=Surf()
4. P\_Obj.Rc=0.0
5. P\_Obj.Thickness=10
6. P\_Obj.Glass="AIR"
7. P\_Obj.Diameter=30.0
9. L1a=Surf()
10. L1a.Rc=9.284706570002484E+001
11. L1a.Thickness=6.0
12. L1a.Glass="BK7"
13. L1a.Diameter=30.0
14. L1a.Axicon=0
16. L1b=Surf()
17. L1b.Rc=-3.071608670000159E+001
19. L1b.Thickness=3.0
20. L1b.Glass="F2"
21. L1b.Diameter=30
23. L1c=Surf()
24. L1c.Rc=-7.819730726078505E+001
25. L1c.Thickness=9.737604742910693E+001
26. L1c.Glass="AIR"
27. L1c.Diameter=30
28. L1c.NamePos=(10,0)
29. L1c.Name="L1c"
31. P\_Ima=Surf()
32. P\_Ima.Rc=0.0
33. P\_Ima.Thickness=0.0
34. P\_Ima.Glass="AIR"
35. P\_Ima.Diameter=100.0
36. P\_Ima.NamePos=(10,0)
37. P\_Ima.Name="Plano imagen"

Se carga la clase ***SigmaZ-Setup***, dentro de esta se establecerán los directorios de los catálogos de vidrios entre otras cosas, en futuras versiones con se podrán establecer diferentes configuraciones para un sistema lo cual puede tener utilidad. Por el momento cargaremos la configuración por defecto en la ***configuracion\_1***.

1. Configuración\_1=SigmaZ\_Setup()

Se realiza un arreglo ***A*** con los elementos declarados para todas las superficies, posteriormente se crea un sistema óptico con todas las superficies contenidas en ***A*** y la ***configuracion\_1***, y finalmente se crea un contenedor para los rayos con ***Raykeeper***

1. A=[P\_Obj,L1a,L1b,L1c,P\_Ima]
2. Doblete=System(A,configuracion\_1)
3. Rayos=Raykeeper(Doblete)

Todo rayo tiene 3 parámetros, unas coordenadas de origen llamadas ***XYZ*** para este ejemplo, una dirección definida por cosenos directores, definidos a continuación como LMN, estos dos parámetros son arreglos con tres valores [x,y,z] y [l, m, n]. el tercer parámetro por definir es la longitud de onda, aquí expresada como ***w*** con un valor de 0.4μm.

1. XYZ = [0.0, 10.0, 0.0]
2. LMN=[0.0,0.0,1.0]
3. W=0.4

Se realiza el trazo del rayo atreves del sistema ***Doblete*** de la siguiente forma

1. Doblete.Trace(XYZ,LMN,W)

Es posible interrogar al sistema ***Doblete*** sobre lo que ha ocurrido con el rayo, la comunicación con el sistema Doblete se realiza con las llamadas mostradas en la tabla 2, por ejemplo, la palabra clave GLASS nos devuelve todos los vidrios por los que ha pasado el rayo y se utiliza de la siguiente forma.

1. **print**(Doblete.GLASS)

Devolviendo el siguiente arreglo ['BK7', 'F2', 'AIR', 'AIR']. Otro ejemplo es la solicitud de las coordenadas del rayo en todas las superficies, además podemos pedir os cosenos directores, para eso utilizaremos el comando ***XYZ*** y ***LMN*** de la siguiente forma:

1. **print**(Doblete.XYZ)
2. **print**(Doblete.LMN)

para lo cual obtenemos las siguientes salidas respectivamente:

|  |
| --- |
| Doblete.XYZ=[array([ 0., 10., 0.]), [0.0, 10.0, 10.54009083298281], [0.0, 9.85609739239213, 14.37575625216807], [0.0, 9.81741071793073, 18.381280697704405], [0.0, 0.05825657548273888, 116.37604742910693]] |
| Doblete.LMN=[array([0., 0., 1.]), array([ 0. , -0.03749061, 0.99929698]), array([ 0. , -0.00965788, 0.99995336]), array([ 0. , -0.09909831, 0.99507765])] |

Ambos resultados son arreglos del tipo Numpy, por lo tanto, podemos realizar las operaciones que deseemos con ellos directamente, por ejemplo, a continuación vemos la forma de los arreglos

1. **print**(np.shape(Doblete.XYZ))
2. **print**(np.shape(Doblete.LMN))

lo que nos regresa:

np.shape(Doblete.XYZ)=(5, 3)

np.shape(Doblete.LMN)=(4, 3)

Nótese que Doblete.XYZ contiene 5 arreglos de tres elementos (coordenadas x,y,z en el espacio tridimensional), mientras que Doblete.LMN solo contiene 4 elementos, esto es porque solo muestra los cosenos directores entre las superficies, es decir, si un sistema tiene 5 elementos desde el plano objeto hasta el plano imagen, entonces solo existen 4 segmentos de rayo entre superficies y por lo tanto solo 4 juegos de cosenos directores, ejemplificados a continuación con -> entre las superficies.

A=[P\_Obj -> L1a -> L1b -> L1c -> P\_Ima]

Después de realizar el trazo de un rayo como se indicó en la línea 56 de ejemplo en código Python nosotros podemos solicitar información a Doblete de la forma mencionada anteriormente, con esta información se pueden realizar graficas o distintos análisis, generalmente en necesario realizar un trazado con muchos rayos, para conservar los resultados para uno o más rayos tenemos el contenedor de rayos en la clase ***Raykeeper***, con la cual para el ejemplo de este documento creamos el objeto Rayos en la línea 52.

Una palabra clave de la clase Raykeeper es push(), con esta tomamos el rayo recién trazado dentro de Doblete de la siguiente forma, una ventaja de esto es que el objeto Doblete no tiene que guardar una memoria con todos los rayos que Tracemos, le dejamos esa tarea al contenedor de rayos, de esta forma podemos tener contenedores de rayos para diferentes circunstancias según deseemos, por ejemplo, podemos crear un contenedor para todos los rayos que trazaremos desde un campo y luego guardar en otro contenedor los rayos que vienen de un campo distinto, el usuario puede encontrar muchas utilidades para esta modalidad.

Cada rayo individual es guardado pasando al sistema en cuestión como parámetro al contenedor de la siguiente manera, esto se repite cada que un rayo es trazado.

1. Rayos.push()

Como se puede ver en las líneas del ejemplo 5 y 6, también importamos dos herramientas, Display2D y Display3D, estas realizan la gráfica del sistema, los parámetros de entrada son el sistema en si mismo y un contenedor de rayos, además recibe un parámetro para diferentes consideraciones en la gráfica.

Para graficas bidimensionales ***Display2D(System, Raykeeper, parámetro)*** donde el parámetro puede ser los números enteros 0 o 1 con lo que se indica si la gráfica será en el plano xz o yz.

Para graficas bidimensionales ***Display3D(System, Raykeeper, parámetro***) donde el parámetro puede ser los números enteros 0 al 2.

|  |  |
| --- | --- |
| 0 | para un despliegue de los elementos completos |
| 1 | para el despliegue con las superficies con un corte de ¼ |
| 2 | para el despliegue de secciones transversales. |

Para el ejemplo en este documento tendremos que remplazar ***System*** y ***Raykeeper*** por los objetos creados con ellos de la siguiente forma

1. Display2D(Doblete,Rayos,0)

Lo que genera la siguiente grafica desplegada en Matplotlib. Recordemos que solo guardamos un rayo en el contenedor, este se muestra en color azul, esto depende de la longitud de onda utilizada, es este caso al definir el rayo utilizamos W=0.4 (línea 55 del código Python del ejemplo).

|  |
| --- |
| Gráfico  Descripción generada automáticamente |
|  |

Para realizar el despliegue tridimensional utilizaremos:

1. Display3D(Doblete,Rayos,2)

Con lo que se abrirá la siguiente ventana del visualizador

|  |
| --- |
|  |
|  |

Podemos almacenar varios rayos, por ejemplo, a continuación, se anida la creación del rayo, su trazo y su almacenado dentro de un siclo ***for***.

1. **for** x **in** range(-10,10):
2. # creamos un rayo paralelo al eje óptico y cambiamos la altura con "x" en el for
3. XYZ = [0.0, x, 0.0]
4. LMN=[0.0,0.0,1.0]
5. W=0.4
6. # Trazamos el rayo
7. Doblete.Trace(XYZ, LMN,W)
8. #Almacenamos el rayo
9. Rayos.push(Doblete)
11. # Despliega sistema con todos los rayos
12. Display3D(Doblete,Rayos,2)

Con lo cual obtenemos todos los rayos

|  |
| --- |
|  |
| Despliegue 3D del sistema óptico con varios rayos |

El contenedor de rayos contiene una lista de conserva casi todos los parámetros que son accesibles en la clase ***System***, sin embargo, ahora, al contener varios rayos estos parámetros son arreglos de arreglos por lo que hay que tener ciertas consideraciones en cuenta.

Por defecto el contenedor rayos ***Raykeeper*** toma en cuenta todos los rayos que se guardan con la instrucción ***Raykeeper.push()***, es posible que algún rayo no llegue siquiera a entrar al sistema debido a su punto de origen o su dirección, de todas formas, el rayo es tomado y la información existente como el punto de origen, los cosenos directores y la longitud de onda, el resto de los datos estarán vacíos, esto también puede ser útil por ejemplo si queremos conocer las características de los rayos que ***no*** entran al sistema. Para conocer el número de rayos almacenados ***Raykeeper*** tiene una variable interna ***nrays*** que se puede llamar directamente de esta forma.

1. **print**(Rayos.nrays)
2. Resultado: 100

Por lo tanto, tenemos en este ejemplo 100 posiciones con rayos sobre los cuales podemos solicitar información al contenedor a través de los parámetros mostrados en la tabla 3 (Columna central), estos tienen la misma información descrita en la Tabla 2 de la clase ***System***. estas llamadas al contenedor ***Raykeeper*** dependen del número de rayo definido por un valor entero ***[#]***. Si no se indica el número de rayo se obtiene el conjunto total de datos en arreglos del tipo Numpy. Como ya se ha mencionado existen rayos que no tocaron ninguna superficie, a los que si tocaron alguna superficie los llamamos rayos ***válidos*** y puede pedírsele al contenedor una lista la dirección de estos en la lista de rayos, esto se muestra a continuación.

1. **print**(Rayos.valid())

con lo que obtenemos la siguiente lista.

[[25][26][27][28][29][30][31][32][33][34][35][36][37][38][39][40][41][42][43][44][45][46][47][48][49][50][51][52][53][54][55]]

Podemos llamar ahora a la información del rayo usando estos valores y las palabras clave de la Tabla 3 (Columna central), sin embargo, al ejecutar la instrucción ***valid()*** de la línea 78 del ejemplo, se crea un nuevo juego de llamadas, están son las que se muestran en la Tabla 3 (columna derecha) estas son idénticas a las de la columna central, pero con el prefijo ***valid\_***, es importante aclarar que si no se ejecuta la instrucción ***valid()***. Las llamadas mostradas en la columna derecha estarán vacías por lo que si no se tiene este cuidado se obtendrán errores.

Los arreglos que se obtienen con este nuevo juego de llamadas contienen únicamente rayos válidos y ahora pueden utilizarse directamente pues se han eliminado en ellos los rayos vacíos, por lo tanto, la numeración estará desplazada según el número de espacios vacíos retirados. Dependiendo de la tarea que se desea realizar pudiese tener aplicación de un modo o el otro.

De la misma forma se crea un juego de llamadas para los rayos inválidos, como es de esperarse, solo se puede obtener la información del rayo inicial, estas llamadas aparecen en la Tabla 3 columna izquierda y está limitada a coordenadas de origen, dirección y longitud de onda. El prefijo para estas llamadas es ***invalid\_***.

|  |  |  |
| --- | --- | --- |
| Tabla 3. Atributos del contenedor ***Raykeeper*** | | |
| ***Todos los rayos*** | ***Rayos válidos*** | ***Rayos inválidos*** |
| Raykeeper.RayWave[#] | Raykeeper.valid\_RayWave[#] |  |
| Raykeeper.SURFACE[#] | Raykeeper.valid\_SURFACE[#] |  |
| Raykeeper.NAME[#] | Raykeeper.valid\_NAME[#] |  |
| Raykeeper.GLASS[#] | Raykeeper.valid\_GLASS[#] |  |
| Raykeeper.S\_XYZ) [#] | Raykeeper.valid\_S\_XYZ[#] |  |
| Raykeeper.T\_XYZ[#] | Raykeeper.valid\_T\_XYZ[#] |  |
| Raykeeper.XYZ[#] | Raykeeper.valid\_XYZ[#] | Raykeeper.invalid\_XYZ[#] |
| Raykeeper.OST\_XYZ[#] | Raykeeper.valid\_OST\_XYZ[#] |  |
| Raykeeper.S\_LMN[#] | Raykeeper.valid\_S\_LMN[#] |  |
| Raykeeper.LMN[#] | Raykeeper.valid\_LMN[#] | Raykeeper.invalid\_LMN[#] |
| Raykeeper.R\_LMN[#] | Raykeeper.valid\_R\_LMN[#] |  |
| Raykeeper.N0[#] | Raykeeper.valid\_N0[#] |  |
| Raykeeper.N1[#] | Raykeeper.valid\_N1[#] |  |
| Raykeeper.WAV[#] | Raykeeper.valid\_WAV[#] | Raykeeper.invalid\_WAV[#] |
| Raykeeper.G\_LMN[#] | Raykeeper.valid\_G\_LMN[#] |  |
| Raykeeper.ORDER[#] | Raykeeper.valid\_ORDER[#] |  |
| Raykeeper.GRATING[#] | Raykeeper.valid\_GRATING[#] |  |
| Raykeeper.DISTANCE[#] | Raykeeper.valid\_DISTANCE[#] |  |
| Raykeeper.OP[#] | Raykeeper.valid\_OP[#] |  |
| Raykeeper.TOP\_S[#] | Raykeeper.valid\_TOP\_S[#] |  |
| Raykeeper.TOP[#] | Raykeeper.valid\_TOP[#] |  |
| Raykeeper.ALPHA[#] | Raykeeper.valid\_ALPHA[#] |  |
| Raykeeper.BULK\_TRANS[#] | Raykeeper.valid\_BULK\_TRANS[#] |  |
| Raykeeper.RP[#] | Raykeeper.valid\_RP[#] |  |
| Raykeeper.RS[#] | Raykeeper.valid\_RS[#] |  |
| Raykeeper.TP[#] | Raykeeper.valid\_TP[#] |  |
| Raykeeper.TS[#] | Raykeeper.valid\_TS[#] |  |
| Raykeeper.TTBE[#] | Raykeeper.valid\_TTBE[#] |  |
| Raykeeper.TT[#] | Raykeeper.valid\_TT[#] |  |

Para borrar los rayos dentro de un ***Raykeeper*** se puede redefinir el contenedor, repitiendo su creación como en la línea 52 de código Python de ejemplo, otra manera es utilizando la implementación interna ***clean***.

1. Rayos.clean()

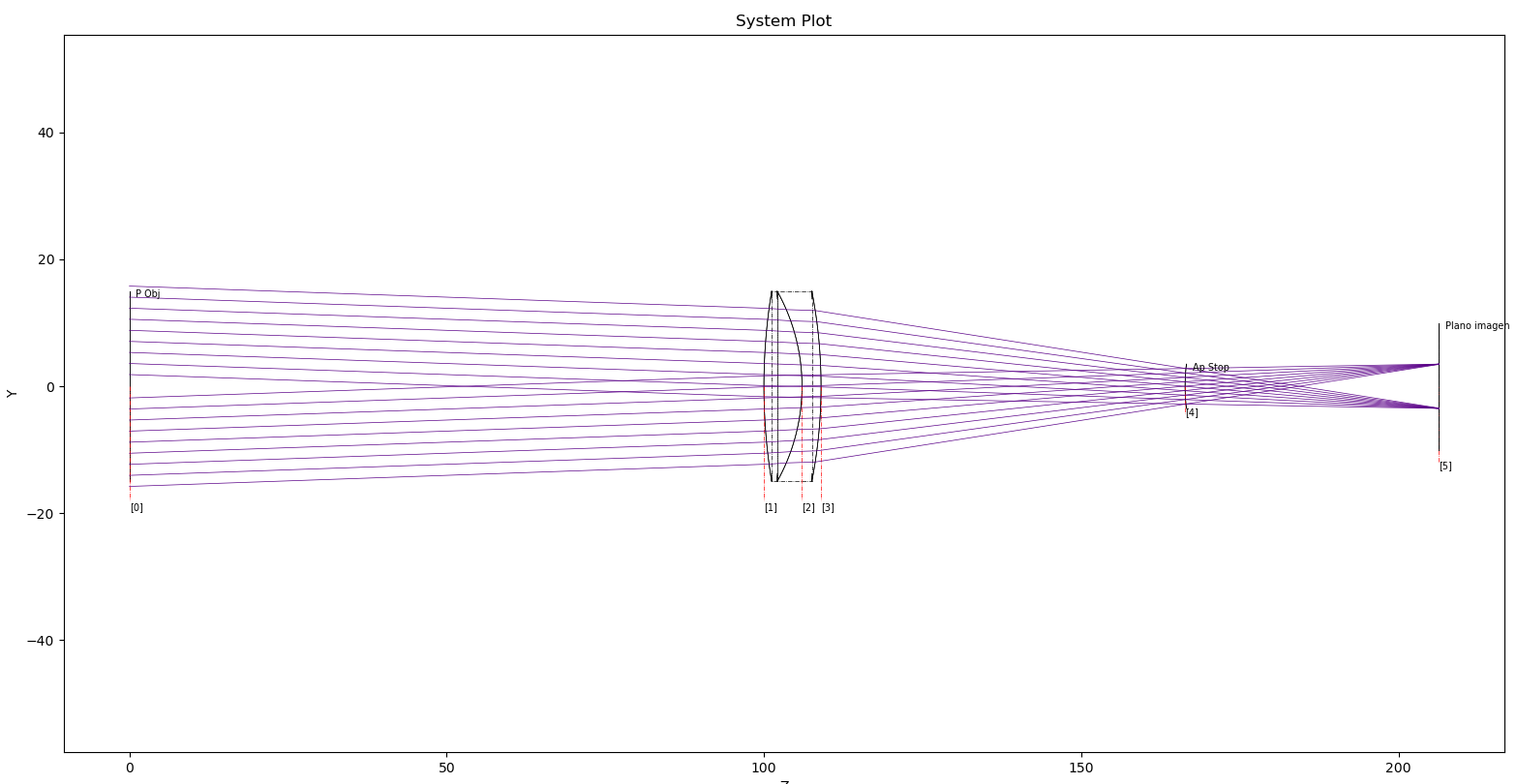
**Manejo del visor 3D**

* Rotar la simulación en cualquier dirección, botón izquierdo del ratón
* Zoom, mover el ratón arriba y abajo oprimiendo el botón derecho del ratón
* Arrastrar la simulación, arrastrar oprimiendo botón izquierdo del ratón y oprimiendo la tecla SHIFT del teclado.

**Herramienta pupilcalc**

La apertura del sistema o como comúnmente es conocido “Aperture Stop” es el elemento que define la cantidad de luz que atraviesa por todo el sistema óptico, la imagen de la apertura del sistema producida por los elementos ópticos anteriores a ella es conocida como la pupila de entrada, la imagen de la pupila de entrada producida por todo el sistema es conocida como la pupila de salida. A diferencia de lo que primeramente se pudiese suponer, la pupila de entrada no necesariamente está definida por el primer elemento em ocasiones el elemento que define la cantidad de luz no es una superficie óptica sino un elemento mecánico.

En la siguiente figura se muestra un ejemplo de un “Apertura stop” en un elemento posterior a una lente, específicamente en la superficie 4. Además, en la figura se muestran dos campos distintos, ambos atraviesan igualmente esta apertura de forma idéntica, aquí surge un problema pues para generar rayos que cubran uniformemente la apertura del sistema se requeriría realizar un trazado de rayos hacia el plano objeto y con esto definir las propiedades del rayo para que atraviese dicha zona de la pupila, nótese en la figura que los campos tienen distintos orígenes, por lo tanto generar uno a uno los rayos puede ser una tarea complicada de realizarse calculándolos manualmente.



Para facilitar la generación de rayos que atraviesan la pupila se tiene la herramienta pupilecalc, esta clase genera un objeto iterativo de la siguiente forma.



Donde *Doblete* es el sistema que generamos con la clase ***system***, *sup* es el numero de superficie que representa la apertura del sistema, en el ejemplo de la imagen esta es la superficie 4, y *W* es la longitud de onda para la cual se calculará la pupila de salida y de entrada. Para este caso donde al objeto se le ha llamado Pup, se obtienen los parámetros de las pupilas como se muestra en la siguiente tabla.

|  |  |
| --- | --- |
| **Atributos del objeto correspondientes a parámetros de la pupila** | |
| Radio pupila de entrada | Pup.RadPupInp |
| Posición pupila de entrada | Pup.PosPupInp |
| Radio pupila de salida | Pup.RadPupOut |
| Posición pupila de salida | Pup.PosPupOut |
| Posición pupila de salida respecto al plano focal | Pup.PosPupOutFoc |
| Orientación pupila de salida | Pup.DirPupSal |

La posición de la pupila es calculada incluso si el sistema cuenta con elementos desplazados e inclinados, en ese caso, esa pupila desplazada cobra relevancia en el calculo de las aberraciones del sistema.

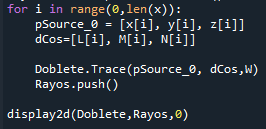
Además de obtener estos parámetros, también puede generar patrones de rayos en la pupila, calculando los cosenos directores y las coordenadas de origen definiendo los parámetros.

|  |  |
| --- | --- |
| Generación de rayos automáticos en base a la pupila | |
| Pup.Samp=# | Numero entero para el muestreo de rayos en la pupila, valor por defecto 5 |
| Pup.Ptype=**"tipo de arreglo"** | **"rteta"** Genera un rayo a un ángulo de la pupila unitaria a una posición radial, el radio y el ángulo deben definirse de la forma:  **Pup.rad=n**  **Pup.teta=m**  Donde n es un numero flotante de 0-1 y teta es el ángulo de 0 a 360 |
| **"chief"** Rayo principal que pasa por el centro de la pupila |
| **"hexapolar",** Arreglo de rayos hexapolar |
| **"square"**, Arreglo de rayos rectangular |
| **"fanx"**, Arreglo lineal solo en el eje x |
| **"fany"**, Arreglo lineal solo en el eje y |
| **"fan"**, Arreglo lineal en los ejes x y |
| **"rand"**, Arreglo al azar |
| Pup.FieldType=**"height" o "angle"** | Define el tipo de campo, en términos de la altura del objeto a la distancia del plano objeto con el parámetro **"height"**, para rayos paralelos que llegan a la pupila desde el infinito se utiliza el parámetro **"angle"** |
| Pup.FieldY=#  Pup.FieldX=# | Valor del campo en milímetros o grados en el eje X y Y dependiendo de el tipo de campo que se ha elegido en FieldType |

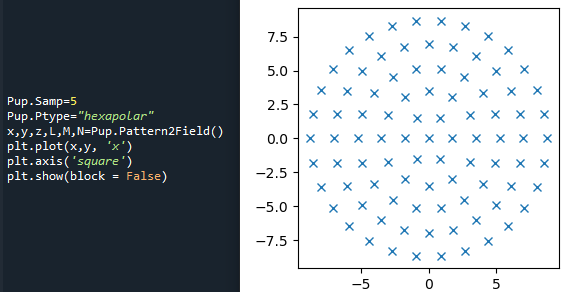
Los atributos mostrados en la tabla anterior definen el tipo de rayos que deseamos, para obtener el arreglo de rayos lo único que debemos de hacer es trasladarlos de la pupila unitaria a la pupila real obteniendo los cosenos directores y las coordenadas de origen en forma de arreglos de la siguiente forma:



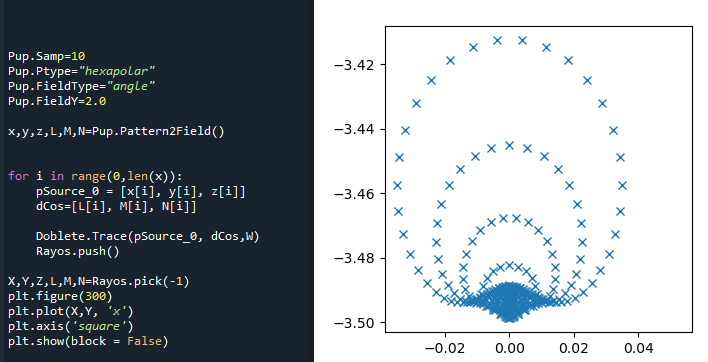
Donde x,y,z son las coordenadas de origen y L,M,N son los cosenos directores del rayo, de esta forma nosotros podemos iterar entre ellos y realizar el trazo de rayos a través del sistema como se muestra en la siguiente ejemplo, donde el sistema se llama **Doblete** y el contenedor de rayos **Rayos**.



Podemos también graficar los puntos de los rayos generados en el plano objeto, por ejemplo, en la siguiente imagen definimos un patrón “hexapolar”



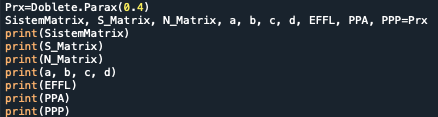
Podemos entonces generar el diagrama de manchas:



Es importante señalar que, si se desean analizar diferentes campos, se deben de realizar arreglos para cada campo y también guardar el trazado de rayos en diferentes contenedores si no se desean combinar los resultados.

**Herramienta Parax**

Realiza una serie de calculos paraxiales los cuales pueden ser llamados posteriormente desde system.



﻿

**Ejemplos**

La mejor manera de comprender el uso de la presente biblioteca es siguiendo los ejemplos incluidos en ella y que se listan a continuación:

|  |
| --- |
| Examp-Doblete-ComandosSystem.py |
| Examp-Doblete.py |
| Examp-Doblete-color.py |
| Examp-Lente\_Tilt.py |
| Examp-Lente\_Tilt-Nulls.py |
| Examp-Diffraction\_reflec.py |
| Examp-Diffraction\_trans.py |
| Examp-Zernike.py |
| Examp-Tel-2M.py |
| Examp-Tel-2M-Spyder.py |
| Examp-Tel-TAOS.py |
| Examp-Espejo\_Shift.py |
| Examp-Lente-cilindrica.py |
| Examp-Axicon.py |
| Examp-STL.py |
| Examp-STL-ARRAY.py |

RMS

Ojo mencionar que despuies del pupilcalc se debe de colocar nuevamente ignorevignetting si se necesita, upilcalc lo pone forzosamente en vignetting al salir