Publicaciones Técnicas
pubtecia@astro.unam.mx

Instituto de Astronomía
UNAM

Manual de Usuario **MU-2022-01**

Recibido: 07-09-2021
Recibido con revisión: 02-12-2021
Aceptado: 06-12-2021
Disponible en línea: 24-01-2022

KrakenOS. Manual de Usuario.

Joel Herrera V., Ilse Plauchu-Frayn, Carlos Guerrero P.

Instituto de Astronomía. Universidad Nacional Autónoma de México.
Carretera Tijuana-Ensenada km 103, Ensenada, B.C., México.

RESUMEN:

El presente manual muestra el uso de la biblioteca de simulación óptica y trazado de rayos *Kraken-Optical Simulator* o *KrakenOS*. Esta biblioteca ha sido desarrollada en el lenguaje *Python* y fue pensada para poder crecer y adaptarse a diferentes requerimientos de usuarios en distintas tareas de óptica. La biblioteca fue desarrollada bajo el paradigma de la programación orientada a objetos, buscando principalmente su simplicidad y la capacidad de ser incorporada en código de análisis o simulaciones numéricas en donde un trazado de rayos sea necesario. Al ser desarrollada en *Python*, existen una gran cantidad de herramientas con las cuales se puede combinar aumentando sus capacidades. Al final de este manual se pone a disposición del usuario un apéndice con una serie de ejemplos útiles que muestran las capacidades y funciones de esta biblioteca.

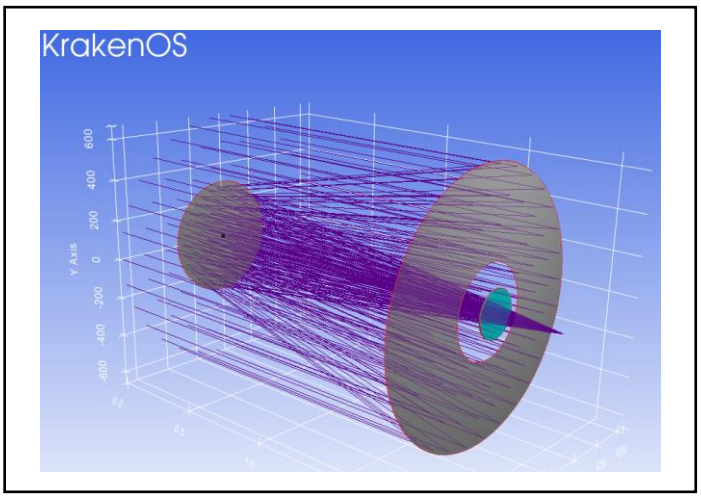
Contenido

1. INTRODUCCIÓN

KrakenOS (v1.0.0.4) es una herramienta para la simulación de sistemas ópticos, consta de una biblioteca desarrollada en el lenguaje de programación *Python* 3.0 y las bibliotecas *numpy*, *PyVTK* y *PyVista*, lo cual le permite proveer visualización tridimensional de los elementos ópticos. Esta herramienta se ha enfocado en el paradigma de la programación orientada a objetos. Esto, al parecer, nos brinda, de forma natural, una mayor simplicidad en la implementación de un sistema, como se verá en este documento.

El objetivo de este manual es presentar la lista con los comandos e implementaciones dentro de la biblioteca, además, también se muestra una serie de ejemplos en los cuales se utilizan todas las funciones disponibles.

Cabe señalar que el contenido de este documento es funcional a partir de la última versión v1.0.0.4, actualizada el 2 de diciembre del 2021.



2. PRERREQUISITOS E INSTALACIÓN

La biblioteca *KrakenOS* ha sido probada en Windows 10, MacOs 11.4 y Ubuntu 20.04LTS. En cualquiera de estos sistemas operativos, se recomienda una memoria RAM de 2 GB o superior, aunque esto depende principalmente del modelo simulado. El ejemplo `Examp_Tel_2M-STL_ImageSlicer.py` es una de las aplicaciones de *KrakenOS* que más memoria del procesador consume (2 GB). Para la mayor parte de las aplicaciones una RAM de 512 MB es suficiente.

La biblioteca ha sido probada con los siguientes paquetes y sus versiones:

- *Python* '3.7.4'
- *numpy* '1.18.5'
- *scipy* '1.7.1'
- *matplotlib* '3.4.3'
- *pyvista* '0.25.3'
- *pyvtk* '0.5.18'
- *vtk* '8.2'
- *csv* '1.0'

Para instalar la biblioteca existen dos maneras:

Como la biblioteca ya se encuentra en los repositorios (*Python package index*) entonces se puede instalar directamente en una terminal de sistema donde se encuentre instalado *Python* 3.7 o superior, con el comando:

```
pip install KrakenOS
```

Esto instalará todo lo necesario.

La otra manera de instalar la biblioteca es:

- Clonar el repositorio del siguiente enlace:

  https://github.com/Garchupiter/Kraken-Optical-Simulator

- Colocar el directorio *KrakenOS* en la misma ruta donde está el código que se desea ejecutar o, en su defecto, importar la biblioteca desde la ruta en donde el directorio de *KrakenOS* se encuentre.
- Todos los archivos incluidos en el directorio y cuyos nombres inician con la palabra "`Ejemplo - -`" son códigos con ejemplos, los cuales hacen uso de las distintas funciones de esta biblioteca.

OJO: si se descarga la biblioteca de GitHub, todos los ejemplos tienen una cabecera que verifica si la biblioteca está instalada y si no, supone que la biblioteca está en el directorio superior, tal y como lo estaría en la carpeta descargada de GitHub, así que los ejemplos funcionarán independientemente si no hay una versión de *KrakenOS* instalada con el comando "**`pip`**".

3. CLASES Y ATRIBUTOS

La biblioteca se ha simplificado al grado de contar únicamente con dos clases de objetos para la definición de un sistema. Estos objetos son `surf` y `system`, cuya aplicación se describe más adelante en la Tabla 1 y Tabla 2.

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`. Cada interfaz óptica, desde el plano objeto hasta el plano imagen, contiene atributos de tamaño, forma, material u orientación.

En la Tabla 1 se presentan los atributos que deben ser definidos para el objeto `surf`.

TABLA 1

Atributos de la clase *surf*

`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. Valor por defecto: (0,0)
`surf.Glass="AIR"`	Nombre de vidrio contenido en el catálogo de *Zemax*, los nombres para aire, vidrio, nulas y superficies absorbentes son “AIR”, “MIRROR”, “NULL” y “ABSORB” respectivamente. Se dan detalles más adelante.
`surf.Note="None"`	Útil para agregar notas del usuario en una superficie. Valor por defecto: None.
`surf.Rc=9999999999.0`	Radio de curvatura paraxial en milímetros. Valor por defecto: 9999999999.0 o 0.0 para un plano.
`surf.Cylinder_Rxy_Ratio=1`	Razón entre el radio de curvatura axial y sagital. De utilidad para lentes cilíndricas y toroides. Valor por defecto: 1
`surf.Axicon=0`	Simulación de axicones, para valores diferentes de cero se genera un axicón con el ángulo introducido. Valor por defecto: 0
`surf.Thickness=0.0`	Separación entre esta superficie (milímetros) y la siguiente superficie. Valor por defecto: 0.0.
`surf.Diameter=1.0`	Diámetro exterior de la superficie en milímetros. Valor por defecto: 1.0.
`surf.InDiameter=0.0`	Diámetro interno de la superficie en milímetros. Útil para elementos como un espejo primario con apertura central. Valor por defecto: 0.0.
`surf.k=0.0`	Constante de conicidad para superficies cónicas clásicas, k=0 para esférica, k=-1 para parábola, etc. Valor por defecto: 0.0
`surf.DespX=0.0`	Desplazamiento de la superficie en el eje X, Y y Z (en milímetros). Valores por defecto: 0.0
`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). Valores por defecto: 0.0
`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 las traslaciones. Valor por defecto: 0
`surf.AxisMove=1`	Define lo que ocurrirá con el eje óptico después de una transformación de coordenadas. Si el valor es 0 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 la transformación. Si el valor es diferente, por ejemplo 2, entonces el eje óptico será afectado al doble. Lo anterior es de utilidad en espejos planos (véase ejemplo más adelante). Valor por defecto: 1
`surf.Diff_Ord=0.0`	Orden de difracción. Convierte al elemento en una rejilla de difracción. El valor de radio de curvatura se omite de forma automática para que sea una rejilla de difracción plana. Esta opción puede utilizarse en transmisión o en refracción. Valor por defecto: 0.0
`surf.Grating_D=0.0`	Separación en milímetros entre las líneas de la rejilla de difracción. Valor por defecto: 0.0.
`surf.Grating_Angle=0.0`	Ángulo de las líneas de la rejilla en el plano de la superficie, es decir, alrededor del eje óptico. Esto es útil para simular dispersión cónica. Valor por defecto: 0.0 (grados).
`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. [2]
`surf.ShiftX=0`	Desplazamiento en milímetros 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. Valor por defecto: 0.
`surf.ShiftY=0`	
`surf.Mask=0`	Tipo de máscara a aplicar. Los diferentes valores son: (0) No aplicar máscara, (1) Utilizar máscara como apertura, (2) Utilizar máscara como obstrucción. Valor por defecto: 0

`surf.Mask_Shape=Objeto_3D`	Forma de la máscara a aplicar, (Máscara construida con biblioteca *PyVista* (*pv*)). *Pyvista* [1] y ejemplo en Apéndice A.20.
`surf.AspherData=np.zeros(Arreglo)`	Arreglo de coeficientes para superficie asférica. Arreglo es una lista del tipo *numpy*.
`self.ExtraData=[f, coef]`	Superficie creada por el usuario con una función de sagita dependiente de (x, y, V), donde V puede contener un arreglo de coeficientes utilizable por la función: -Se define la función de sagita como: def f(x,y,E): DeltaX=E[0]*np.rint(x/E[0]) DeltaY=E[0]*np.rint(y/E[0]) x=x-DeltaX y=y-DeltaY s = np.sqrt((x * x) + (y * y)) c = 1.0 / E[1] InRoot = 1 - (E[2] + 1.0) * c * c * s * s z = (c * s * s / (1.0 + np.sqrt(InRoot))) return z -Se definen los coeficientes si es necesario en forma de una lista. Coef = [3.0, -3, 0] -Se le asigna la forma de la función a la superficie L1c.ExtraData = [f, Coef]
`Surf.Error_map=[X,Y,Z, SPACE]`	Recibe un mapa de error generado con un arreglo del tipo *numpy* para las coordenadas en X, Y y la altura en Z con un valor de espacio entre X, Y generado con el siguiente código. Ejemplo de la generación del mapa de error: 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`	Valor igual a 1 para que el elemento sea dibujado en el graficado 3D o valor igual a 0 para omitir el dibujo del elemento.
`surf.Color=[0,0,0]`	Define el color del elemento en el formato [1,1,1]. Porcentajes de color RGB (red, green, blue). Valor por defecto: (0,0,0) es el color negro. Otros colores son: rojo (1,0,0), verde (0,1,0) y azul (0,0,1).
`surf.Solid_3d_stl="None"`	Ruta del objeto sólido 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 atraviesa el rayo. Para comprender la manera en que estos elementos son llamados, el lector puede consultar el ejemplo en el Apéndice A.11.

En la Tabla 2 se presentan las implementaciones públicas del objeto `system`.

TABLA 2

Implementaciones y atributos de la clase *system*

`system.Trace(pS, dC, wV)`	***Trace*** es la implementación principal del objeto ***system***, ésta 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”. Véanse los siguientes ejemplos: pS = [1.0, 0.0, 0.0] dC=[0.0,0.0,1.0] wV=0.4

`system.NsTrace(pS, dC, wV)`	Similar a ***Trace***, pero de forma no secuencial. Los parámetros del rayo se definen de la misma manera que en ***Trace***.
`Prx=system.Parax(w)`	Devuelve los siguientes cálculos paraxiales accesibles también desde ***system*** que están ordenados en una lista con los siguientes elementos: Prx=SistemMatrix, S_Matrix, N_Matrix, a, b, c, d, EFFL, PPA, PPP, CC, N_Prec, DD
`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`	
`system.SURFACE`	Devuelve una lista del número de superficies por las que pasó el rayo.
`system.NAME`	Devuelve una lista de los nombres de las superficies por las que pasó el rayo. Si no se indica 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.GLASS`	Devuelve una lista de los materiales que ha atravesado el rayo.
`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 un valor único con el camino óptico total del rayo desde la fuente hasta el último punto de intersección.
`system.TOP_S`	Devuelve una lista acumulativa con valores 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. Dichos valores los obtiene del catálogo de materiales proporcionado originalmente.
`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]* del 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 atraviesa el rayo. Este índice es calculado con la dispersión del material y la longitud de onda del rayo en cuestión.
`system.N1`	Índices de refracción después de cada interfaz por las que atraviesa el rayo. Este índice es calculado con la dispersión del material y la longitud de onda del rayo en cuestión. La diferencia con `system.N0` es que la lista comienza desde el segundo índice de refracción. Esto es de utilidad para diferenciar entre la lista de índices del medio antes y después de una iteración. Ejemplo: N0= [n1, n2, n3, n4, n5] N1= [n2, n3, n4, n5, n5] De tal forma que si queremos saber la dirección de un rayo en la primera interfaz utilizaremos N0[0] y N1[0].

`system.WAV`	Longitud de onda del rayo en cuestión (micras). Aunque este comando devuelve una lista todos los valores, estos 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.
`system.G_LMN`	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.ORDER`	Devuelve una lista con los órdenes de difracción asociados al rayo en cuestión.
`system.GRATING_D`	Distancia en micras 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`	
`system.TP`	
`system.TS`	
`system.TTBE`	Energía total transmitida o reflejada por elemento.
`system.TT`	Energía total transmitida o reflejada total.
`system.targ_surf(int)`	Limita el trazado de rayos hasta la superficie definida. Este es un número entero igual al número de superficie, si el valor es cero (por defecto) significa que el trazado de rayos se realiza hasta la última superficie.
`system.flat_surf(int)`	Se define con el número entero de la superficie que requiere hacerse plana. El valor -1 se usa para restaurar la superficie a su forma original.

4. TRABAJANDO CON LA BIBIOTECA KrakenOS

En esta sección, se presenta un ejemplo en el cual se genera un rayo que pasará por varias superficies para, posteriormente, extraer información de éste.

Es muy importante mencionar que, en *Python*, las clases definen a un objeto con sus atributos. Estos, al crearse, adquieren todos los atributos disponibles en la clase. Por ejemplo, si se crea un objeto de la clase `system`, esta creación requerirá como parámetro una lista de superficies y una configuración, estos pueden tener el nombre que el usuario desee.

Lo anterior se muestra más adelante dentro de este manual, específicamente en el Código 3. En este código, la lista de superficies es nombrada ***A***, la configuración del sistema es `configuracion_1` y el sistema óptico creado es nombrado por conveniencia como ***Doblete***. Dichos nombres (***A***,

`configuración_1` y ***Doblete***) no están determinados dentro de la biblioteca; simplemente son nombres que hemos asignado a estos nuevos objetos, con los cuales podemos interactuar a través de los atributos de la clase utilizada para crearlos.

4.1. GENERACIÓN DE UN RAYO

Todos los rayos y sistemas ópticos son dibujados y trazados de izquierda a derecha, sin embargo, *KrakenOS* permite que se tracen rayos de derecha a izquierda.

La separación entre superficies se define con signo positivo de izquierda a derecha y negativo en caso contrario. El usuario notará que no es extraño encontrarse con valores de grosor o `Thickness` negativos, el ejemplo más claro es en el uso de espejos. Véase Apéndice A.22 (EJEMPLO - TEL 2M PUPILA).

Dentro de *Python*, y para hacer uso de la biblioteca *KrakenOS*, importamos primero las bibliotecas *numpy* y *KrakenOS*. En el siguiente ejemplo, se muestra que, después de haber sido importadas ambas bibliotecas, estas han sido renombradas como "`np`" y "`Kos`", para la biblioteca *numpy* y *KrakenOS*, respectivamente.

```
import numpy as np
import KrakenOS as Kos
```

Para obtener ayuda sobre la biblioteca en línea de comandos, después de importar la biblioteca se puede utilizar la documentación en código por medio de:

```
help(Kos), help(Kos.system) o help(Kos.surf)
```

El ejemplo que detallamos a continuación se encuentra dentro del programa: EJEMPLO -- RAY (véase Apéndice A.1). Todas las superficies ópticas deben ser declaradas por separado. Al declarar una superficie se hace uso de todos o algunos de los posibles parámetros que ésta posee. Para una descripción de cada uno de estos parámetros véase la Tabla 1. En el Código 1 se muestra cómo se han definido cinco superficies diferentes.

Código 1	

`P_Obj = Kos.surf()`
`P_Obj.Rc = 0.0`
`P_Obj.Thickness = 0.1`
`P_Obj.Glass = "AIR"`
`P_Obj.Diameter = 30.0`
	Aquí se define el plano objeto como una superficie del tipo `surf`, se asignan valores a los atributos de Radio de curvatura (`Rc`), separación con la siguiente superficie (`Thickness`), el tipo de material (`Glass`) y el diámetro de éste (`Diameter`).

`L1a = Kos.surf()`
`L1a.Rc = 92.847`
`L1a.Thickness = 6.0`
`L1a.Glass = "BK7"`
`L1a.Diameter = 30.0`
`L1a.Axicon = 0`
	Aquí se define la primera cara de un doblete y sus atributos. El material BK7 es uno de los materiales más utilizados en óptica, es un vidrio del tipo `Crown` (baja dispersión).

`L1b = Kos.surf()`
`L1b.Rc = -30.716`
`L1b.Thickness = 3.0`
`L1b.Glass = "F2"`
`L1b.Diameter = 30`
	Aquí se define la segunda cara de un doblete y sus atributos. El material F2 es uno de los materiales más utilizados en óptica, es un vidrio del tipo `Flint` (alta dispersión).
	El BK7 y el F2 son muy utilizados en la construcción de dobletes acromáticos. [3, Sec. 11.6]

`L1c = Kos.surf()`
`L1c.Rc = -78.19`
`L1c.Thickness = 97.37`
`L1c.Glass = "AIR"`
`L1c.Diameter = 30`
	Aquí se define la tercera cara de un doblete y sus atributos.

`P_Ima = Kos.surf()`
`P_Ima.Rc = 0.0`
`P_Ima.Thickness = 0.0`
`P_Ima.Glass = "AIR"`
`P_Ima.Diameter = 18.0`
`P_Ima.Name = "Plano imagen"`
	Aquí se define la superficie referente al plano imagen y sus atributos.

Dentro de la biblioteca *KrakenOS* se cuenta con el objeto `Setup` que tiene únicamente la función de configuración del entorno. Existen ciertos parámetros que deben de ser cargados desde la creación de un sistema óptico, tal es el caso de los catálogos de vidrios que se utilizarán. El contenido de esta clase (`Setup`) es únicamente el catálogo de SCHOTT que es el catálogo por defecto y es proveído libremente en (https://www.schott.com/en-gb/products/optical-glass-p1000267/downloads). Para incluir más catálogos se deben seguir los pasos mostrados en Código 2.

Una buena fuente para obtener catálogos para vidrios de otros fabricantes es el proyecto de GitHub (https://github.com/nzhagen/zemaxglass), en donde se realiza una descripción de los parámetros internos de los archivos de catálogo `*.AGF`

Al declararse un vidrio para una superficie, por ejemplo, `Surf1.Glass = "NombreVidrio"` éste se debe de escribir tal cual se encuentra escrito en el catálogo. Existen nombres reservados para el sistema como "`AIR`", "`MIRROR`", "`NULL`" para superficies nulas y "`ABSORB`" para superficies completamente absorbentes. Las superficies definidas como "`NULL`" no realizan ninguna acción en la luz, pero pueden ser utilizadas para transformaciones del sistema coordenadas, puesto que, para las superficies subsecuentes, este elemento sí existe.

Alternativamente, el usuario puede definir directamente de forma numérica el valor del índice de refracción, en este caso *KrakenOS* lo utiliza directamente. Esto es irreal porque la longitud de onda varía con el índice de refracción, así que el usuario debe de tomar en cuenta esto al trazar rayos en diferentes longitudes de onda.

En futuras versiones de la biblioteca *KrakenOS* se podrán establecer diferentes configuraciones para un sistema, lo cual puede tener utilidad para sistemas con múltiples configuraciones con distintas condiciones. Para fines prácticos de este manual de usuario, cargaremos la configuración por defecto en la variable `configuracion_1`, lo cual se hace de la siguiente manera:

```
configuracion_1 = Kos.Setup()
```

Código 2

```
config_1 = Kos.Setup()
"""En caso de un catalogo de vidrios externo:"""

file = ["/Tu ruta a tu catalogo/SCHOTT.AGF"]
config_1.Load(file)

"""En caso de muchos catálogos:"""

file = ["/Tu ruta /Cat1.AGF", "/ Tu ruta /Cat2.AGF"]
config_1.Load(file)
```

Como se muestra en el Código 3, se crea un arreglo ***A*** con los elementos declarados para todas las superficies, se carga la configuración por defecto y, posteriormente se crea un sistema óptico ***Doblete*** con todas las superficies contenidas en ***A*** y la `configuracion_1`.

Código 3

`A = [P_Obj, L1a, L1b, L1c, P_Ima]`	Aquí se crea una lista nombrada como ***A*** con las superficies creadas.
`configuracion_1 = Kos.Setup()`	Se carga la configuración por defecto con los catálogos en el archivo "`SetupClass.py`".
`Doblete = Kos.system(A, configuracion_1)`	Y se genera el objeto nombrado ***Doblete***.

Todo rayo tiene tres parámetros: las coordenadas de origen representadas en este ejemplo por la variable ***XYZ,*** la cual es una lista del tipo $[x_1, y_1, z_1]$ y una dirección definida por cosenos directores, representada por la variable ***LMN,*** que es una lista del tipo [L, M, N]. Esta última variable de cosenos directores, para un rayo paralelo al eje óptico, tendría los siguientes valores [0,0,1], es decir que este vector no tiene una componente en X o en Y. Las ecuaciones que definen los cosenos directores están dadas por:

$$L = \frac{x_2 - x_1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}}$$

$$M = \frac{y_2 - y_1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}}$$

$$N = \frac{z_2 - z_1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}}$$

Donde los puntos de intersección del rayo en la primera superficie están dados por $[x_2, y_2, z_2]$.

Antes de realizar el trazado de rayos se desconoce el punto en que éste tocará la primera superficie. Por ello, es necesario definir los cosenos directores en función de la dirección del rayo, o, en otras palabras, de su ángulo de incidencia. Supongamos un rayo que llega en un ángulo ***Theta*** con respecto al eje Y, pero sin ningún ángulo con respecto al eje X. Entonces, este rayo sólo tendrá componentes en el eje Z y Y, y el valor de ***L*** seguirá siendo igual cero, mientras que ***M*** será ***Seno***(***Theta***) y ***N*** será ***Coseno***(***Theta***).

El tercer parámetro por definir es la longitud de onda, la cual es expresada como ***W*** igual a 0.4 μm en el Código 4.

Código 4

```
XYZ = [0, 14, 0]
Theta = 0.1 # Un campo cualquiera para el ejemplo
LMN = [0.0, np.sin(np.deg2rad(Theta)), -np.cos(np.deg2rad(Theta))]
W = 0.4
```

En el Código 4 se ha definido la dirección del rayo de forma muy general con el ángulo de incidencia del mismo `Theta`**.** Para generar rayos con distintas componentes en dirección X o Y se ha integrado una herramienta para generación automática de rayos dentro de la herramienta ***PupilCalc*** que se verá más adelante.

Continuando con el ejemplo anterior, se realiza el trazo del rayo a través del sistema `Doblete` de la siguiente forma:

```
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:

```
print(Doblete.GLASS)
```

El comando anterior nos devuelve como salida el siguiente arreglo `['BK7', 'F2', 'AIR', 'AIR']`. En particular, para este ejemplo, el rayo ha pasado por `BK7` que es `L1a`, `F2` que es `L1b` y dos superficies de aire que son `L1c` y el plano imagen (`P_Ima`).

Otro tipo de información que es posible extraer serían las coordenadas del rayo en todas las superficies y/o los cosenos directores. Lo anterior se obtiene con los comandos `XYZ` y `LMN` de la siguiente forma:

```
print(Doblete.XYZ)
print(Doblete.LMN)
```

con lo cual obtenemos las siguientes salidas de consola 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 directamente sobre ellos las operaciones que deseemos. A continuación, vemos la forma de ambos arreglos:

```
print(np.shape(Doblete.XYZ))
print(np.shape(Doblete.LMN))
```

lo cual produce como salida:

```
np.shape(Doblete.XYZ) = (5, 3)
np.shape(Doblete.LMN) = (4, 3)
```

Nótese que `Doblete.XYZ` contiene cinco arreglos de tres elementos (coordenadas X,Y,Z en el espacio tridimensional). Por otro lado, `Doblete.LMN` contiene solo cuatro elementos, debido a que solo muestra los cosenos directores entre las superficies. Esto es: si un sistema tiene cinco elementos desde el plano objeto hasta el plano imagen, entonces solo existen cuatro segmentos de rayo entre superficies y, por lo tanto, solo cuatro juegos de cosenos directores. En la siguiente línea de comando, cada juego de cosenos entre las superficies es ejemplificado con la simbología "->":

```
A=[P_Obj -> L1a -> L1b -> L1c -> P_Ima]
```

4.2. EXTRACCIÓN DE LA INFORMACIÓN DEL RAYO

Una vez que hemos realizado el trazo del rayo con la opción `Trace`, podemos extraer información de este con los atributos del objeto `Doblete`, haciendo uso de los parámetros descritos en la Tabla 2. Con la información obtenida es posible generar gráficas, pero generalmente es necesario contar con un gran número de rayos. Por ejemplo, un diagrama de manchas es una herramienta que en óptica necesita trazar un gran número de rayos que provengan del objeto, atraviesen la pupila del sistema y lleguen al plano imagen. Los puntos de intersección de estos rayos son graficados y proporcionan gran cantidad de información sobre las aberraciones del sistema.

Con el fin de conservar los resultados para uno o más rayos, tenemos el contenedor de rayos en la clase `Raykeeper`. Esta es una clase de objeto adicional que es independiente de `Surf` y `System`. Este tiene sus propios atributos e implementaciones; sus atributos son la manera de obtener la información del rayo y pueden ser vistos en la Tabla 3. Además de sus atributos, la clase `Raykeeper` contiene las siguientes implementaciones, `push()`, `clean()` y `pick()`, que se discutirán más adelante.

Continuando con el ejemplo `Ray` (Apéndice A.1), creamos el objeto "`Rayos`" del tipo `Raykeeper`, el cual contendrá toda la información del rayo o rayos:

```
Rayos = Kos.raykeeper(Doblete)
```

`push()` es la implementación con la cual indicamos que se guarde el rayo recién trazado dentro de `Doblete`. Una ventaja de esto es que el objeto `Doblete` no tiene que guardar una memoria con todos los rayos que tracemos, de esta tarea se encarga el contenedor de rayos. De esta forma podemos generar contenedores de rayos para diferentes circunstancias según se desee. 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 encontrará muy útil esta herramienta de *KrakenOS*.

La siguiente línea de comando indica cómo es guardada esta información, la cual se debe repetir cada vez que un rayo es trazado por `Doblete`:

```
Rayos.push()
```

4.3. GENERACIÓN DEL GRÁFICO DEL SISTEMA ÓPTICO

Si se quiere generar el gráfico del sistema óptico, se cuenta con dos herramientas: `Display2D` y `Display3D`. Los parámetros de entrada para estas dichas herramientas son: el sistema en sí mismo y un contenedor de rayos.

Adicionalmente, cada una de estas herramientas recibe un parámetro extra. En el caso de `Display2D(System, Raykeeper, parámetro)`, con el cual se genera un gráfico del sistema en 2 dimensiones, "***parámetro***" toma el valor 0 para indicar que el gráfico será en el plano XZ o el valor 1 en el plano YZ. Así mismo, si se quiere generar un gráfico del sistema en 3 dimensiones, usaremos `Display3D(System, Raykeeper, parámetro)` donde "`parámetro`" puede tomar los siguientes valores:

- 0 para el despliegue de los elementos completos
- 1 para el despliegue de las superficies con un corte de 1/4
- 2 para el despliegue de las superficies con un corte de 1/2

Para el ejemplo en este documento tendremos que reemplazar `system` y `Raykeeper` por los objetos creados con ellos de la siguiente forma:

```
Kos.Display2D(Doblete,Rayos,0)
```

La línea de comando anterior generará una gráfica que se despliega en ***Matplotlib***. Recordemos que solo guardamos un rayo en el contenedor. Este rayo se muestra en color morado, ya que depende de la longitud de onda utilizada originalmente para definir el rayo (i.e., W=0.4). Si se hubiese asignado un valor a W=0.6, el rayo se hubiera graficado automáticamente en color rojo. Por lo tanto, el color del rayo es definido por el programa dependiendo del valor de longitud de onda utilizada. Si la longitud de onda asignada fuera superior o inferior al rango del espectro visible, entonces el rayo será desplegado en color negro.

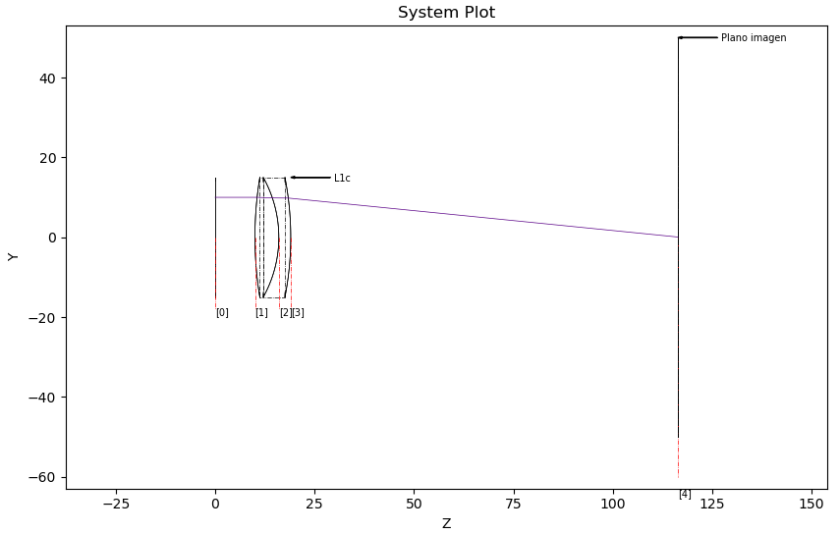


Figura 1: *Visualización 2D de un rayo trazado a través de un doblete.*

`Display2d` tiene un cuarto parámetro opcional, cuyo valor por defecto es igual a cero. Si este valor se modifica por un valor mayor a cero (i.e., valor positivo), entonces cada rayo es graficado con una flecha adicional. Lo anterior es de utilidad si se desea indicar la dirección del rayo en el gráfico. El valor de este cuarto parámetro define el tamaño de la flecha del rayo. El efecto de utilizar esta opción puede verse más adelante en la *Figura 4*.

```
Kos.Display2D(Doblete,Rayos,0, 1)
```

Si, por otro lado, se quiere desplegar el gráfico del mismo sistema óptico en su forma tridimensional, utilizamos la siguiente línea de comando:

```
Kos.Display3D(Doblete,Rayos,2)
```

Con el comando anterior se abrirá la siguiente ventana del visualizador, como se muestra en la *Figura 2*.

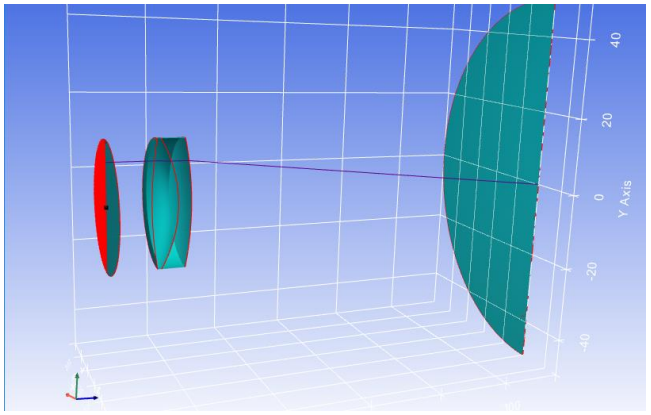


Figura 2: *Visualización 3D de una sección transversal del doblete y un rayo*

También es posible almacenar varios rayos. Por ejemplo, en el Código 5 se anida la creación del rayo, su trazo y su almacenado dentro de un ciclo ***for***.

Código 5	

`for x in range(-10, 10):`
`  XYZ = [0.0, x, 0.0]`
`  LMN =[0.0,0.0,1.0]`
`  W=0.4`
	Aquí se crea un rayo paralelo al eje óptico dentro del ciclo `for` y se indica la altura en la posición "`x`".

`  Doblete.Trace(XYZ, LMN,W)`	Se realiza el trazado de los rayos.
`  Rayos.push()`	Se almacena la información del rayo.
`Kos.Display3D(Doblete,Rayos,2)`	Y se despliega el sistema con todos los rayos.

Con el Código 4 se obtiene cualquier número arbitrario N de rayos, en vez de uno solo. En este ejemplo, 20 rayos dentro del intervalo -10 a 10 (20 rayos porque *Python* no realiza el último elemento del ciclo ***for***). Si el rayo no toca ninguna superficie, entonces este rayo no es trazado. En la *Figura 3* se puede notar que los 20 rayos generados atraviesan el sistema óptico.

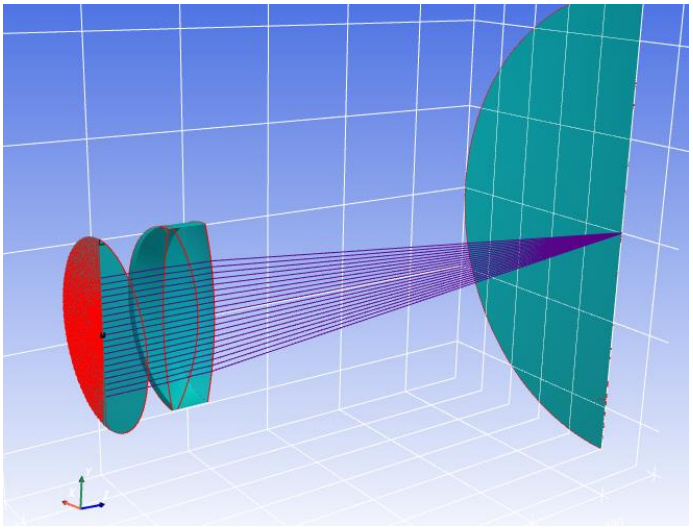


Figura 3: *Despliegue en 3D del sistema óptico con varios rayos*

El contenedor de rayos comprende una lista que 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 es necesario tener en cuenta ciertas consideraciones.

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 dirección. En cualquier caso, el rayo es tomado y la información existente de este: punto de origen, cosenos directores y longitud de onda, pero el resto de los datos estarán vacíos.

Lo anterior puede ser de utilidad si, por ejemplo, lanzamos 100 rayos. Para conocer el número de rayos almacenados, `Raykeeper` tiene una variable interna `nrays` que se puede llamar directamente de la siguiente forma:

```
print(Rayos.nrays) Resultado: 100
```

Por lo tanto, tendríamos 100 posiciones con rayos sobre los cuales podemos solicitar información al contenedor, esto mediante el uso de los parámetros mostrados en la Tabla 3 (columna central). Dichos parámetros poseen la misma información descrita en la Tabla 2 para la clase `system`. Las llamadas al contenedor `Raykeeper` dependen del número de rayos definidos 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 se ha mencionado anteriormente, existen rayos que no tocaron ninguna superficie y que nombramos "**rayos vacíos**", y rayos que sí tocan alguna superficie y que nombramos "**rayos válidos**". Podemos solicitarle al contenedor una lista de los rayos válidos contenidos en la lista de rayos con el siguiente comando:

```
print(Rayos.valid())
```

Así, 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]]
```

Una vez hecho lo anterior, podemos solicitar información del rayo haciendo uso de estos valores y las palabras clave de la Tabla 3 (columna izquierda). Sin embargo, al ejecutar la instrucción `valid()` de la línea (`Rayos.valid()`), se crea un nuevo juego de llamadas. Estas llamadas son las que se muestran en la columna central de la Tabla 3, las cuales son similares a aquellas en la columna izquierda (Tabla 3), pero con el prefijo "`valid_`". Es importante hacer notar que, si no se añade la instrucción `valid`, las llamadas mostradas en la columna **central** estarán vacías y, por lo tanto, se obtendrán errores.

Los arreglos que se obtienen con este nuevo juego de llamadas contienen únicamente rayos válidos, mismos que pueden utilizarse directamente pues se han eliminado los rayos vacíos. Por lo tanto, la numeración estará desplazada según el número de rayos vacíos descartados. Dependiendo de la tarea que se desea realizar pudiese tener aplicación de un modo u otro.

De la misma forma se crea un juego de llamadas para los rayos inválidos, que son los rayos vacíos, y cuyo trazado no se realizó, ya que no intersecan ninguna superficie o porque no llegan al plano imagen. Las llamadas a estos rayos se muestran en la Tabla 3 (columna derecha). A partir de estas llamadas solo es posible extraer la información del rayo inicial, la cual está limitada a coordenadas de origen, dirección y longitud de onda. El prefijo para dichas 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[#]	
Otros		
Raykeeper.valid	Enumera el número de rayos válidos	
Raykeeper.nrays	Enumera el número de rayos totales guardados	

Para eliminar la información de los rayos contenida dentro del contenedor `Raykeeper`, bastará con redefinir dicho contenedor o borrar su contenido utilizando la implementación interna `clean`.

```
Rayos.clean()
```

5. HERRAMIENTA PARAX

La herramienta `Parax` es una implementación de la clase `system`. Esta herramienta realiza una serie de cálculos paraxiales que básicamente proporcionan toda la información necesaria para el trazado de rayos paraxiales en forma de matriz. Estos cálculos se realizan con el objeto del sistema y para la longitud de onda solicitada de la siguiente manera:

```
Prx = Doublet.Parax(0.4)
SistemMatrix, S_Matrix, N_Matrix, a, b, c, d, EFFL, PPA, PPP, C, N, D = Prx
```

Matriz total del sistema:

```
SistemMatrix = [ [ 8.69329466e-01 -9.80840791e-03]
                 [ 1.00167690e+02  2.01470656e-02] ]
```

Matriz superficie tras superficie:

```
S_Matrix = [matrix([[1., 0.],[0., 1.]]),
            matrix([[ 1.,  0.],[10.,  1.]]),
            matrix([[ 0.65323249, -0.00361793],[ 0.        ,  1.        ]]),
            matrix([[1., 0.],[6., 1.]]),
            matrix([[0.92657697, 0.00239038],[0.        , 1.        ]]),
            matrix([[1., 0.],[3., 1.]]),
            matrix([[ 1.65215474, -0.00833986],[ 0.        ,  1.        ]]),
            matrix([[ 1.        ,  0.        ],[97.37604743,  1.        ]]),
            matrix([[1., 0.],[0., 1.]]),
            matrix([[1., 0.],[0., 1.]])]
```

Nombre de la matriz con fines de apoyo en la identificación por el usuario:

```
N_Matrix = [ 'R sup: 0',
                'T sup: 0 to 1',
             'R sup: 1',
             'T sup: 1 to 2',
                'R sup: 2',
             'T sup: 2 to 3',
             'R sup: 3',
             'T sup: 3 to 4',
             'R sup: 4',
             'T sup: 4 to 5']
```

Valores [**abcd**] de la matriz total del sistema:

```
a = 0.8693294662072478
b = -0.009808407908592333
c = 100.16768993870667
d = 0.02014706564149671
```

"Distancia focal efectiva" (**EFFL**), "Plano principal anterior" (**APP**) y "Plano principal posterior" (**PPP**):

```
EFFL = 101.9533454684305
PPA = -99.89928472490783
PPP = 13.322298074316684
```

Lista de curvaturas (C), Lista de índices de refracción (N) y lista de separaciones entre las superficies(D):

```
C = [ 1.00000000e-12, 1.04332892e-02, -3.25562791e-02, -1.27881671e-02, 1.00000000e-12]
N = [1.0, 1.5308485382492993, 1.6521547400480732, 1.0, 1.0]
D = [10., 6., 3., 97.37604743, 0.]
```

6. HERRAMIENTAS PupilCalc

La apertura del sistema o, como comúnmente se conoce, "*Aperture Stop*" [4, Sec. 6.2], es el elemento que define la cantidad de luz que atraviesa por todo el sistema óptico. La imagen de la apertura del sistema óptico, producida por los elementos ópticos anteriores a ella, es conocida como la pupila de entrada. Así mismo, la imagen de la pupila de entrada, producida por todo el sistema óptico, es conocida como la pupila de salida. A diferencia de lo que inicialmente se podría suponer, la pupila de entrada no necesariamente está definida por el primer elemento del sistema óptico, ya que en ocasiones el elemento que define la cantidad de luz que entra al sistema no es una superficie óptica sino un elemento mecánico**.**

En la *Figura 4* se muestra un ejemplo de una "*Aperture stop*" o "Apertura del sistema" en un elemento posterior a una lente, específicamente en la superficie 4 (para más detalle, consulte el Apéndice A.10 `Examp_Doublet_Lens_Pupil.py`). Similarmente, en el lado izquierdo de esta figura se muestran dos haces de rayos a dos ángulos de campo distintos (+2° y -2°). Dichos haces provienen de dos objetos (desde menos infinito) y, por lo tanto, los haces son colimados y atraviesan simultáneamente esta apertura (superficie 4) [4, Sec. 6.2].

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 4*, que los campos tienen distintos orígenes en el plano objeto, por lo tanto, generar uno a uno los rayos puede ser una tarea complicada si se quiere hacer el cálculo manualmente, como se realizó en el Código 3.

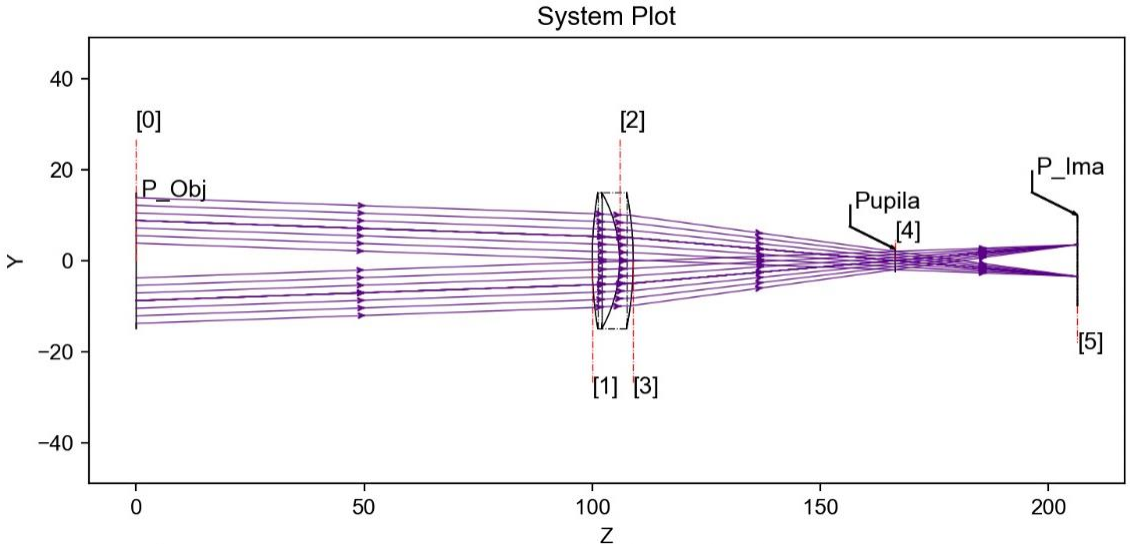


Figura 4: *Visualización de dos haces que coinciden en la posición definida como pupila.*

Por lo anterior, y para facilitar la generación de rayos que atraviesan la pupila, se cuenta con la herramienta `PupilCalc`. Esta es una clase, la cual genera un objeto interactivo, como se muestra en el Código 6:

Código 6

```
W = 0.4
Surf= 4
AperVal = 10
AperType = "EPD"
Pup = Kos.PupilCalc(Doblete, Surf, W, AperType, AperVal)
```

Donde:

- `Doblete` es el sistema óptico que generamos con la clase ***system***
- `Surf` es el número de la superficie que representa la apertura del sistema (superficie 4 en la *Figura 4*).
- `W` es la longitud de onda para la cual se calculará la pupila de salida y entrada.
- `AperType` es un parámetro que puede tener los siguientes dos valores: "`STOP`" o "`EPD`". Al usar el valor "`STOP`" se estará indicando que la superficie definida en **Surf** es la apertura del sistema, mientras que si usamos "`EPD`" se estará definiendo el diámetro de la pupila de entrada.
- `AperVal` es el diámetro de la apertura del sistema o pupila de entrada, dependiendo de cómo se haya configurado `AperType`.

En el Código 6, para el objeto llamado `Pup`, se obtienen los parámetros de las pupilas como se muestra en la Tabla 4:

TABLA 4

Atributos del objeto correspondientes a parámetros de la pupila

`Pup.RadPupInp`	Radio pupila de entrada
`Pup.PosPupInp`	Posición pupila de entrada
`Pup.RadPupOut`	Radio pupila de salida
`Pup.PosPupOut`	Posición pupila de salida
`Pup.PosPupOutFoc`	Posición pupila de salida respecto al plano focal
`Pup.DirPupSal`	Orientación pupila de salida

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

Además de obtener estos parámetros, también es posible generar patrones de rayos en la pupila mediante el cálculo de los cosenos directores y las coordenadas de origen, a partir de la definición de los parámetros.

TABLA 5

Generación de rayos automáticos en base a la pupila

`Pup.Samp=#`	Número entero para el muestreo de rayos en la pupila. El valor por defecto es 5.

`Pup.Ptype="tipo de arreglo"`	Donde "`tipo de arreglo`" puede tomar los siguientes valores:
	
	"`rtheta`" genera un rayo a un ángulo de la pupila unitaria a una posición radial. El radio y el ángulo deben definirse de la siguiente forma:
	
	`Pup.rad=n`
	`Pup.theta=m`
	
	Donde `n` es un número flotante **entre 0 y 1** y theta un el ángulo entre 0 y 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, esto en términos de la altura del objeto y la distancia del plano objeto. Se usa "height" para rayos paralelos que llegan a la pupila y 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 del tipo de campo que se ha elegido en FieldType.

Los atributos mostrados en la Tabla 5 definen el tipo de rayos deseado. Para obtener el arreglo de rayos, lo único que se debe hacer es trasladarlos de la pupila unitaria a la pupila real, obteniendo los cosenos directores y las coordenadas de origen en forma de arreglos. Lo anterior se realiza de la siguiente forma:

```
x, y, z, L, M, N = Pup.Pattern2Field()
```

Donde x, y, z son las coordenadas de origen y L, M, N son los cosenos directores del rayo. De esta forma, es posible iterar entre ellos y realizar el trazo de rayos a través del sistema, como se muestra en el Código 7, donde continuamos trabajando con el sistema óptico Doblete y el contenedor de rayos Rayos.

Código 7

```
for i in range(0, len(x)):
    pSource_0 = [x[i], y[i], z[i]]
    dCos = [L[i], M[i], N[i]]
    Doblete.Trace(pSource_0, dCos, W)
    Rayos.push()
```

Por otro lado, también es posible graficar los puntos de los rayos generados en el plano objeto. Por ejemplo, en la *Figura 5* se ha definido un patrón “hexapolar”.

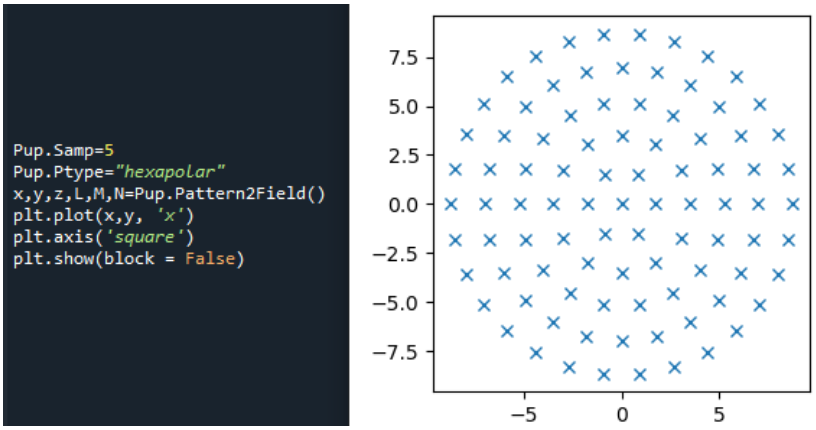


Figura 5: *Código y patrón generado para el origen de los rayos con una distribución hexapolar*

Así mismo, se puede generar el diagrama de manchas (ver *Figura 6*):

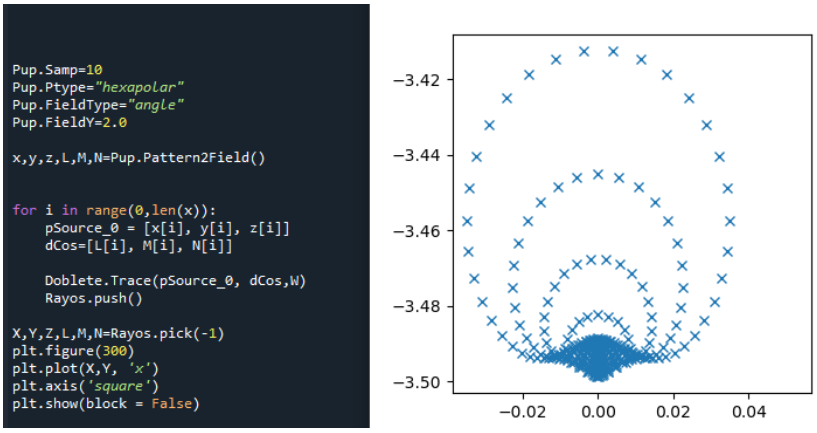


Figura 6: *Código y diagrama de manchas generado con un patrón hexapolar y con un ángulo de campo de 2°*

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.

6.1. REFRACCIÓN ATMOSFÉRICA EN PupilCalc

Si el parámetro `FieldType` (véase Tabla 5) es definido como "`angle`", su uso será apropiado para sistemas ópticos como lo son los telescopios, donde los rayos se consideran provenientes del infinito. Con esta idea en mente, es que se ha incluido la biblioteca de código abierto ***AstroAtmosphere*** [2].

La biblioteca ***AstroAtmosphere*** realiza el cálculo de la desviación de un rayo, dependiendo de la longitud de onda de referencia, la distancia cenital a la que se está observando el objeto (grados) y de los parámetros físicos del observatorio, tales como la temperatura (Kelvin), presión atmosférica (Pa), humedad relativa, cantidad de CO2 en la atmósfera (partes por millón), latitud geográfica (grados), la altura sobre el nivel del mar (metros). La herramienta ***PupilCalc*** utiliza esta biblioteca internamente, para calcular la modificación que los rayos requieren. Esta función se configura como parámetro de ***PupilCalc*** como se muestra en el Código 8:

Código 8

```
Pup.AtmosRef = 1      # 0 to disable, 1 to enable
Pup.T    = 283.15     # Temperature (k)
Pup.P    = 101300     # Atmospheric presure(Pa)
Pup.H    = 0.5        # Humidity (0 to 1)
Pup.xc   = 400        # CO2 (ppm)
Pup.lat  = 31         # Latitude (degrees)
Pup.h    = 2800       # Observatory height (meters)
Pup.l1   = 0.60169    # Reference wavelength(micron)
Pup.l2   = 0.50169    # wavelength of interest(micron)
Pup.z0   = 55.0       # Zenith distance (degrees)
```

A continuación, en el Código 9 se muestra un ejemplo utilizando un telescopio donde se generan 3 grupos de rayos, únicamente cambiando la longitud de onda entre ellos:

Código 9

```
W1 = 0.50169
Pup.l2   = W1
xa,ya,za,La,Ma,Na=Pup.Pattern2Field()

W2 = 0.60169
Pup.l2   = W2
xb,yb,zb,Lb,Mb,Nb=Pup.Pattern2Field()

W3 = 0.70169
Pup.l2   = W3
xc,yc,zc,Lc,Mc,Nc=Pup.Pattern2Field()
```

En el Código 10 se realiza entonces el trazado de rayos para los tres grupos y se almacenan en diferentes contenedores.

Código 10

```
for i in range(0,len(xa)):
    pSource_0 = [xa[i], ya[i], za[i]]
    dCos=[La[i], Ma[i], Na[i]]
    Telescopio.Trace(pSource_0, dCos, W1)
    Rayos1.push()

for i in range(0,len(xb)):
    pSource_0 = [xb[i], yb[i], zb[i]]
    dCos=[Lb[i], Mb[i], Nb[i]]
    Telescopio.Trace(pSource_0, dCos, W2)
    Rayos2.push()

for i in range(0,len(xc)):
    pSource_0 = [xc[i], yc[i], zc[i]]
    dCos=[Lc[i], Mc[i], Nc[i]]
    Telescopio.Trace(pSource_0, dCos, W3)
    Rayos3.push()
```

Como resultado de realizar los diagramas de manchas obtenemos el gráfico de la *Figura 7*, generado con el Código 11, donde se puede ver una separación de aproximadamente 72 µm entre las longitudes de onda extremas (cruces azul y verde en la *Figura 7*).

Código 11

```
X,Y,Z,L,M,N=Rayos1.pick(-1)
plt.plot(X*1000.0,Y*1000.0, 'x', c="b")

X,Y,Z,L,M,N=Rayos2.pick(-1)
plt.plot(X*1000.0,Y*1000.0, 'x', c="r")

X,Y,Z,L,M,N=Rayos3.pick(-1)
plt.plot(X*1000.0,Y*1000.0, 'x', c="g")
```

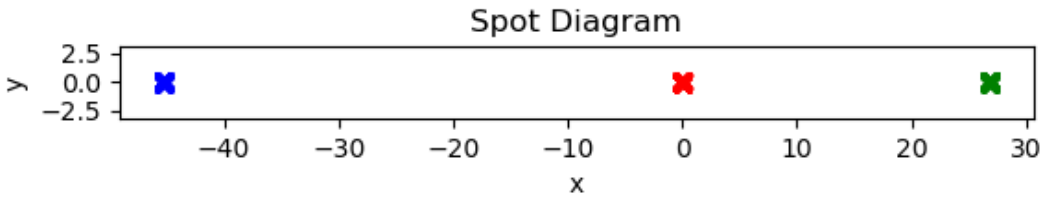


Figura 7: *Imágenes formadas por un telescopio las cuales son desplazadas espectralmente por acción de la refracción atmosférica.*

6.2. COMPLEMENTOS DE PupilCalc

Para mayor comprensión de las implementaciones mostradas en esta sección, estudiar el algoritmo mostrado en la Sección 7.

Una vez que se ha realizado el cálculo de la posición de la pupila, podemos realizar, incluyendo rayos a través del sistema, algunos otros cálculos, como un mapa de fase. Para eso tenemos la implementación "`Phase`", que tiene la siguiente forma:

```
X, Y, Z, P2V = Kos.Phase(Pupila)
```

El parámetro de entrada es directamente el objeto Pupila, tal y como esté configurado; esto se verá a mayor detalle en el algoritmo de la Sección 7.

Otra implementación es ***"Zernike_Fitting"***, que recibe como parámetros los valores que arroja la función "`Phase`" como se muestra a continuación:

```
Zcoef, Mat, RMS2Chief, RMS2Centroid, FITTINGERROR = Kos.Zernike_Fitting(X, Y, Z, A)
```

Donde `X` y `Y` son coordenadas en la pupila de salida, `Z` es la fase en esos puntos, y `A` es un arreglo del tipo *numpy* creado de esta forma:

```
A = np.ones(NC)
```

`NC` es el número de coeficientes que deseamos utilizar en el ajuste. Si un número del arreglo es cero, entonces ese término no se utiliza para el ajuste.

La implementación ***"Zernike_Fitting"*** tiene como salida una lista de los coeficientes en longitudes de onda, una lista con las expresiones matemáticas para ayudar en el despliegue y comprensión de las aberraciones asociadas. El valor `RMS2Chief` es el valor RMS de los coeficientes sin el pistón, mientras que el `RMS2Centroid` sin pistón ni tilts según la nomenclatura utilizada por *Zemax*.

Finalmente, la implementación "`psf`" utiliza los coeficientes de los polinomios de Zernike para crear una imagen "`I`" en un arreglo *numpy*, con la PSF (*Point Spread Function*) con la integral de difracción de Fraunhofer por medio de la transformada rápida de Fourier.

```
I = Kos.psf(Zcoef, Focal, Diameter, Wave, pixels=265, plot=1, sqr = 0)
```

Si el valor de `plot` es igual a 1, se realiza la gráfica, `pixels` es el número de pixeles por lado, `Diameter` es el diámetro de la pupila de salida y `Focal` es la distancia focal del sistema.

Si el valor de `sqr =1`, le saca la raíz cuadrada a la `I` para aumentar el contraste en el despliegue, únicamente a la imagen graficada, no al valor de `I` entregado.

7. PRECISIÓN

En la presente sección se muestra la definición en *KrakenOS* de un diseño correspondiente al telescopio de 2.1 m del OAN-SPM con *f/7.5*. Con este se utilizan una serie de herramientas con las cuales queremos mostrar que los resultados obtenidos con esta biblioteca son confiables; en este caso particular se está desalineando el espejo secundario 1.0 mm, además se está utilizando un campo de 0.1°.

A continuación, se muestra el código completo para la simulación del telescopio y se comentan las salidas en donde se indica la similitud con los datos obtenidos con el software *Zemax*, el cual es un estándar en la industria.

```
import os
import sys
import matplotlib.pyplot as plt
import numpy as np
import pkg_resources

""" If KrakenOS is installed, if not, it assumes that
an folder downloaded from github is run"""

required = {'KrakenOS'}
installed = {pkg.key for pkg in pkg_resources.working_set}
missing = required - installed

if missing:
    print("Not installed")
    import sys
    sys.path.append("../..")

import KrakenOS as Kos
currentDirectory = os.getcwd()
sys.path.insert(1, currentDirectory + '/library')

# Plano objeto #
P_Obj = Kos.surf()
P_Obj.Rc = 0
P_Obj.Thickness = 1000*0 + 3452.2
P_Obj.Glass = "AIR"
P_Obj.Diameter = 1059. * 2.0
P_Obj.Drawing=0

# Espejo primario #
```

```
Thickness = 3452.2
M1 = Kos.surf()
M1.Rc = -9638.0
M1.Thickness = -Thickness
M1.k = -1.07731
M1.Glass = "MIRROR"
M1.Diameter = 1.059E+003 * 2.0
M1.InDiameter = 250 * 2.0
M1.TiltY = 0.0
M1.TiltX = 0.0
M1.AxisMove = 0

# Espejo secundario #
M2 = Kos.surf()
M2.Rc = -3930.0
M2.Thickness = Thickness + 1037.525880
M2.k = -4.3281
M2.Glass = "MIRROR"
M2.Diameter = 336.5 * 2.0
M2.TiltY = 0.0
M2.TiltX = 0.0
M2.DespY = 0.0
""" Se desplaza el secundario """
M2.DespX = 1.0
M2.AxisMove = 0 #

# Plano imagen #
P_Ima = Kos.surf()
P_Ima.Diameter = 300.0
P_Ima.Glass = "AIR"
P_Ima.Name = "Plano imagen"

# _____________________________________#

A = [P_Obj, M1, M2, P_Ima]
configuracion_1 = Kos.Setup()
Telescopio = Kos.system(A, configuracion_1)

# _____________________________________#
"""Vamos a definir los parámetros de la pupila del sistema,
definiremos esta pupila en la superficie 1, esta corresponde al
espejo primario"""
Surf = 1

"""Definimos la longitud de onda en micras"""
W = 0.50169

```

```
""" Indicamos que la apertura del sistema definirá la pupila"""
AperType = "EPD"

""" Definimos el Diametro de la apertura del sistema"""
AperVal = 2000.
Pupil = Kos.PupilCalc(Telescopio, Surf, W, AperType, AperVal)

""" la pupila tendrá un arreglo exapolar con 11 anillos"""

Pupil.Samp = 11
Pupil.Ptype = "hexapolar"
"""Indicamos que los campos son tel tipo angulo, como en el caso de los telescopios
    con luz desde el infinito, para diseños con objeto certano este parametro es la
    Altura del objeto"""

Pupil.FieldType = "angle"
""" Definimos que el campo es 0 en x y cero en y """

Pupil.FieldX = 0.1
Pupil.FieldY = 0.0

""" Calculamos la fase del frente de onda en la pupila, las coordenadas X, Y
son las coordendas en la pupila, el valor de Z es la fase en cada punto X, Y
y P2V es el valor pico a valle."""

X, Y, Z, P2V = Kos.Phase(Pupil)
print("Peak to valley: ", P2V)

"""Indicamos el grado de expanción para los polinomios de Zernike"""
NC = 35

"""Generamos un arreglo Numpy con las mismas dimensiones de la expanción
Definida para los coeficientes. """
A = np.ones(NC)

""" Calculamos los polinomios de Zernike con la fase calculada y el
numero de elementos deseados en la expanción, Zcoef son los coeficientes
en longitudes de onda, Mat es la expreción matematica de Zeidel para dicho
coeficiente, esto con fines ilustrativos, w_rms es el error del ajuste"""

Zcoef, Mat, RMS2Chief, RMS2Centroid, FittEr = Kos.Zernike_Fitting(X, Y, Z, A)

"""Se despliegan los resultados"""
for i in range(0, NC):
    print("z", i + 1, "  ", "{0:.8f}".format(float(Zcoef[i])), ":", Mat[i])

print("(RMS) Fitting error: ", FittEr)
```

```
126.print(RMS2Chief, "RMS(to chief) From fitted coefficents")
```

A continuación, en la Tabla 6, se muestra el listado de los coeficientes de los polinomios de Zernike que se han ajustado al frente de onda que sale del telescopio. Se puede ver que hay una gran similitud entre los obtenidos con *Zemax* y con *KrakenOS*, con una diferencia de 0.0012 longitudes de onda en el ajuste con el centroide, esto da un buena de confiabilidad en los resultados.

TABLA 6

Polinomios de Zernike (Nomenclatura de Noll) ajustados al frente de onda con Zemax y KrakenOS (OPD referenciado al rayo principal)

Calculados con KrakenOS From fitted coefficents RMS(to chief): 1.5862914 Waves RMS(to centroid): 0.5808909776waves RMS Fit error: 6.32445818087e-07		Calculados con Zemax From integration of the fitted coefficients: RMS (to chief): 1.58742701 waves RMS (to centroid): 0.58216212 waves RMS fit error: 0.00000058 waves
KrakenOS	Zemax	Termino polinomio
0.39835347	0.40380980	1
-1.47610507	-1.47682489	4^(1/2) (p) * COS (A)
0.00000000	0.00000000	4^(1/2) (p) * SIN (A)
0.23324038	0.23639165	3^(1/2) (2p^2 - 1)
0.00000000	0.00000000	6^(1/2) (p^2) * SIN (2A)
0.11084497	0.11084486	6^(1/2) (p^2) * COS (2A)
0.00000000	0.00000000	8^(1/2) (3p^3 - 2p) * SIN (A)
-0.52032843	-0.52032699	8^(1/2) (3p^3 - 2p) * COS (A)
0.00000000	0.00000000	8^(1/2) (p^3) * SIN (3A)
0.00000914	0.00000914	8^(1/2) (p^3) * COS (3A)
0.00193793	0.00193864	5^(1/2) (6p^4 - 6p^2 + 1)
-0.00013629	-0.00013629	10^(1/2) (4p^4 - 3p^2) * COS (2A)
0.00000000	0.00000000	10^(1/2) (4p^4 - 3p^2) * SIN (2A)
0.00000000	-0.00000005	10^(1/2) (p^4) * COS (4A)
0.00000000	0.00000000	10^(1/2) (p^4) * SIN (4A)
0.00097859	0.00097858	12^(1/2) (10p^5 - 12p^3 + 3p) * COS (A)
0.00000000	0.00000000	12^(1/2) (10p^5 - 12p^3 + 3p) * SIN (A)
-0.00000000	0.00000000	12^(1/2) (5p^5 - 4p^3) * COS (3A)
0.00000000	0.00000000	12^(1/2) (5p^5 - 4p^3) * SIN (3A)
0.00000000	0.00000000	12^(1/2) (p^5) * COS (5A)
0.00000000	0.00000000	12^(1/2) (p^5) * SIN (5A)
-0.00048962	-0.00048980	7^(1/2) (20p^6 - 30p^4 + 12p^2 - 1)
0.00000000	0.00000000	14^(1/2) (15p^6 - 20p^4 + 6p^2) * SIN (2A)
-0.00000014	-0.00000014	14^(1/2) (15p^6 - 20p^4 + 6p^2) * COS (2A)
0.00000000	0.00000000	14^(1/2) (6p^6 - 5p^4) * SIN (4A)
0.00000000	-0.00000005	14^(1/2) (6p^6 - 5p^4) * COS (4A)
0.00000000	0.00000000	14^(1/2) (p^6) * SIN (6A)
0.00000000	0.00000000	14^(1/2) (p^6) * COS (6A)
0.00000000	0.00000000	16^(1/2) (35p^7 - 60p^5 + 30p^3 - 4p) * SIN (A)
-0.00000411	-0.00000412	16^(1/2) (35p^7 - 60p^5 + 30p^3 - 4p) * COS (A)
0.00000000	0.00000000	16^(1/2) (21p^7 - 30p^5 + 10p^3) * SIN (3A)
0.00000000	0.00000000	16^(1/2) (21p^7 - 30p^5 + 10p^3) * COS (3A)
0.00000000	0.00000000	16^(1/2) (7p^7 - 6p^5) * SIN (5A)
0.00000000	0.00000000	16^(1/2) (7p^7 - 6p^5) * COS (5A)
0.00000000	0.00000000	16^(1/2) (p^7) * SIN (7A)

```
127.print(RMS2Centroid, "RMS(to centroid) From fitted coefficents")
128.COEF = Zcoef
129.Focal = Pupil.EFFL
130.Diameter = 2.0 * Pupil.RadPupInp
131.Wave = W
132.I= Kos.psf(COEF, Focal, Diameter, Wave, pixels=265, plot=1)
```

A continuación, en la *Figura 8*, se muestra la figura de la PSF obtenida con *Zemax* y con *KrakenOS* en donde se puede ver la gran similitud entre los diagramas obtenidos.

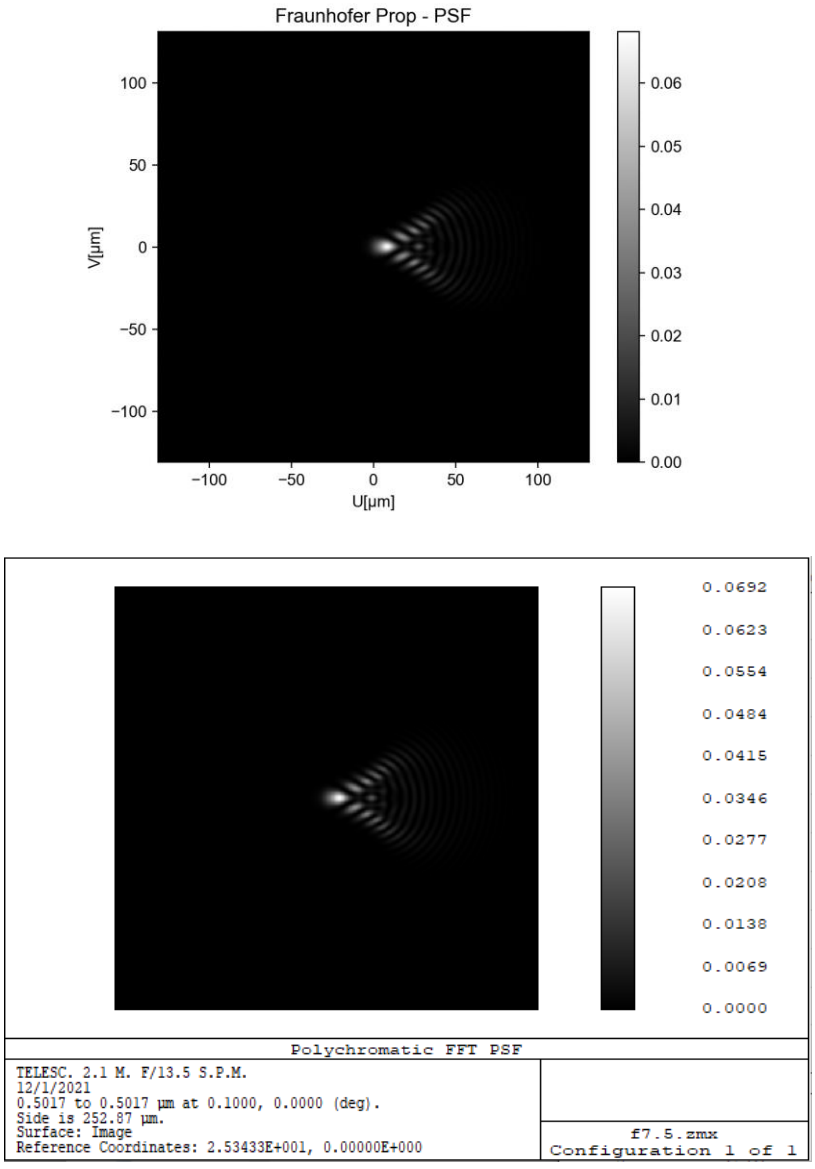


Figura 8: *PSF calculada con el frente de onda.*
Superior: (KrakenOS), Inferior: (Zemax).

```
133."""Se genera un contenedor de rayos"""
134.
```

```
RR = Kos.raykeeper(Telescopio)

""" Se generan rayos que pasan por la pupila """
x, y, z, L, M, N = Pupil.Pattern2Field()

# ____________________________________#

""" Se trazan esos rayos y se almacenan"""

for i in range(0, len(x)):
    pSource_0 = [x[i], y[i], z[i]]
    dCos = [L[i], M[i], N[i]]
    Telescopio.Trace(pSource_0, dCos, W)
    RR.push()

# ____________________________________#

""" Se grafica el telescopio con los rayos almacenados"""
Kos.display3d(Telescopio, RR, 1 )
```

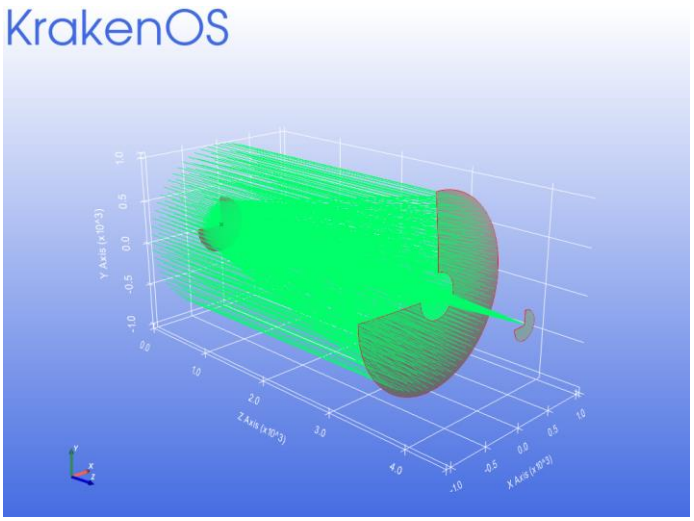


Figura 9: *Despliegue tridimensional del Sistema óptico con rayos.*

```
X, Y, Z, L, M, N = RR.pick(-1)
""" Se grafica el diagrama de manchas """
plt.figure(2)
plt.plot(X, Y, 'x')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('spot Diagram')
plt.axis('square')
plt.show()
```

Se realizó el diagrama de manchas de la imagen con aberración de coma por motivo de la desalineación, se puede ver en las imágenes de la *Figura 10* que el valor del radio RMS es prácticamente el mismo.

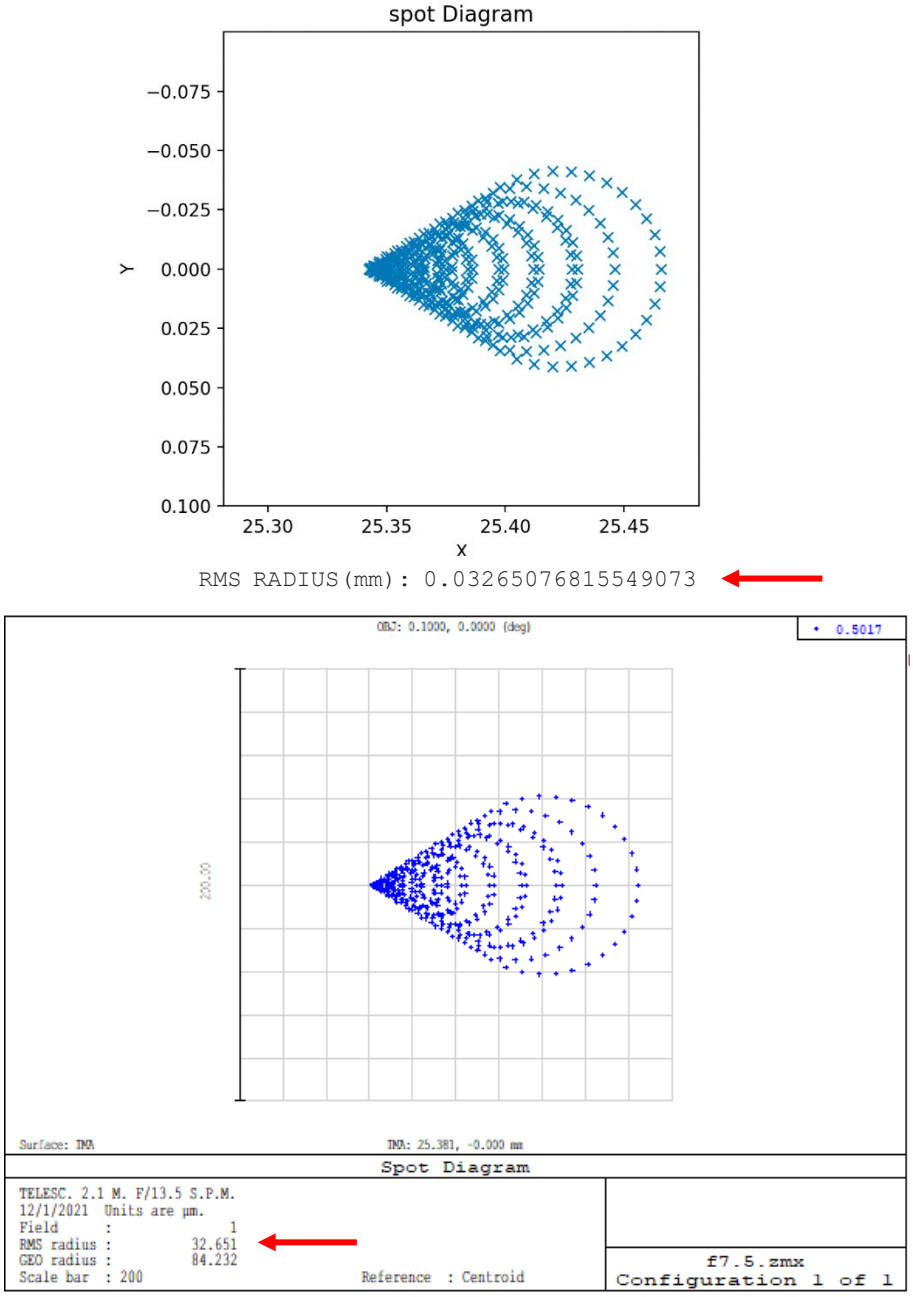


Figura 10: *Diagrama de manchas con un patron hexapolar. Superior (KrakenOS), inferior (Zemax)*

```
163.X = X - np.mean(X)
164.Y = Y - np.mean(Y)
165.
166.R=np.sqrt(X**2 + Y**2)
167.RMS = np.sqrt(np.mean(R**2))
```

```
168.print("Radio RMS: ", RMS)
169.
170."""  Se prepara una imagen con los coeficientes de 400x400"""
171.ima = Kos.WavefrontData2Image(Zcoef, 400)
172.
173.print("Peak 2 valley. ", np.max(ima)-np.min(ima))
174.
175."""  Se grafica el interferograma """
176.Type = "interferogram"
177. Kos.ZernikeDataImage2Plot(ima, Type)
```

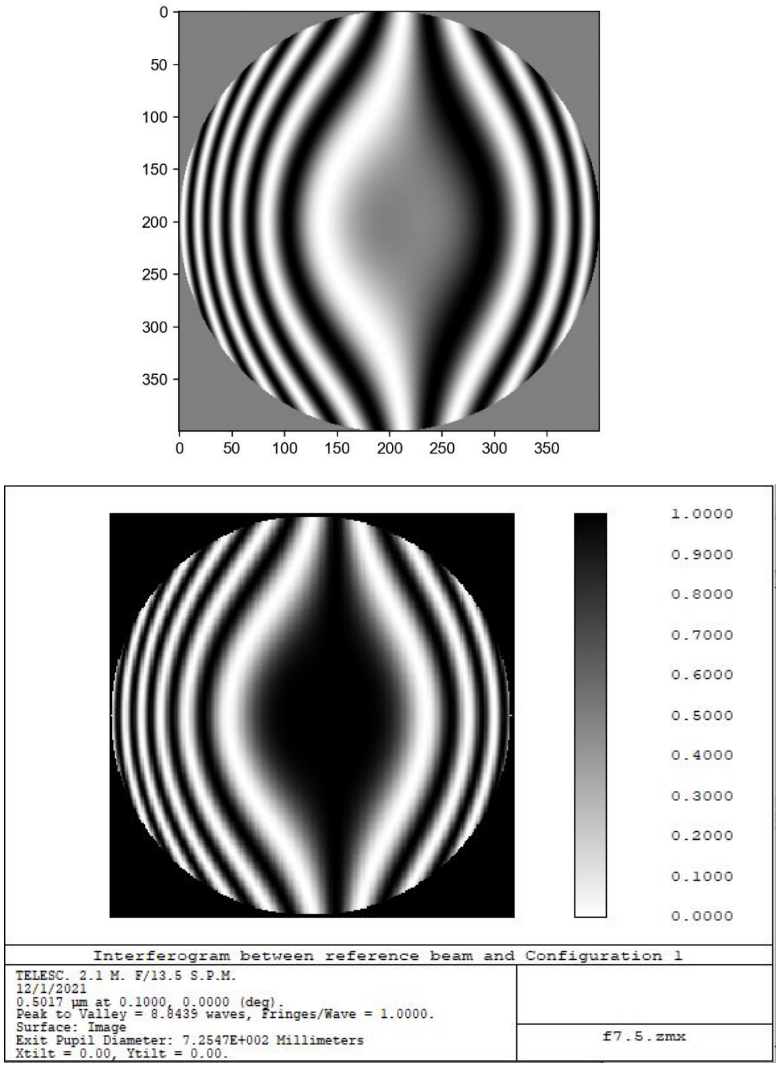


Figura 11: *Interferograma con el frente de onda obtenido. Superior (KrakenOS), inferior (Zemax).*

8. MANEJO DEL VISOR 3D

Al generar un gráfico tridimensional es posible, además, realizar las siguientes acciones para una mejor visualización:

- El visor se apodera de la ejecución en *Python*, así que `Display3D` debe de ser la última instrucción de un *script*. Si coloca `Display3D` antes de otro cálculo, el núcleo *Python* quedará en pausa hasta que la ventana del visor 3D sea cerrada con el botón de cerrar.
- Rotar la simulación en cualquier dirección. Para ello, mover el ratón mientras se mantiene oprimido el botón izquierdo del mismo.
- Alejar o acercar. Para ello, mover el ratón hacia arriba y abajo, mientras se mantiene oprimido el botón derecho del mismo.
- Arrastrar la simulación. Para ello, mover el ratón, mientras se oprime simultáneamente el botón izquierdo del mismo y la tecla [SHIFT].

9. REFERENCIAS

[1] https://docs.pyvista.org/examples/00-load/create-geometric-objects.html#sphx-glr-examples-00-load-create-geometric-objects-py

[2] Noll, R.
"Zernike polynomials and atmospheric turbulence"
J. Opt. Soc. Am. 66, 207-211
1976.

[3] van den Born & Jellema
"Quantification of the expected residual dispersion of the MICADO Near-IR imaging instrument"
MNRAS, DOI: 10.1093/mnras/staa1870.

[4] Malacara-Hernández, D., Malacara-Hernández, Z., Malacara, Z.
Handbook of Optical Design
Edición 2, CRC Press, 2003, ISBN: 0203912942, 9780203912942

[5] Warren J. Smith
Modern Optical Engineering. The Design of Optical Systems.
Third Edition, McGraw-Hill, 2000. ISBN 0-07-136360-2

APÉNDICE A. EJEMPLOS

Los autores de este manual consideramos que la mejor manera de comprender el uso de la biblioteca *KrakenOS* es practicar con los ejemplos incluidos dentro de la misma. A continuación, se enlistan los nombres de los archivos de cada ejemplo disponible, seguido del código para cada uno.

Examp-Axicon.py
Examp-Axicon_And_Cylinder.py
Examp-Diffraction_Grating_Reflection.py
Examp-Diffraction_Grating_Transmission.py
Examp-Doublet_Lens-ParaxMatrix.py
Examp-Doublet_Lens.py
Examp-Doublet_Lens_3Dcolor.py
Examp-Doublet_Lens_CommandsSystem.py
Examp-Doublet_Lens_Cylinder.py
Examp-Doublet_Lens_NonSec.py
Examp-Doublet_Lens_Pupil.py
Examp-Doublet_Lens_Pupil_Seidel.py
Examp-Doublet_Lens_Tilt-Nulls.py
Examp-Doublet_Lens_Tilt.py
Examp-Doublet_Lens_Tilt_non_sec.py
Examp-Doublet_Lens_Zernike.py
Examp-ExtraShape_Micro_Lens_Array.py
Examp-ExtraShape_Radial_Sine.py
Examp-ExtraShape_XY_Cosines.py
Examp-Flat_Mirror_45Deg.py
Examp-MultiCore.py
Examp-ParabolicMirror_Shift.py
Examp-Perfect_lens.py
Examp-Ray.py
Examp-Solid_Object_STL.py
Examp-Solid_Object_STL_ARRAY.py
Examp-Source_Distribution_Function.py
Examp-Tel_2M.py
Examp-Tel_2M_Error_Map.py
Examp-Tel_2M_Pupila.py
Examp-Tel_2M_Spider_Spot_Diagram.py
Examp-Tel_2M_Spider_Spot_RMS.py
Examp-Tel_2M_Spider_Spot_Tilt_M2.py
Examp-Tel_2M_Atmospheric_Refraction.py
Examp-Tel_2M_Wavefront_Fitting.py

APÉNDICE A.1 EJEMPLO-RAY

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Ray"""
import numpy as np
import KrakenOS as Kos


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 0.1
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

P_Obj2 = Kos.surf()
P_Obj2.Rc = 0.0
P_Obj2.Thickness = 10
P_Obj2.Glass = "AIR"
P_Obj2.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 18.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, P_Obj2, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

pSource_0 = [0, 14, 0]
tet = 0.1
dCos = [0.0, np.sin(np.deg2rad(tet)), -np.cos(np.deg2rad(tet))]
W = 0.4
Doblete.Trace(pSource_0, dCos, W)
Rayos.push()

Kos.display3d(Doblete, Rayos, 2)
```

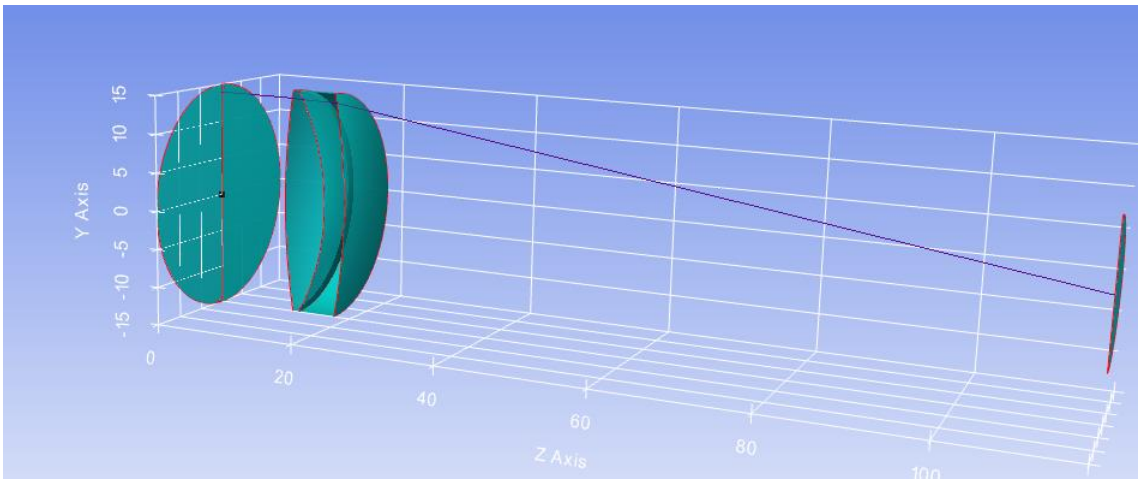


Figura A.1: *Ejemplo del trazo de un único rayo a través de un sistema óptico*

APÉNDICE A.2 EJEMPLO-PERFECT LENS

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Perfect Lens"""
import time
import matplotlib.pyplot as plt
import numpy as np
import KrakenOS as Kos

start_time = time.time()

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 50
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Thin_Lens = 100.
L1a.Thickness = (100 + 50)
L1a.Rc = 0.0
L1a.Glass = "AIR"
L1a.Diameter = 30.0

L1b = Kos.surf()
L1b.Thin_Lens = 50.
L1b.Thickness = 100.
L1b.Rc = 0.0
L1b.Glass = "AIR"
L1b.Diameter = 30.0

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 100.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, P_Ima]
config_1 = Kos.Setup()

Doblete = Kos.system(A, config_1)
Rayos1 = Kos.raykeeper(Doblete)
Rayos2 = Kos.raykeeper(Doblete)
Rayos3 = Kos.raykeeper(Doblete)
RayosT = Kos.raykeeper(Doblete)

tam = 10
rad = 10.0
tsis = len(A) - 1
for j in range(-tam, tam + 1):
    for i in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos1.push()
```

```
            RayosT.push()

Kos.display3d(Doblete, RayosT, 0)
X, Y, Z, L, M, N = Rayos1.pick(-1)

plt.plot(X, Y, 'x')
plt.xlabel('numbers')
plt.ylabel('values')
plt.title('Stop Diagram')
plt.axis('square')
plt.show()
print("--- %s seconds ---" % (time.time() - start_time))
```

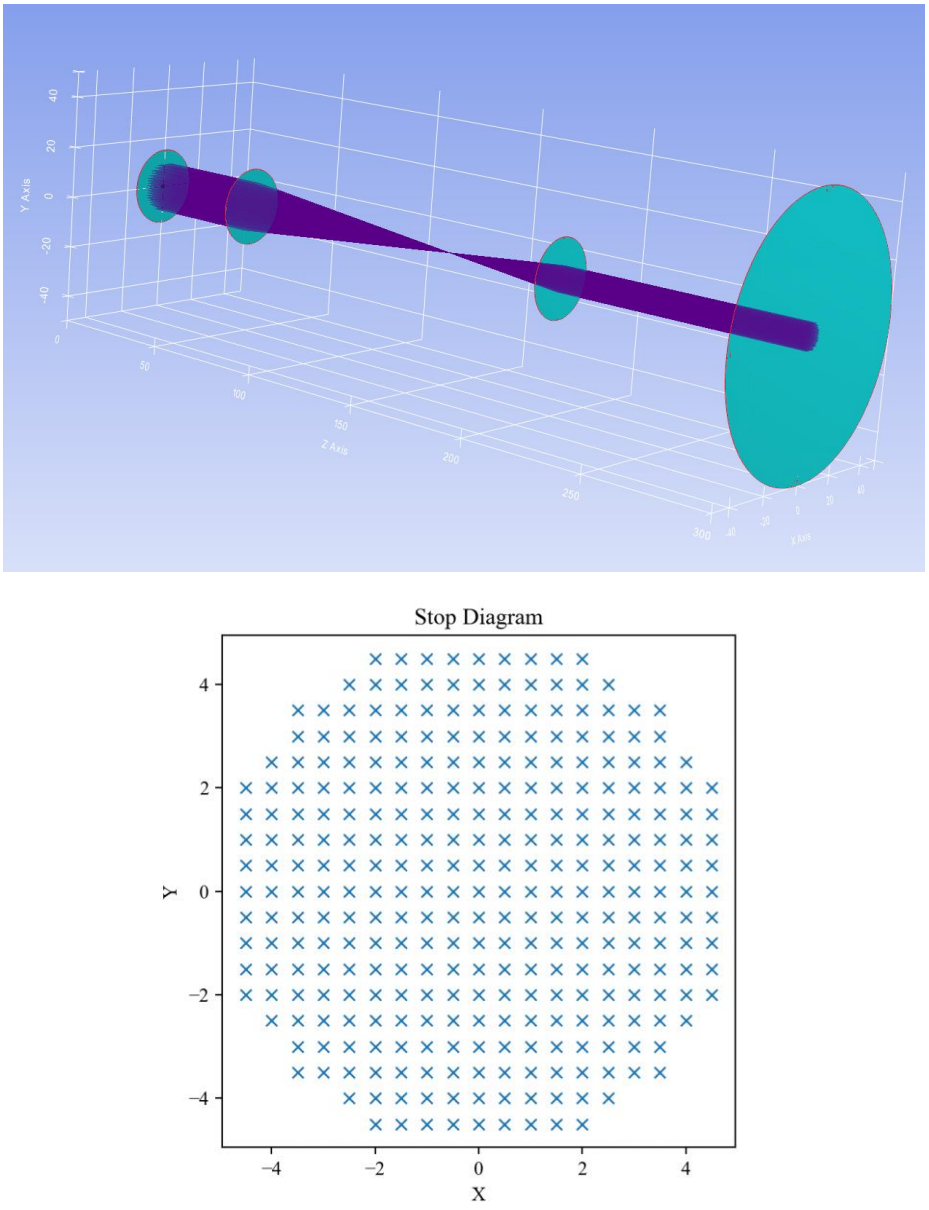


Figura A.2: *Ejemplo de lentes perfectas que no presentan aberraciones.*

APÉNDICE A.3 EJEMPLO-DOUBLET LENS 3D COLOR

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens 3D color"""
import numpy as np
import KrakenOS as Kos


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0
L1a.Color = [.8, .7, .4]

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30
L1b.Color = [.7, .4, .4]

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 100.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 10.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
```

```
            Rayos.push()
            W = 0.5
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.6
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display3d(Doblete, Rayos, 1)
```

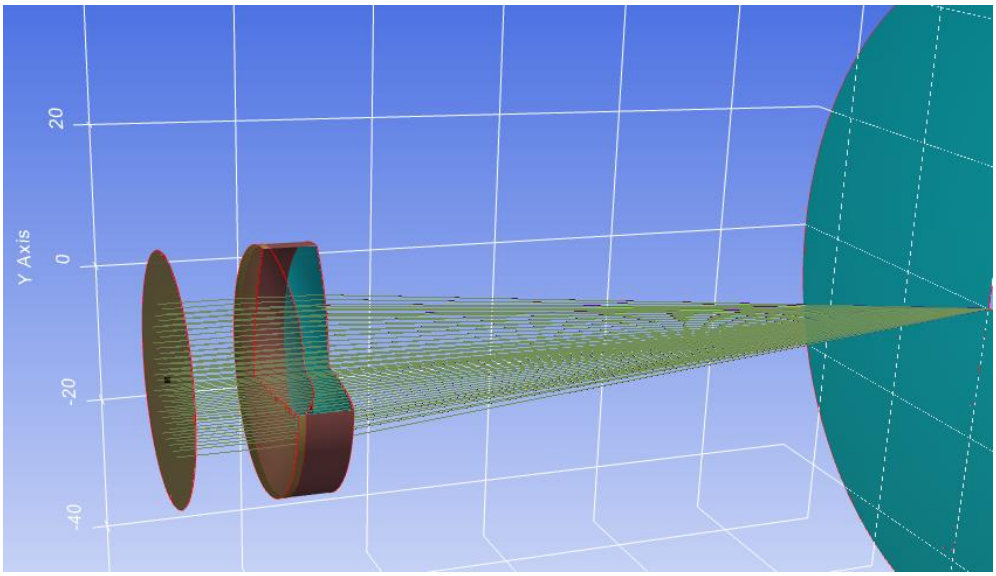


Figura A.3: *Vista de un doblete donde se ha definido un cambio en el color de la superficie.*

APÉNDICE A.4 EJEMPLO-DOUBLET LENS TILT

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens Tilt"""
import numpy as np
import KrakenOS as Kos


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.AxisMove = 1
L1a.TiltX = 1
L1a.DespY = 10.

L1b = Kos.surf()
L1b.Rc = (-3.071608670000159E+001)
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = (-7.819730726078505E+001)
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 100.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, P_Obj, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 10.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
```

```
            Rayos.push()
            W = 0.5
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.6
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display3d(Doblete, Rayos, 2)
Kos.display2d(Doblete, Rayos, 0)# !/usr/bin/env python3
```

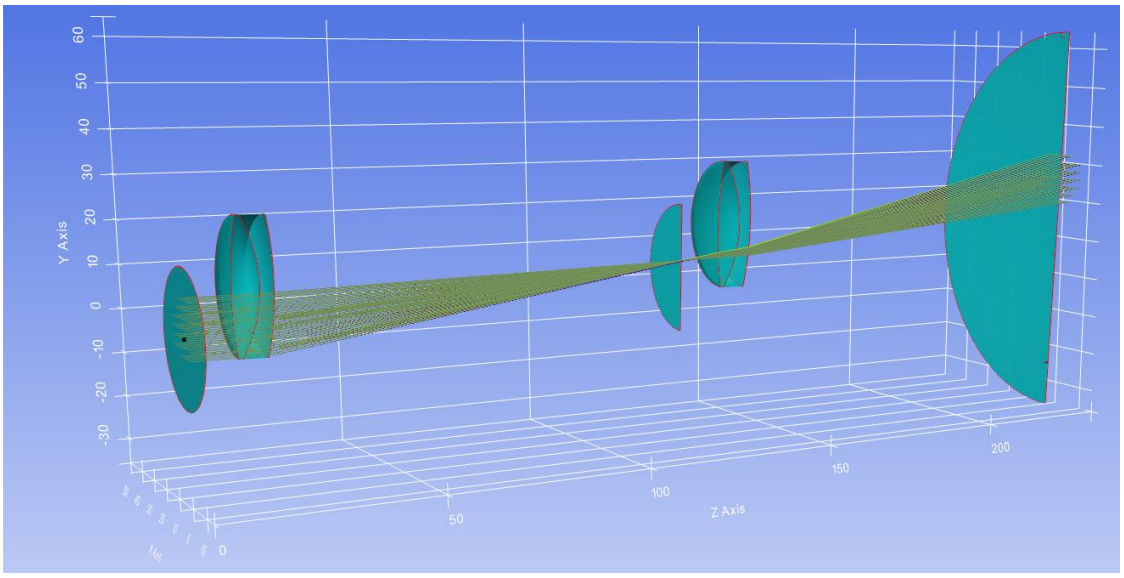


Figura A.4: *Visualización 3D de un sistema fuera de eje generado con repetición de superficies. La modificación de una superficie afecta a todas las partes del diseño en donde esta sea utilizada.*

APÉNDICE A.5 EJEMPLO-DOUBLET LENS (CÁLCULOS PARAXIALES)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens Para Matrix"""
import time
import KrakenOS as Kos

start_time = time.time()


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 3.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, P_Ima]
config_1 = Kos.Setup()

Doblete = Kos.system(A, config_1)
print("Calculando para cierta wave --------")
Prx = Doblete.Parax(0.4)
SistemMatrix, S_Matrix, N_Matrix, a, b, c, d, EFFL, PPA, PPP, CC, N_Prec, DD = Prx
print(EFFL)

L1a.Rc = L1a.Rc + 1
Doblete.ResetData()
print("Calculando para cierta wave --------")
Prx = Doblete.Parax(0.4)
SistemMatrix, S_Matrix, N_Matrix, a, b, c, d, EFFL, PPA, PPP, CC, N_Prec, DD = Prx
print(EFFL)

L1a.Rc = L1a.Rc + 1
Doblete.ResetData()
print("Calculando para cierta wave --------")
Prx = Doblete.Parax(0.4)
```

```
SistemMatrix, S_Matrix, N_Matrix, a, b, c, d, EFFL, PPA, PPP, CC, N_Prec, DD = Prx
print(EFFL)

L1a.Rc = L1a.Rc + 1
Doblete.ResetData()
print("Calculando para cierta wave --------")
Prx = Doblete.Parax(0.4)
SistemMatrix, S_Matrix, N_Matrix, a, b, c, d, EFFL, PPA, PPP, CC, N_Prec, DD = Prx
print(EFFL)
```

```
100.18502274454893
100.78009773058245
101.3695118403712
101.9533454684305
```

Figura A.5: *Distancias focales resultantes para un incremento repetido de 1mm entre cálculos.*

APÉNDICE A.6 EJEMPLO-DOUBLET LENS TILT NULLS

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens Tilt Nulls"""
import numpy as np
import KrakenOS as Kos


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 1
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 5.513435044607768E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.TiltX = 4

Null1_L1a = Kos.surf()
Null1_L1a.Thickness = -L1a.Thickness
Null1_L1a.Glass = "NULL"
Null1_L1a.Diameter = L1a.Diameter
Null1_L1a.TiltX = -L1a.TiltX
Null1_L1a.Order = 1

Null2_L1a = Kos.surf()
Null2_L1a.Thickness = L1a.Thickness
Null2_L1a.Glass = "NULL"
Null2_L1a.Diameter = L1a.Diameter

L1b = Kos.surf()
L1b.Rc = -4.408716526030626E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -2.246906271406796E+002
L1c.Thickness = 9.737871661422000E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 100.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, Null1_L1a, Null2_L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 10.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
```

```
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.5
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.6
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display2d(Doblete, Rayos, 0)
```

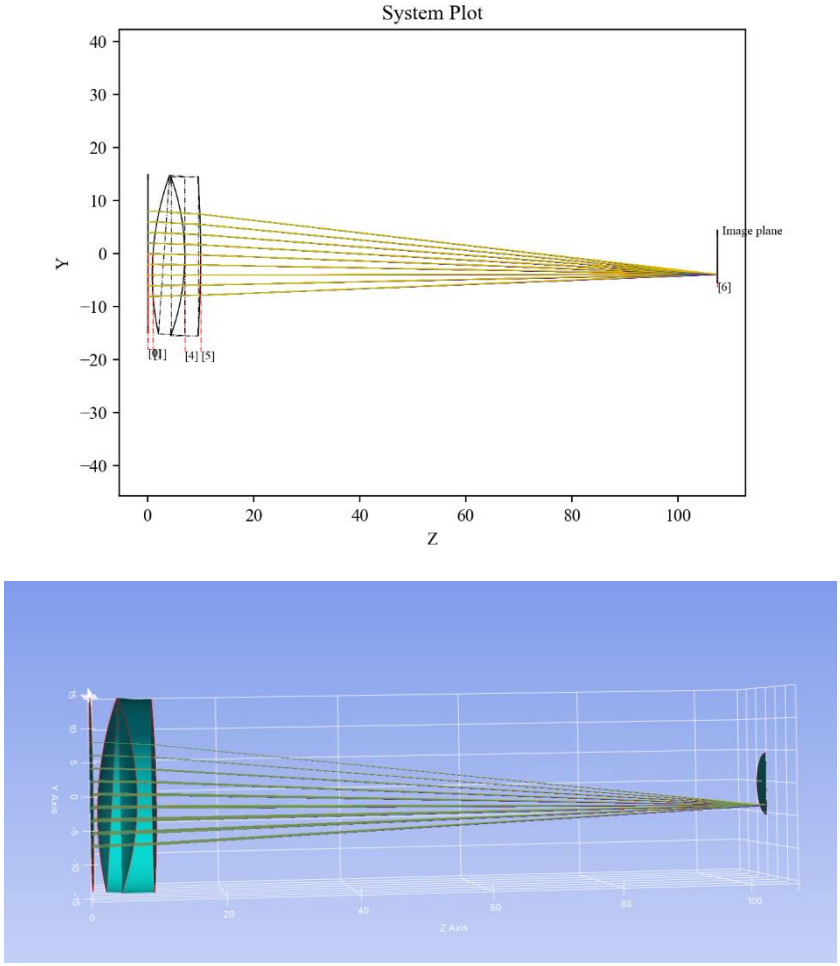


Figura A.6: *Vista 2D y 3D de un doblete con una cara inclinada.*

APÉNDICE A.7 EJEMPLO-DOUBLET LENS NONSEC

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens NonSec"""
import numpy as np
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 0
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

P_Obj2 = Kos.surf()
P_Obj2.Rc = 0.0
P_Obj2.Thickness = 10
P_Obj2.Glass = "AIR"
P_Obj2.Diameter = 100.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "MIRROR"
P_Ima.Diameter = 30.0
P_Ima.Name = "Plano imagen"
P_Ima.DespZ = 10
P_Ima.TiltX = 6.

A = [P_Obj, P_Obj2, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 10
rad = 14.0
tsis = len(A) - 1
for nsc in range(0, 100):
    for j in range(-tam, tam + 1):
        x_0 = (0 / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
```

```
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.NsTrace(pSource_0, dCos, W)
            Rayos.push()

Kos.display2d(Doblete, Rayos, 0)
```

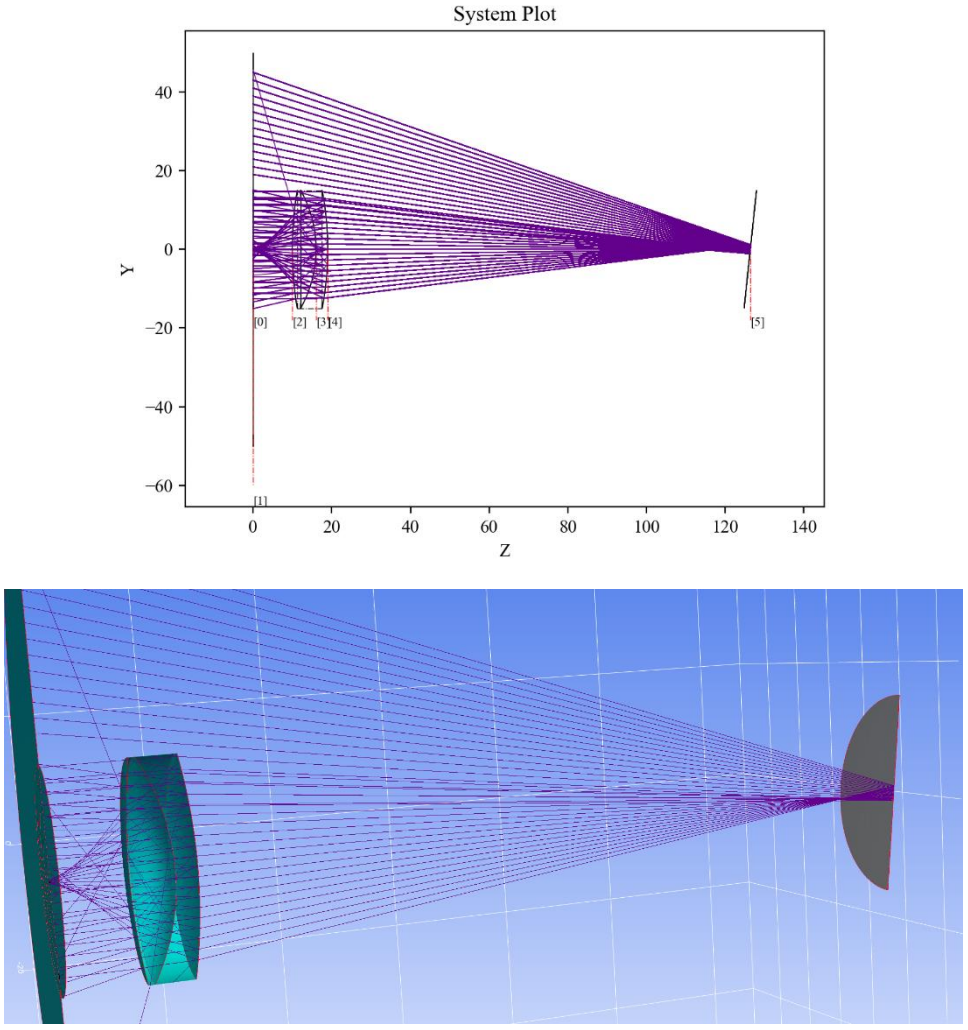


Figura A7: *Visualización 2D y 3D de un trazado de rayos no secuencial, algunos rayos son reflejados de regreso dependiendo del valor en los coeficientes de Fresnel, estos dependen de la longitud de onda, el material y el ángulo del rayo con respecto a la normal de la superficie en el punto de intersección.*

APÉNDICE A.8 EJEMPLO-DOUBLET LENS ZERNIKE

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Doublet Lens Zernike"""
import KrakenOS as Kos
import numpy as np
import matplotlib.pyplot as plt
import time

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 1
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 5.513435044607768E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0

L1b = Kos.surf()
L1b.Rc = -4.408716526030626E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -2.246906271406796E+002
L1c.Thickness = 9.737871661422000E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

Z = np.zeros(36)
Z[8] = 0.5
Z[9] = 0.5
Z[10] = 0.5
Z[11] = 0.5
Z[12] = 0.5
Z[13] = 0.5
Z[14] = 0.5
Z[15] = 0.5
L1c.ZNK = Z

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 100.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 12.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
```

```
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display3d(Doblete, Rayos, 2)
Kos.display2d(Doblete, Rayos, 0)
```

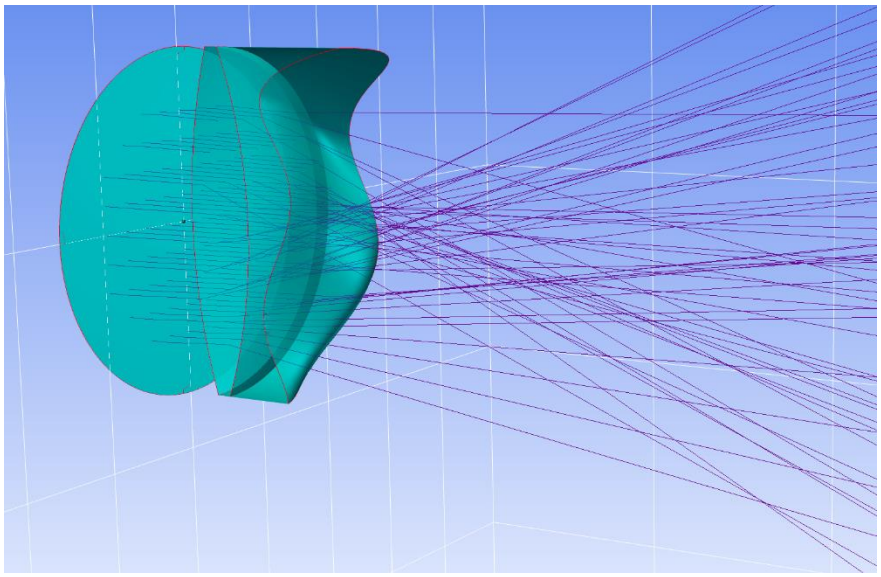


Figura A8: *Ejemplo de lente con una superficie definida por coeficientes de los polinomios de Zernike.*

APÉNDICE A.9 EJEMPLO-DOUBLET LENS TILT NONSEC

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens Tilt nonSec"""
import numpy as np
import KrakenOS as Kos


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.AxisMove = 1
L1a.TiltX = 13.0
L1a.DespZ = 5.0

L1b = Kos.surf()
L1b.Rc = (-3.071608670000159E+001)
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = (-7.819730726078505E+001)
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 1000.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, P_Obj, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 30
rad = 18.0
tsis = len(A) - 1
for j in range(-tam, tam + 1):
    i = 0
    x_0 = (i / tam) * rad
    y_0 = (j / tam) * rad
    r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
    if r < rad:
        tet = 0.0
        pSource_0 = [x_0, y_0, 0.0]
        dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
        W = 0.4
```

```
        Doblete.NsTrace(pSource_0, dCos, W)
        Rayos.push()
        W = 0.5
        Doblete.NsTrace(pSource_0, dCos, W)
        Rayos.push()
        W = 0.6
        Doblete.NsTrace(pSource_0, dCos, W)
        Rayos.push()

Kos.display3d(Doblete, Rayos, 2)
```

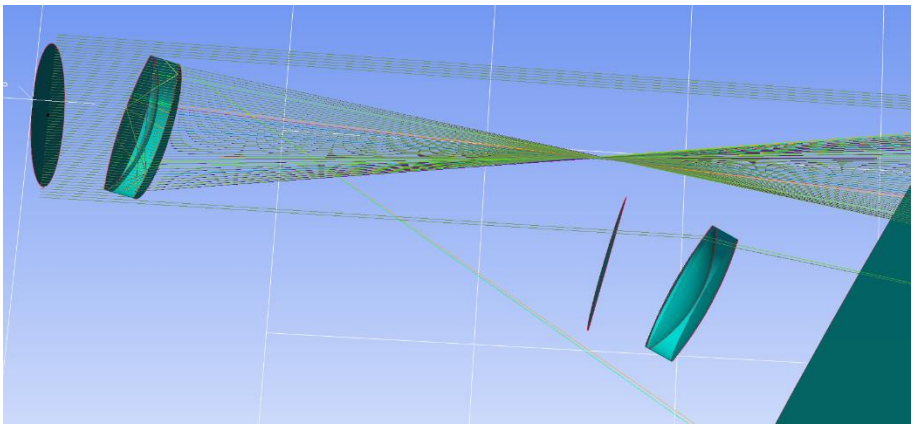


Figura A9: *Ejemplo de trazado de rayos no secuencial. Los rayos que no tocan al primer elemento son trazados hasta un segundo elemento donde sí tocan la superficie.*

APÉNDICE A.10 EJEMPLO-DOUBLET LENS PUPIL

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens Pupil"""
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 100
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0
P_Obj.Name = "P Obj"

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001 - 40
L1c.Glass = "AIR"
L1c.Diameter = 30

pupila = Kos.surf()
pupila.Rc = 30
pupila.Thickness = 40.
pupila.Glass = "AIR"
pupila.Diameter = 5
pupila.Name = "Ap Stop"
pupila.DespY = 0.

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 20.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, pupila, P_Ima]
config_1 = Kos.Setup()

Doblete = Kos.system(A, config_1)
Rayos = Kos.raykeeper(Doblete)

W = 0.4
Surf= 4
AperVal = 10
AperType = "EPD"
Pup = Kos.PupilCalc(Doblete, sup, W, AperType, AperVal)

print("Radio pupila de entrada: ")
print(Pup.RadPupInp)
```

```
print("Posición pupila de entrada: ")
print(Pup.PosPupInp)
print("Rádio pupila de salida: ")
print(Pup.RadPupOut)
print("Posicion pupila de salida: ")
print(Pup.PosPupOut)
print("Posicion pupila de salida respecto al plano focal: ")
print(Pup.PosPupOutFoc)
print("Orientación pupila de salida")
print(Pup.DirPupSal)

[L, M, N] = Pup.DirPupSal
print(L, M, N)

Pup.Samp = 5
Pup.Ptype = "fan"
Pup.FieldType = "angle"
Pup.FieldY = 2.0
x, y, z, L, M, N = Pup.Pattern2Field()
for i in range(0, len(x)):
    pSource_0 = [x[i], y[i], z[i]]
    dCos = [L[i], M[i], N[i]]
    Doblete.Trace(pSource_0, dCos, W)
    Rayos.push()

Pup.FieldY = -2.0
x, y, z, L, M, N = Pup.Pattern2Field()
for i in range(0, len(x)):
    pSource_0 = [x[i], y[i], z[i]]
    dCos = [L[i], M[i], N[i]]
    Doblete.Trace(pSource_0, dCos, W)
    Rayos.push()

Kos.display2d(Doblete, Rayos, 0)
```

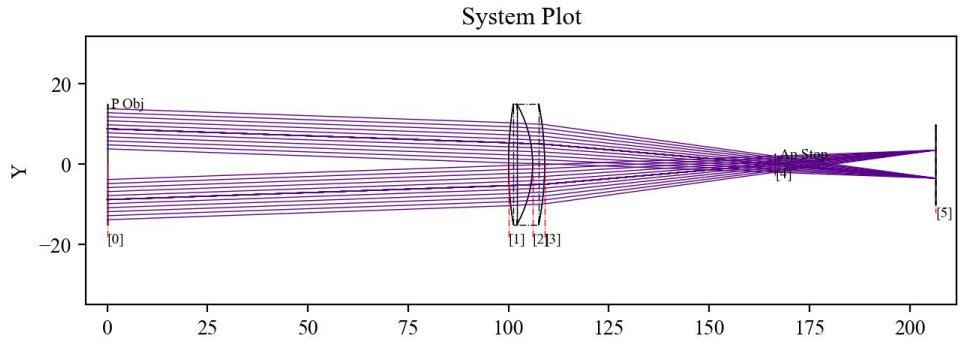


Figura A10: *Ejemplo de haces de rayos generados con la función de pupila. En ésta se define el lugar en donde se encuentra la pupila o la apertura del sistema.*

APÉNDICE A.11 EJEMPLO-DOUBLET LENS COMMANDS SYSTEM

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens Commands System"""
import numpy as np
import KrakenOS as Kos


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 0.1
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.

P_Obj2 = Kos.surf()
P_Obj2.Rc = 0.0
P_Obj2.Thickness = 10
P_Obj2.Glass = "AIR"
P_Obj2.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 18.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, P_Obj2, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

pSource_0 = [0, 14, 0]
tet = 0.1
dCos = [0.0, np.sin(np.deg2rad(tet)), -np.cos(np.deg2rad(tet))]
W = 0.4
Doblete.Trace(pSource_0, dCos, W)
Rayos.push()

Kos.display3d(Doblete, Rayos, 2)
```

```
print("Distancia focal efectiva")
print(Doblete.EFFL)
print("Plano principal anterior")
print(Doblete.PPA)
print("Plano principal posterior")
print(Doblete.PPP)
print("Superficies tocadas por el rayo")
print(Doblete.SURFACE)
print("Nombre de la superficie")
print(Doblete.NAME)
print("Vidrio de la superficie")
print(Doblete.GLASS)
print("Coordenadas del rayo en las superficies")
print(Doblete.XYZ)
print("Etc, ver documentaciòn")
print(Doblete.S_XYZ)
print(Doblete.T_XYZ)
print(Doblete.OST_XYZ)
print(Doblete.DISTANCE)
print(Doblete.OP)
print(Doblete.TOP)
print(Doblete.TOP_S)
print(Doblete.ALPHA)
print(Doblete.S_LMN)
print(Doblete.LMN)
print(Doblete.R_LMN)
print(Doblete.N0)
print(Doblete.N1)
print(Doblete.WAV)
print(Doblete.G_LMN)
print(Doblete.ORDER)
print(Doblete.GRATING)
print(Doblete.RP)
print(Doblete.RS)
print(Doblete.TP)
print(Doblete.TS)
print(Doblete.TTBE)
print(Doblete.TT)
print(Doblete.BULK_TRANS)
```

APÉNDICE A.12 EJEMPLO-DOUBLET LENS PUPIL SEIDEL

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens Pupil Seidel"""
import KrakenOS as Kos
import numpy as np


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 100
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0
P_Obj.Name = "P Obj"

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "N-BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001 - 40
L1c.Glass = "AIR"
L1c.Diameter = 30

pupila = Kos.surf()
pupila.Rc = 0
pupila.Thickness = 40.
pupila.Glass = "AIR"
pupila.Diameter = 15.0
pupila.Name = "Ap Stop"

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 20.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, pupila, P_Ima]
config_1 = Kos.Setup()

Doblete = Kos.system(A, config_1)

W = 0.6
Surf= 4
AperVal = 3
AperType = "EPD"
field = 3.25
fieldType = "angle"

AB = Kos.Seidel(Doblete, sup, W, AperType, AperVal, field, fieldType)
```

```
print( AB[0][0])
print(np.sum(AB[1][0]), np.sum(AB[1][1]), np.sum(AB[1][2]), np.sum(AB[1][3]),
np.sum(AB[1][4]))
j=1
print( AB[0][0+j])
print(np.sum(AB[1+j][0]), np.sum(AB[1+j][1]), np.sum(AB[1+j][2]), np.sum(AB[1+j][3]),
np.sum(AB[1+j][4]))
j=2
print( AB[0][0+j])
print(np.sum(AB[1+j][0]), np.sum(AB[1+j][1]), np.sum(AB[1+j][2]), np.sum(AB[1+j][3]),
np.sum(AB[1+j][4]))
j=3
print( AB[0][0+j])
print(np.sum(AB[1+j][0]), np.sum(AB[1+j][1]), np.sum(AB[1+j][2]), np.sum(AB[1+j][3]),
np.sum(AB[1+j][4]))

Pup = Kos.PupilCalc(Doblete, sup, W, AperType, AperVal)
Pup.Samp = 25
Pup.Ptype = "fan"
Pup.FieldY = field
x, y, z, L, M, N = Pup.Pattern2Field()
Rayos = Kos.raykeeper(Doblete)

for i in range(0, len(x)):
    pSource_0 = [x[i], y[i], z[i]]
    dCos = [L[i], M[i], N[i]]
    Doblete.Trace(pSource_0, dCos, W)
    Rayos.push()

Kos.display2d(Doblete, Rayos, 0)
```

```
------------------------------------
Seidel Aberration Coefficents
['si', 'sii', 'siii', 'siv', 'sv']
[-1.924670463514677e-06, -2.7721266737423735e-05, -0.0003183543169943356,
-5.20847647729677e-05, -0.0033844088571035732]
------------------------------------
Seidel coefficients in waves
['W040', 'W131', 'W222', 'W220', 'W311']
[-0.0004009730132322241, -0.023101055614519778, -0.2652952641619465,
-0.02170198532206988, -2.8203407142529766]
------------------------------------
Transverse Aberration Coefficents
['TSPH', 'TSCO', 'TTCO', 'TAST', 'TPFC', 'TSFC', 'TTFC', 'TDIS']
[6.41640871597908e-05, 0.0009241632834494345, 0.0027724898503483017,
-0.021226401641783688, -0.0017363862801152308, 0.012349587101007076,
0.03357598874279076, -0.11282840829541621]
------------------------------------
Longitudinal Aberration Coefficents
['LSPH', 'LSCO', 'LTCO', 'LAST', 'LPFC', 'LSFC', 'LTFC', 'LDIS']
[-0.0042781662202380916, -0.06161895721186755, -0.1848568716356025,
-1.4152788343257612, -0.11577425095091264, -0.8234136681137931,
-2.2386925024395543, 7.522879330467116]
------------------------------------
```

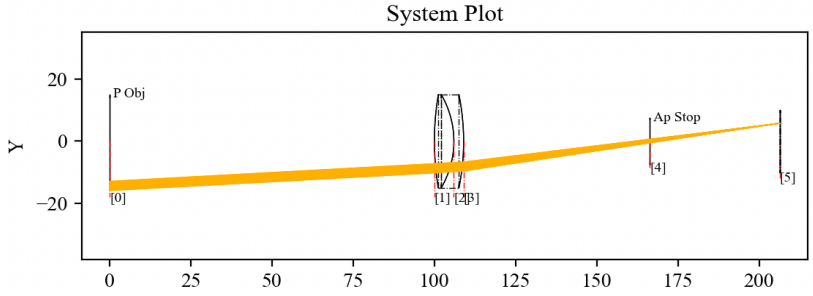


Figura A11. *Arriba: Valores de las sumas de Seidel. Abajo: Doblete con el que se calcularon las aberraciones utilizando ese campo delimitado por la imagen de la pupila.*

APÉNDICE A.13 EJEMPLO-DOUBLET LENS CYLINDER

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Doublet Lens Cylinder"""
import numpy as np
import KrakenOS as Kos


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30
L1b.TiltZ = 30
L1b.AxisMove = 0

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30
L1c.Cylinder_Rxy_Ratio = 0
L1c.TiltZ = 45
L1c.AxisMove = 0

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 100.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 10.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
```

```
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.5
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.6
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display3d(Doblete, Rayos, 1)
```

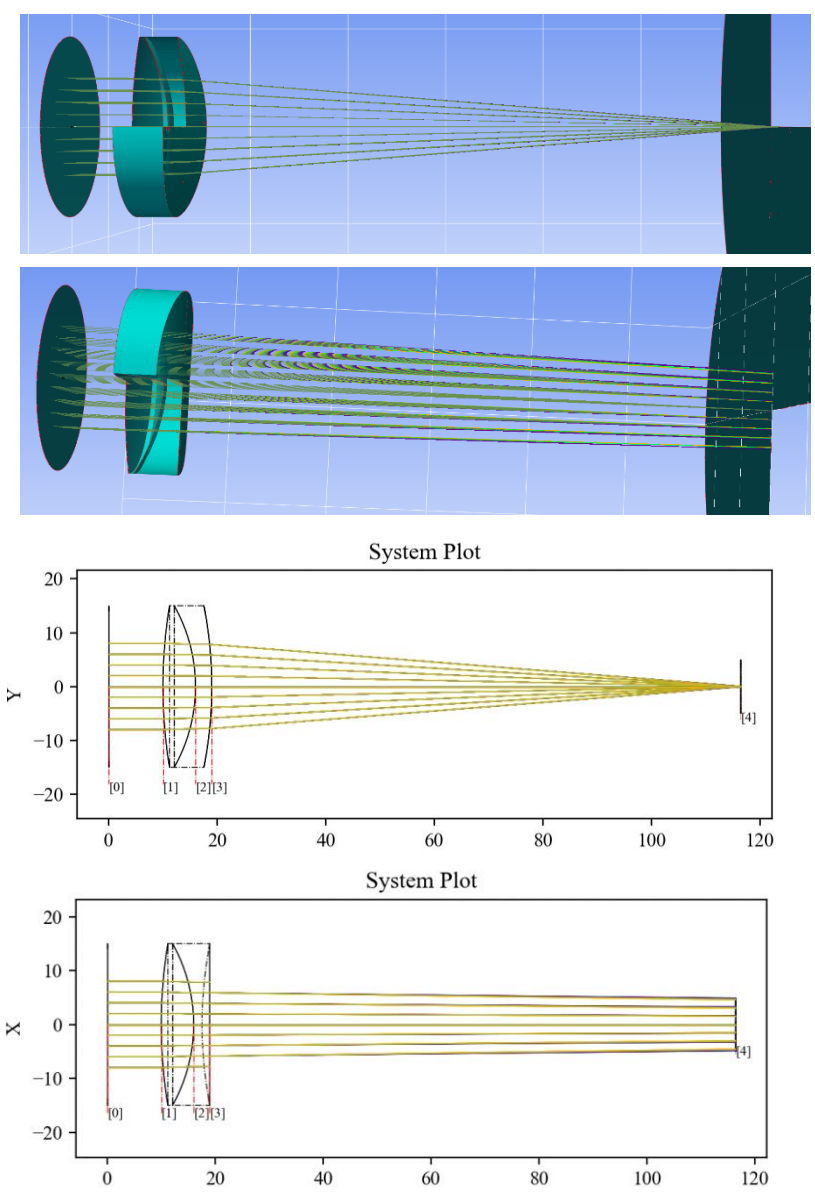


Figura A12: *Visualización 2D y 3D vistos en la dirección del eje X y posteriormente del eje Y. La última cara de la lente tiene un radio de curvatura que puede verse desde el eje Y, ésta es plana al verla desde el eje X.*

APÉNDICE A.14 EJEMPLO-AXICON

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Axicon"""
import numpy as np
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 0
L1a.Thickness = 26.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0

L1c = Kos.surf()
L1c.Rc = 0
L1c.Thickness = 9.737604742910693E+001
L1c.Axicon = -35.0
L1c.ShiftY = 0
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 100.0
P_Ima.Name = "Plano imagen"

configuracion_1 = Kos.Setup()
A = [P_Obj, L1a, L1c, P_Ima]

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 10.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.5
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.6
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
```

```
Kos.display3d(Doblete, Rayos, 2)
```

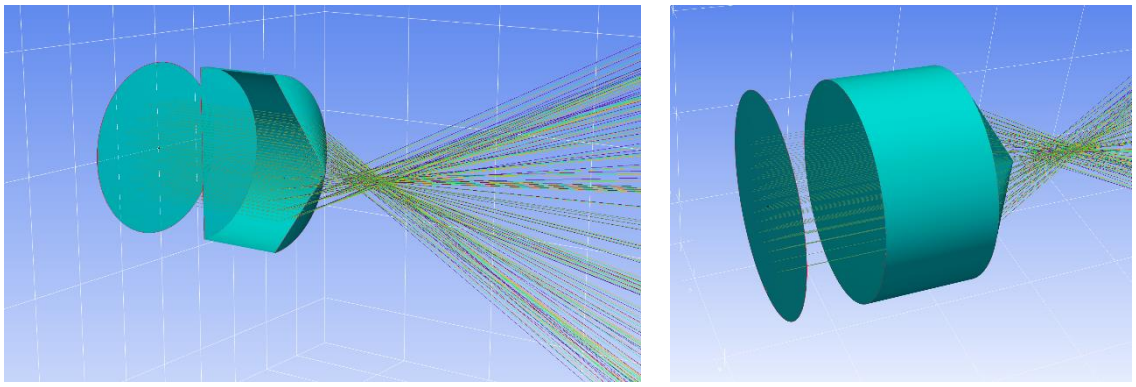


Figura A13: *Vista 3D de un Axicon, vista en sección transversal y completa.*

APÉNDICE A.15 EJEMPLO-AXICON AND CYLINDER

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Axicon and Cylinder"""
import numpy as np
import KrakenOS as Kos

configuracion_1 = Kos.Setup()

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 0
L1a.Thickness = 26.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0


L1c = Kos.surf()
L1c.Rc = 0.
L1c.K = -1
L1c.Thickness = 9.737604742910693E+001
L1c.Axicon = (-35.0)
L1c.ShiftY = 0
L1c.Cylinder_Rxy_Ratio = 0
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 100.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1c, P_Ima]

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 10.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.5
            Doblete.Trace(pSource_0, dCos, W)
```

```
            Rayos.push()
            W = 0.6
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display3d(Doblete, Rayos, 0)
```

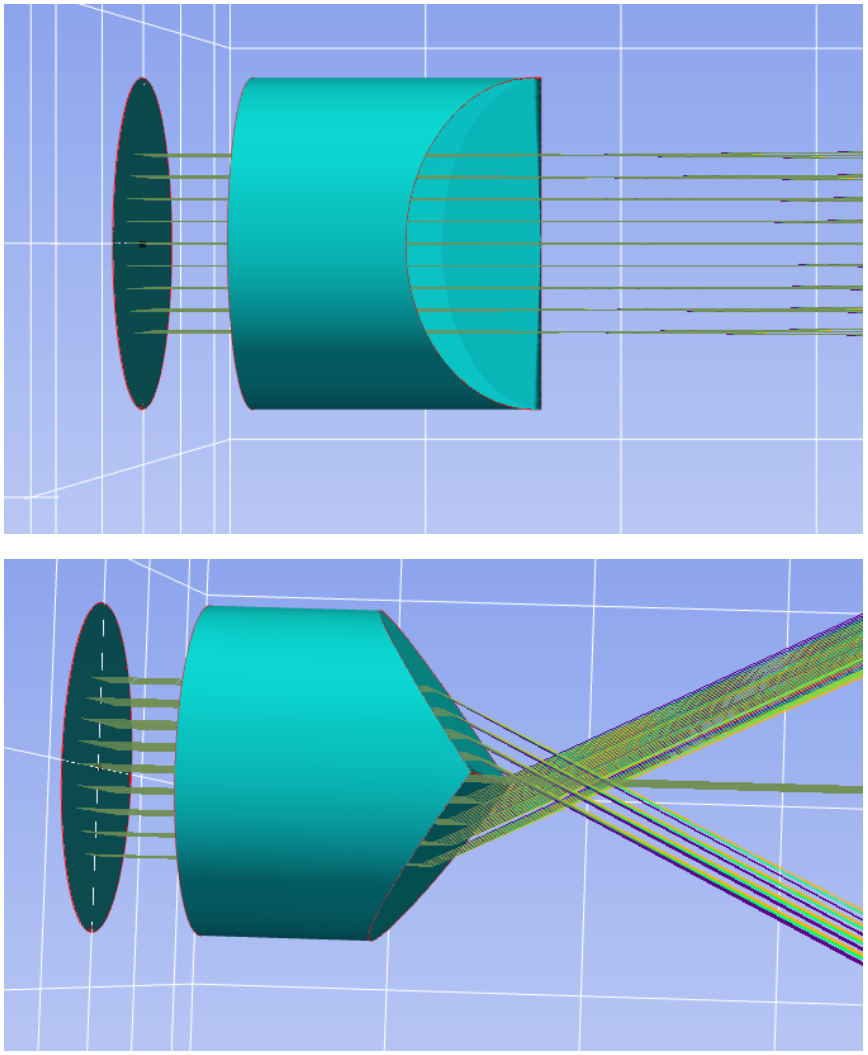


Figura A14: *Ejemplo de lente cilíndrica combinada con axicon.*

APÉNDICE A.16 EJEMPLO-FLAT MIRROR 45 DEG

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Flat Mirror 45 Deg"""
import time
import matplotlib.pyplot as plt
import numpy as np
import KrakenOS as Kos

start_time = time.time()


P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

POS_ESP = -40
L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001 + POS_ESP
L1c.Glass = "AIR"
L1c.Diameter = 30

Esp90 = Kos.surf()
Esp90.Thickness = POS_ESP
Esp90.Glass = "MIRROR"
Esp90.Diameter = 30.0
Esp90.Name = "Espejo a 90 grados"
Esp90.TiltX = 45.
Esp90.AxisMove = 2.

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 3.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, Esp90, P_Ima]
config_1 = Kos.Setup()

Doblete = Kos.system(A, config_1)
Rayos1 = Kos.raykeeper(Doblete)
Rayos2 = Kos.raykeeper(Doblete)
Rayos3 = Kos.raykeeper(Doblete)
RayosT = Kos.raykeeper(Doblete)
```

```
tam = 10
rad = 10.0
tsis = len(A) - 1
for j in range(-tam, tam + 1):
    for i in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos1.push()
            RayosT.push()
            W=0.5
            Doblete.Trace(pSource_0, dCos,W)
            Rayos2.push()
            RayosT.push()
            W=0.6
            Doblete.Trace(pSource_0, dCos,W)
            Rayos3.push()
            RayosT.push()

Kos.display3d(Doblete, RayosT, 2)

X, Y, Z, L, M, N = Rayos1.pick(-1)
plt.plot(X, Z, 'x')
X, Y, Z, L, M, N = Rayos2.pick(-1)
plt.plot(X, Z, 'x')
X, Y, Z, L, M, N = Rayos3.pick(-1)
plt.plot(X, Z, 'x')
plt.xlabel('numbers')
plt.ylabel('values')
plt.title('Spot Diagram')
plt.axis('square')
plt.show()
```

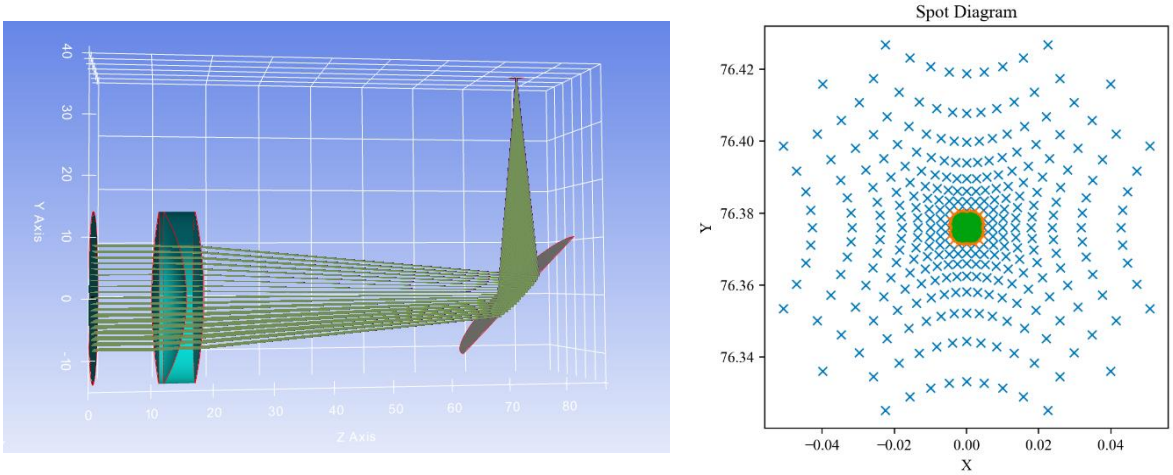


Figura A15. *Izquierda: Ejemplo de espejo diagonal con una rotación de 45°*
Derecha: Diagrama de manchas para tres distintas longitudes de onda, 0.4 µm, 0.5 µm y 0.6µm.

APÉNDICE A.17 EJEMPLO- PARABOLIC MIRROR SHIFT

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Parabolic Mirror Shift"""
import numpy as np
import KrakenOS as Kos


P_Obj = Kos.surf()
P_Obj.Thickness = 1000.0
P_Obj.Diameter = 300
P_Obj.Drawing = 0

M1 = Kos.surf()
M1.Rc = -2000.0
M1.Thickness = M1.Rc / 2
M1.k = -1.0
M1.Glass = "MIRROR"
M1.Diameter = 300
M1.ShiftY = 200

P_Ima = Kos.surf()
P_Ima.Glass = "AIR"
P_Ima.Diameter = 1600.0
P_Ima.Drawing = 0
P_Ima.Name = "Plano imagen"

A = [P_Obj, M1, P_Ima]
configuracion_1 = Kos.Setup()

Espejo = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Espejo)

tam = 5
rad = 150.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Espejo.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display3d(Espejo, Rayos, 0)
```

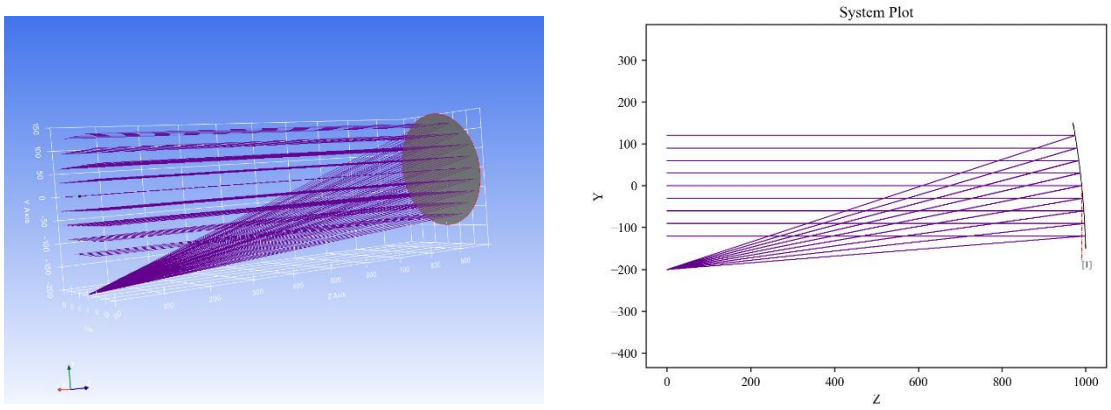


Figura A16: *Vista 3D y 2D de una parábola fuera de eje.*

APÉNDICE A.18 EJEMPLO- DIFFRACTION GRATING TRANSMISSION

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Diffraction Grating Transmission"""
import numpy as np
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

Dif_Obj_c1 = Kos.surf()
Dif_Obj_c1.Rc = 0.0
Dif_Obj_c1.Thickness = 1
Dif_Obj_c1.Glass = "BK7"
Dif_Obj_c1.Diameter = 30.0
Dif_Obj_c1.Grating_D = 1.0
Dif_Obj_c1.Diff_Ord = 1.
Dif_Obj_c1.Grating_Angle = 45.

Dif_Obj_c2 = Kos.surf()
Dif_Obj_c2.Rc = 0.0
Dif_Obj_c2.Thickness = 10
Dif_Obj_c2.Glass = "AIR"
Dif_Obj_c2.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 5.513435044607768E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0

L1b = Kos.surf()
L1b.Rc = -4.408716526030626E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -2.246906271406796E+002
L1c.Thickness = 9.737871661422000E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Name = "Plano imagen"
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 300.0

A = [P_Obj, Dif_Obj_c1, Dif_Obj_c2, L1a, L1b, L1c, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 10.0
```

```
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.5
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.6
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()


Kos.display2d(Doblete, Rayos, 0)
```

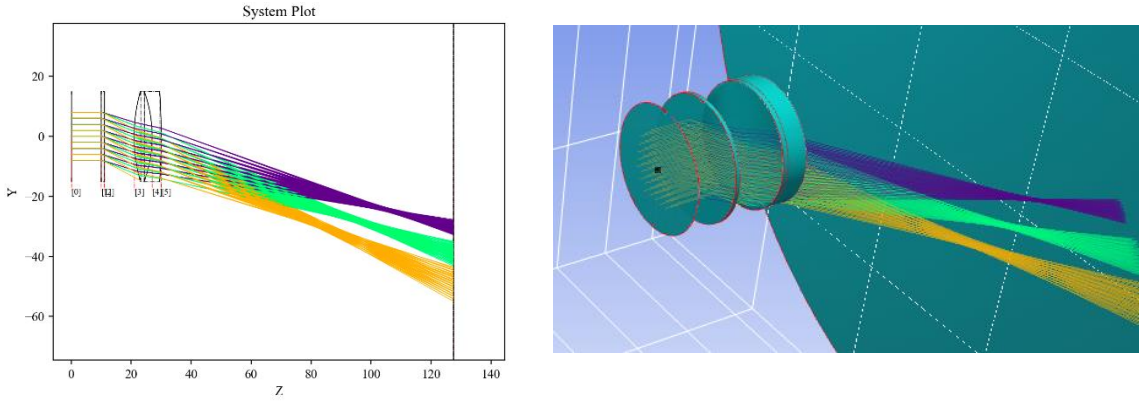


Figura A17: *Visualización 3D y 2D de un sistema que tiene una rejilla de difacción por transmisión.*

APÉNDICE A.19 EJEMPLO- DIFFRACTION GRATING REFLECTION

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Diffraction Grating Reflection"""
import numpy as np
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 5.513435044607768E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0

L1b = Kos.surf()
L1b.Rc = -4.408716526030626E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -2.246906271406796E+002
L1c.Thickness = 9.737871661422000E+001 - 50.0
L1c.Glass = "AIR"
L1c.Diameter = 30

Dif_Obj = Kos.surf()
Dif_Obj.Rc = 0.0
Dif_Obj.Thickness = -50
Dif_Obj.Glass = "MIRROR"
Dif_Obj.Diameter = 30.0
Dif_Obj.Grating_D = 1.0
Dif_Obj.Diff_Ord = 1
Dif_Obj.Grating_Angle = 45.0
Dif_Obj.Surface_type = 1

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Name = "Plano imagen"
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 300.0
P_Ima.Drawing = 0

A = [P_Obj, L1a, L1b, L1c, Dif_Obj, P_Ima]
configuracion_1 = Kos.Setup()

Doblete = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Doblete)

tam = 5
rad = 10.0
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
```

```
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.5
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()
            W = 0.6
            Doblete.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display2d(Doblete, Rayos, 1)
```

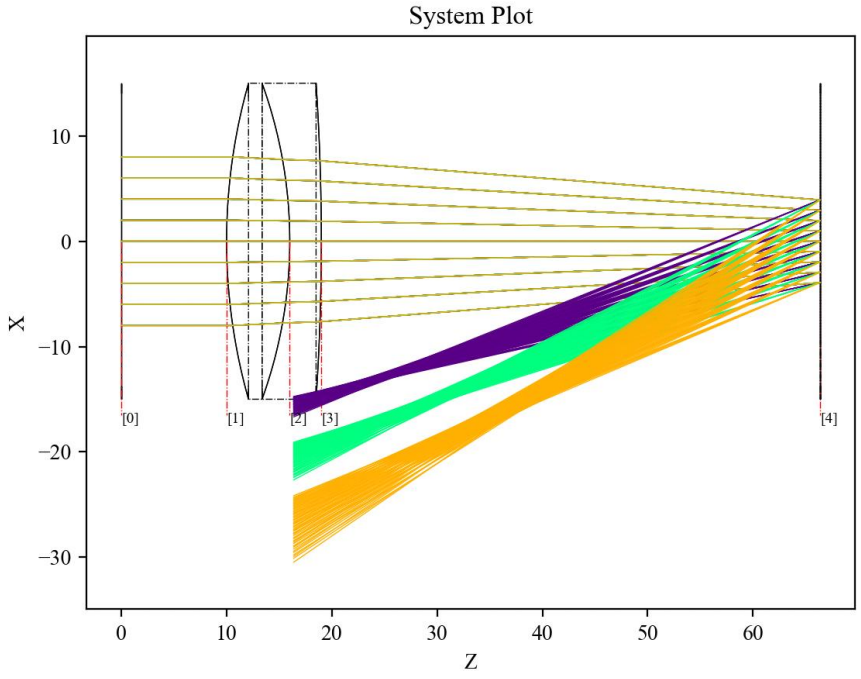


Figura A18: *Vista 2D de un sistema con rejilla de difracción por reflexión.*

APÉNDICE A.20 EJEMPLO- TEL 2M SPIDER SPOT DIAGRAM

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Tel 2M Spider Spot Diagram"""
import os
import numpy as np
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Thickness = 2.000000000000000E+003
P_Obj.Glass = "AIR"
P_Obj.Diameter = 6.796727741707513E+002 * 2.0
P_Obj.Drawing = 0

M1 = Kos.surf()
M1.Rc = -6.06044E+003
M1.Thickness = -1.774190000000000E+003 + 1.853722901194000E+000
M1.k = -1.637E+000
M1.Glass = "MIRROR"
M1.Diameter = 6.63448E+002 * 2.0
M1.InDiameter = 228.6 * 2.0
M1.DespY = 0.0
M1.TiltX = 0.0000
M1.AxisMove = 1

M2 = Kos.surf()
M2.Rc = -6.06044E+003
M2.Thickness = -M1.Thickness
M2.k = -3.5782E+001
M2.Glass = "MIRROR"
M2.Diameter = 2.995730651164167E+002 * 2.0
ED0 = np.zeros(20)
ED0[2] = 4.458178314555000E-018
M2.AspherData = ED0

Vertex = Kos.surf()
Vertex.Thickness = 130.0
Vertex.Glass = "AIR"
Vertex.Diameter = 600.0
Vertex.Drawing = 0

currentDirectory = os.getcwd()
direc = r"Prisma.stl"

objeto = Kos.surf()
objeto.Diameter = 118.0 * 2.0
objeto.Solid_3d_stl = direc
objeto.Thickness = 600
objeto.Glass = "BK7"
objeto.TiltX = 55
objeto.TiltY = 0
objeto.TiltZ = 45
objeto.DespX = 0
objeto.DespY = 0
objeto.AxisMove = 0

P_Ima = Kos.surf()
P_Ima.Rc = 0
P_Ima.Thickness = 100.0
P_Ima.Glass = "BK7"
P_Ima.Diameter = 500.0
```

```
P_Ima.Drawing = 1

A = [P_Obj, M1, M2, Vertex, objeto, P_Ima]
configuracion_1 = Kos.Setup()

Telescope = Kos.system(A, configuracion_1)
Rays = Kos.raykeeper(Telescope)

W = 0.633
tam = 5
rad = 6.56727741707513E+002
tsis = len(A) + 2
for gg in range(0, 10):
    for j in range(-tam, tam + 1):
        # j=0
        for i in range(-tam, tam + 1):
            x_0 = (i / tam) * rad
            y_0 = (j / tam) * rad
            r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
            if r < rad:
                tet = 0.0
                pSource_0 = [x_0, y_0, 0.0]
                # print("-..............")
                dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
                W = 0.633
                Telescope.NsTrace(pSource_0, dCos, W)
                Rays.push()

Kos.display3d(Telescope, Rays, 0)
print(Telescope.EFFL)
```

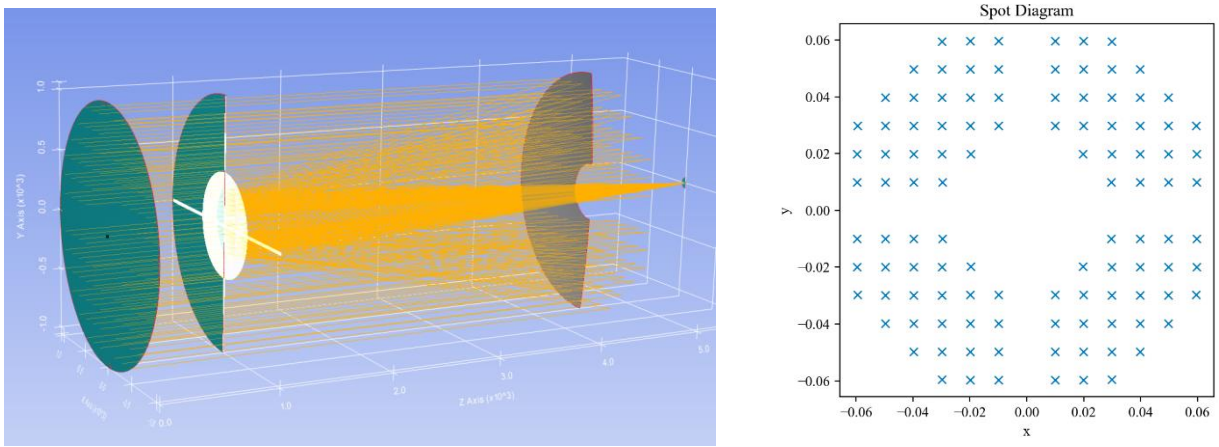
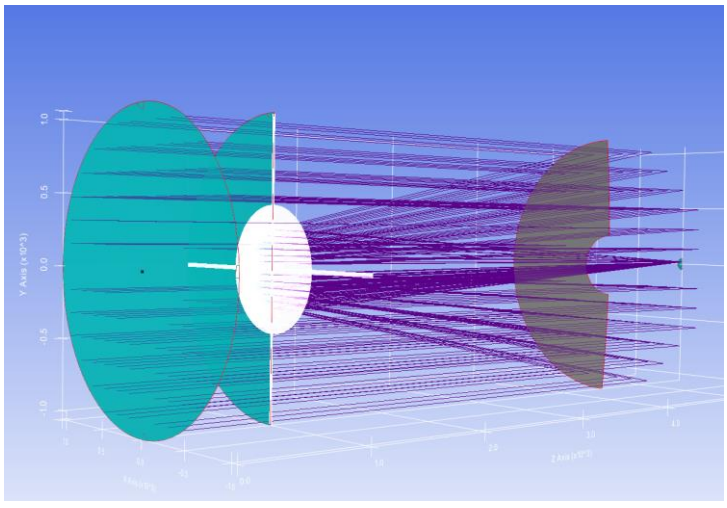


Figura A19. *Izquierda: vista del telescopio con la sombra de la araña*
Derecha: diagrama de manchas con un patrón rectangular; se puede notar la falta de los rayos que fueron obstruidos por la araña.

APÉNDICE A.21 EJEMPLO- TEL 2M SPIDER SPOT TILT M2

```
#             Rays.push()
#Kos.display3d(Telescope, Rays, 0)
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Tel 2M Spider Spot Tilt M2"""
import matplotlib.pyplot as plt
import numpy as np
import pyvista as pv
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Rc = 0
P_Obj.Thickness = 1000
P_Obj.Glass = "AIR"
P_Obj.Diameter = 1.059E+003 * 2.0

Spider = Kos.surf()
Spider.Rc = 999999999999.0
Spider.Thickness = 3.452229924716749E+003 + 100.0
Spider.Glass = "AIR"
Spider.Diameter = 1.059E+003 * 2.0

plane1 = pv.Plane(center=[0, 0, 0], direction=[0, 0, 1], i_size=30, j_size=2100,
i_resolution=10, j_resolution=10)
plane2 = pv.Plane(center=[0, 0, 0], direction=[0, 0, 1], i_size=2100, j_size=30,
i_resolution=10, j_resolution=10)
Baffle1 = pv.Disc(center=[0.0, 0.0, 0.0], inner=0, outer=875 / 2.0, normal=[0, 0, 1],
r_res=1, c_res=100)
Baffle2 = Baffle1.boolean_add(plane1)
Baffle3 = Baffle2.boolean_add(plane2)

AAA = pv.MultiBlock()
AAA.append(plane1)
AAA.append(plane2)
AAA.append(Baffle1)

Spider.Mask_Shape = AAA
Spider.Mask_Type = 2
Spider.TiltZ = 0

Thickness = 3.452200000000000E+003
M1 = Kos.surf()
M1.Rc = -9.638000000004009E+003
M1.Thickness = -Thickness
M1.k = -1.077310000000000E+000
M1.Glass = "MIRROR"
M1.Diameter = 1.059E+003 * 2.0
M1.InDiameter = 250 * 2.0

M2 = Kos.surf()
M2.Rc = -3.93E+003
M2.Thickness = Thickness + 1.037535322418897E+003
M2.k = -4.328100000000000E+000
M2.Glass = "MIRROR"
M2.Diameter = 3.365E+002 * 2.0
M2.TiltX = -9.657878504276254E-002
M2.DespY = -2.000000000000000E+000
M2.AxisMove = 0

P_Ima = Kos.surf()
```

```
P_Ima.Diameter = 100.0
P_Ima.Glass = "AIR"
P_Ima.Name = "Plano imagen"

A = [P_Obj, Spider, M1, M2, P_Ima]
configuracion_1 = Kos.Setup()

Telescopio = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Telescopio)

tam = 7
rad = 2200 / 2
tsis = len(A) - 1
for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Telescopio.Trace(pSource_0, dCos, W)
            Rayos.push()

Kos.display3d(Telescopio, Rayos, 2)
X, Y, Z, L, M, N = Rayos.pick(-1)

plt.plot(X, Y, 'x')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Spot Diagram')
plt.axis('square')
plt.show()
```



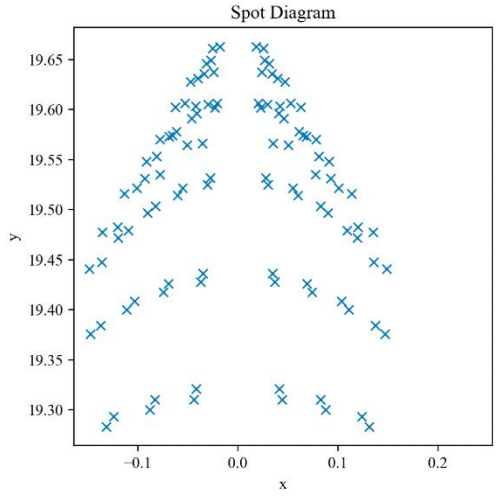


Figura A20: *Vista del telescopio con araña y un diagrama de manchas que muestra aberración de coma por el hecho de inclinar el espejo secundario.*

APÉNDICE A.22 EJEMPLO- TEL 2M PUPILA

```
# !/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp  TEl 2M Pupila"""
import matplotlib.pyplot as plt
import numpy as np
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Rc = 0
P_Obj.Thickness = 1000 + 3.452200000000000E+003
P_Obj.Glass = "AIR"
P_Obj.Diameter = 1.059E+003 * 2.0

Thickness = 3.452200000000000E+003
M1 = Kos.surf()
M1.Rc = -9.638000000004009E+003
M1.Thickness = -Thickness
M1.k = -1.077310000000000E+000
M1.Glass = "MIRROR"
M1.Diameter = 1.059E+003 * 2.0
M1.InDiameter = 250 * 2.0

M2 = Kos.surf()
M2.Rc = -3.93E+003
M2.Thickness = Thickness + 1.037525880125084E+003
M2.k = -4.328100000000000E+000
M2.Glass = "MIRROR"
M2.Diameter = 3.365E+002 * 2.0
M2.TiltY = 0.1
M2.TiltX = 0.1
M2.AxisMove = 0

P_Ima = Kos.surf()
P_Ima.Diameter = 300.0
P_Ima.Glass = "AIR"
P_Ima.Name = "Plano imagen"

A = [P_Obj, M1, M2, P_Ima]
configuracion_1 = Kos.Setup()
Telescopio = Kos.system(A, configuracion_1)

W = 0.4
Surf= 1
AperVal = 2010
AperType = "EPD" # "STOP"
Pup = Kos.PupilCalc(Telescopio, sup, W, AperType, AperVal)

print("Radio pupila de entrada: ")
print(Pup.RadPupInp)
print("Posicion pupila de entrada: ")
print(Pup.PosPupInp)
print("Radio pupila de salida: ")
print(Pup.RadPupOut)
print("Posicion pupila de salida: ")
print(Pup.PosPupOut)
print("Posicion pupila de salida respecto al plano focal: ")
print(Pup.PosPupOutFoc)
print("Orientación pupila de salida")
print(Pup.DirPupSal)
[L, M, N] = Pup.DirPupSal
```

```
print(L, M, N)
TetX = np.rad2deg(np.arcsin(-M))
TetY = np.rad2deg(np.arcsin(L / np.cos(np.arcsin(-M))))
print(TetX, TetY)
print("-----------------------------------------------------------------")

Pup.Samp = 10
Pup.Ptype = "hexapolar"
Pup.FieldY = 0.0
Pup.FieldType = "angle"
x, y, z, L, M, N = Pup.Pattern2Field()
Rayos = Kos.raykeeper(Telescopio)

for i in range(0, len(x)):
    pSource_0 = [x[i], y[i], z[i]]
    dCos = [L[i], M[i], N[i]]
    W = 0.4
    Telescopio.Trace(pSource_0, dCos, W)
    Rayos.push()

Kos.display3d(Telescopio, Rayos, 2)

X, Y, Z, L, M, N = Rayos.pick(-1)
plt.figure(300)
plt.plot(X, Y, 'x')
plt.axis('square')
plt.show(block=False)
```

```
Radio pupila de entrada:
1005.0
Posicion pupila de entrada:
[   0.     0.  4452.2]
Radio pupila de salida:
384.13536608352524
Posicion pupila de salida:
[  -4.37108607    4.37107941 -252.21310955]
Posicion pupila de salida respecto al plano focal:
[-4.37108607e+00  4.37107941e+00 -5.74193899e+03]
Orientación pupila de salida
[ 0.00349065 -0.00349064  0.99998782]
```

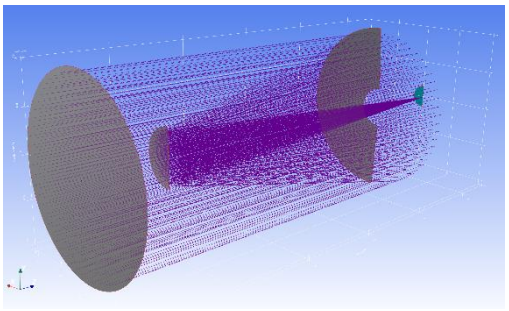


Figura A21. *Izquierda: obtención de los datos de la pupila. Derecha: Visualización 3D del telescopio.*

APÉNDICE A.23 EJEMPLO- TEL 2M ERROR MAP

```
# -*- coding: utf-8 -*-
Examp Tel 2M Error Map"""
import matplotlib.pyplot as plt
import numpy as np
import KrakenOS as Kos
import time

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]

P_Obj = Kos.surf()
P_Obj.Rc = 0
P_Obj.Thickness = 3500
P_Obj.Glass = "AIR"
P_Obj.Diameter = 1.059E+003 * 2.0

Thickness = 3.452200000000000E+003
M1 = Kos.surf()
M1.Rc = -9.638000000004009E+003
M1.Thickness = -Thickness
M1.k = -1.077310000000000E+000
M1.Glass = "MIRROR"
M1.Diameter = 1.059E+003 * 2.0
M1.InDiameter = 250 * 2.0
M1.Error_map = ErrorGen()

M2 = Kos.surf()
M2.Rc = -3.93E+003
M2.Thickness = Thickness + 1.037525880125084E+003
M2.k = -4.328100000000000E+000
M2.Glass = "MIRROR"
M2.Diameter = 3.365E+002 * 2.0
M2.AxisMove = 0

P_Ima = Kos.surf()
P_Ima.Diameter = 1000.0
P_Ima.Glass = "AIR"
P_Ima.Name = "Plano imagen"

A = [P_Obj, M1, M2, P_Ima]
configuracion_1 = Kos.Setup()
```

```
Telescopio = Kos.system(A, configuracion_1)
Rayos = Kos.raykeeper(Telescopio)

tam = 9
rad = 2100 / 2
tsis = len(A) - 1

start_time = time.time()


for i in range(-tam, tam + 1):
    for j in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            print(i)
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            W = 0.4
            Telescopio.Trace(pSource_0, dCos, W)
            Rayos.push()

print("--- %s seconds ---" % (time.time() - start_time))
Kos.display3d(Telescopio, Rayos, 2)
print(Telescopio.EFFL)
X, Y, Z, L, M, N = Rayos.pick(-1)

plt.plot(X, Y, 'x')
plt.xlabel('numbers')
plt.ylabel('values')
plt.title('Spot Diagram')
plt.axis('square')
plt.show()
```

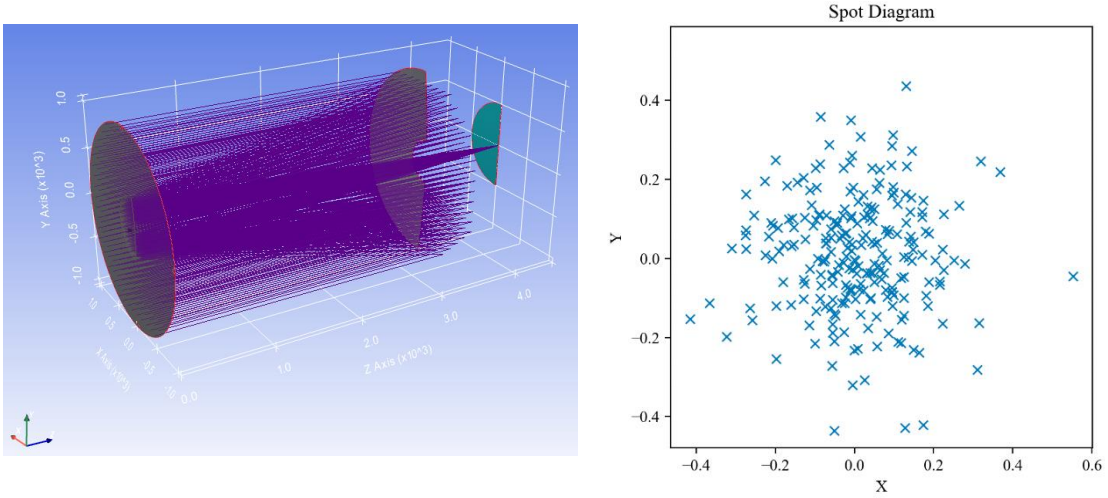


Figura A22. *Izquierda: visualización 3D de un telescopio generado con un mapa de deformaciones sumado a la función de forma del espejo primario. Derecha: Diagrama de manchas originado con este sistema.*

APÉNDICE A.24 EJEMPLO- TEL 2M WAVEFRONT FITTING

```
# !/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Tel 2M Wavefront Fitting"""
import os
import sys
import matplotlib.pyplot as plt
import numpy as np
import KrakenOS as Kos
from PhaseCalc import Phase

currentDirectory = os.getcwd()
sys.path.insert(1, currentDirectory + '/library')

P_Obj = Kos.surf()
P_Obj.Rc = 0
P_Obj.Thickness = 1000 + 3.452200000000000E+003
P_Obj.Glass = "AIR"
P_Obj.Diameter = 1.059E+003 * 2.0

Thickness = 3.452200000000000E+003
M1 = Kos.surf()
M1.Rc = -9.638000000004009E+003
M1.Thickness = -Thickness
M1.k = -1.077310000000000E+000
M1.Glass = "MIRROR"
M1.Diameter = 1.059E+003 * 2.0
M1.InDiameter = 250 * 2.0
M1.TiltY = 0.0
M1.TiltX = 0.0

M1.AxisMove = 0
M2 = Kos.surf()
M2.Rc = -3.93E+003
M2.Thickness = Thickness + 1037.525880
M2.k = -4.328100000000000E+000
M2.Glass = "MIRROR"
M2.Diameter = 3.365E+002 * 2.0
M2.TiltY = 0.0
M2.TiltX = 0.0
M2.DespY = 0.0
M2.DespX = 0.0
M2.AxisMove = 0

P_Ima = Kos.surf()
P_Ima.Diameter = 300.0
P_Ima.Glass = "AIR"
P_Ima.Name = "Plano imagen"

A = [P_Obj, M1, M2, P_Ima]
configuracion_1 = Kos.Setup()
Telescopio = Kos.system(A, configuracion_1)

W = 0.4
Surf= 1
Samp = 10
Ptype = "hexapolar"
FieldY = 0.1
FieldX = 0.0
FieldType = "angle"
```

```
AperType = "STOP"
fieldType = "angle"
AperVal = 2100.

Z, X, Y, P2V = Phase(Telescopio, sup, W, AperType, AperVal, configuracion_1, Samp, Ptype,
FieldY, FieldX, FieldType)
NC = 38
A = np.ones(NC)

z_coeff, MatNotation, w_rms, fitt_error = Kos.Zernike_Fitting(X, Y, Z, A)
A = np.abs(z_coeff)
Zeros = np.argwhere(A > 0.0001)
AA = np.zeros_like(A)
AA[Zeros] = 1
A = AA
z_coeff, MatNotation, w_rms, fitt_error = Kos.Zernike_Fitting(X, Y, Z, A)
print("Peak to valley: ", P2V)

for i in range(0, NC):
    print("z ", i + 1, "  ", "{0:.6f}".format(float(z_coeff[i])), "  :  ",
MatNotation[i])

print("RMS: ", "{:.4f}".format(float(w_rms)), " Error del ajuste: ", fitt_error)
z_coeff[0] = 0
print("RMS to chief: ", np.sqrt(np.sum(z_coeff * z_coeff)))
z_coeff[1] = 0
z_coeff[2] = 0
print("RMS to centroid: ", np.sqrt(np.sum(z_coeff * z_coeff)))

RR = Kos.raykeeper(Telescopio)
Pup = Kos.PupilCalc(Telescopio, sup, W, AperType, AperVal)
Pup.FieldX = FieldX
Pup.FieldY = FieldY
x, y, z, L, M, N = Pup.Pattern2Field()

for i in range(0, len(x)):
    pSource_0 = [x[i], y[i], z[i]]
    dCos = [L[i], M[i], N[i]]
    Telescopio.Trace(pSource_0, dCos, W)
    RR.push()

Kos.display3d(Telescopio, RR, 2)
X, Y, Z, L, M, N = RR.pick(-1)

plt.plot(X, Y, 'x')
plt.xlabel('numbers')
plt.ylabel('values')
plt.title('spot Diagram')
plt.axis('square')
plt.show()
```

```
Peak to valley:  -4.6972543604773245
z 1    0.562333 : 1^(1/2)(1.0)   Piston
z 2    -0.663146 : 4^(1/2)(1.0r^1)cos(T)   Tilt x, (about y axis)
z 3    0.022264 : 4^(1/2)(1.0r^1)sin(T)   Tilt y, (about x axis)
z 4    0.327227 : 3^(1/2)(-1.0+2.0r^2)   Power or Focus
z 5    0.016728 : 6^(1/2)(1.0r^2)sin(2T)   Astigmatism y, (45deg)
z 6    -0.087753 : 6^(1/2)(1.0r^2)cos(2T)   Astigmatism x, (0deg)
z 7    0.018916 : 8^(1/2)(-2.0r^1+3.0r^3)sin(T)   Coma y
z 8    -0.232776 : 8^(1/2)(-2.0r^1+3.0r^3)cos(T)   Coma x
z 9    0.000000 : 8^(1/2)(1.0r^3)sin(3T)   Trefoil y
z 10    0.000000 : 8^(1/2)(1.0r^3)cos(3T)   Trefoil x
z 11    0.000729 : 5^(1/2)(1.0+-6.0r^2+6.0r^4)   Primary Spherical
z 12    0.000155 : 10^(1/2)(-3.0r^2+4.0r^4)cos(2T)   Secondary Astigmatism x
z 13    0.000000 : 10^(1/2)(-3.0r^2+4.0r^4)sin(2T)   Secondary Astigmatism y
z 14    0.000000 : 10^(1/2)(1.0r^4)cos(4T)   Tetrafoil x
z 15    0.000000 : 10^(1/2)(1.0r^4)sin(4T)   Tetrafoil y
z 16    0.001140 : 12^(1/2)(3.0r^1+-12.0r^3+10.0r^5)cos(T)   Secondary Coma x
z 17    -0.000999 : 12^(1/2)(3.0r^1+-12.0r^3+10.0r^5)sin(T)   Secondary Coma y
z 18    0.000000 : 12^(1/2)(-4.0r^3+5.0r^5)cos(3T)   Secondary Trefoil x
```

Figura A23: *Valor de los coeficientes de los polinomios de Zernike ajustados al frente de onda del sistema para un campo dado. El resultado también incluye la expresión matemática del polinomio.*

APÉNDICE A.25 EJEMPLO- TEL 2M-STL_IMAGESLICER.PY

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Ejemplo - -2M-STL_ImageSlicer.py
"""
import KrakenOS as Kos
import numpy as np
import matplotlib.pyplot as plt
import scipy
import os
A1=1
if A1==0:

    P_Obj=Kos.surf()
    P_Obj.Rc=0
    P_Obj.Thickness=1000+3.452200000000000E+003
    P_Obj.Glass="AIR"
    P_Obj.Diameter=1.059E+003*2.0

    Thickness=3.452200000000000E+003
    M1=Kos.surf()
    M1.Rc=-9.638000000004009E+003
    M1.Thickness=-Thickness
    M1.k=-1.077310000000000E+000
    M1.Glass="MIRROR"
    M1.Diameter=1.059E+003*2.0
    M1.InDiameter=250*2.0

    M2=Kos.surf()
    M2.Rc=-3.93E+003
    M2.Thickness=Thickness+1.037525880125084E+003
    M2.k=-4.328100000000000E+000
    M2.Glass="MIRROR"
    M2.Diameter=3.365E+002*2.0
    M2.AxisMove=0

    P_Image_A=Kos.surf()
    P_Image_A.Diameter=10.0
    P_Image_A.Glass="AIR"
    P_Image_A.Thickness=10
    P_Image_A.Name="Image plane Tel"
    P_Image_A.DespZ=-100.0

    A=[P_Obj,M1,M2,P_Image_A]



    configuracion_1=Kos.Setup()
    Telescopio=Kos.system(A,configuracion_1)
    Rayos=Kos.raykeeper(Telescopio)


    # Gaussian
    def f(x):
        x=np.rad2deg(x)
        seing=1.2/3600.0
        sigma=seing/2.3548
        mean = 0
        standard_deviation = sigma
        y=scipy.stats.norm(mean, standard_deviation)
```

```
    res=y.pdf(x)
    return res


Sun = Kos.SourceRnd()
Sun.field=4*1.2/(2.0*3600.0)
Sun.fun = f
Sun.dim = 2100
Sun.num = 100000
L, M, N, X, Y, Z = Sun.rays()


Xr=np.zeros_like(L)
Yr=np.zeros_like(L)
Zr=np.zeros_like(L)

Lr=np.zeros_like(L)
Mr=np.zeros_like(L)
Nr=np.zeros_like(L)

NM=np.zeros_like(L)

con=0
con2=0
W = 0.6

for i in range(0,Sun.num):
    if con2==10:
        print(100.*i/Sun.num)
        con2=0

    pSource_0 = [X[i], Y[i], Z[i]]
    dCos = [L[i], M[i], N[i]]
    Telescopio.Trace(pSource_0, dCos, W)

    x,y,z=Telescopio.XYZ[-1]
    l,m,n=Telescopio.LMN[-1]
    Xr[con]=x
    Yr[con]=y
    Zr[con]=z
    Lr[con]=l
    Mr[con]=m
    Nr[con]=n


    if Telescopio.NAME[-1]=="Image plane Tel":
        NM[con]=i
    else:
        NM[con]=-1

    con=con+1
    con2=con2+1
    # Rayos.push()


args=np.argwhere(NM!=-1)


X=Xr[args]
Y=Yr[args]
Z=Zr[args]

L=Lr[args]
```

```
    M=Mr[args]
    N=Nr[args]
    W=W*np.ones_like(N)



    Rays=np.hstack((X,Y,Z,L,M,N,W))
    outfile="savedRays.npy"
    np.save(outfile, Rays)



#####################################################################
else:
    P_Obj=Kos.surf()
    P_Obj.Rc=0
    P_Obj.Thickness=100.+0.5
    P_Obj.Glass="AIR"
    P_Obj.Diameter=10



    currentDirectory = os.getcwd()
    direc = r"Jherrera-ImageSlicerBW-00.stl"
    P_ImageSlicer=Kos.surf()
    P_ImageSlicer.Diameter=10.0
    P_ImageSlicer.Glass="BK7"
    P_ImageSlicer.Name="Image slicer"
    P_ImageSlicer.Solid_3d_stl = direc
    P_ImageSlicer.Thickness = 13
    P_ImageSlicer.TiltX=180.0
    P_ImageSlicer.DespX=-0.55
    P_ImageSlicer.DespY=-0.03
    P_ImageSlicer.AxisMove=0

    P_Ima=Kos.surf()
    P_Ima.Diameter=10.0
    P_Ima.Glass="AIR"
    P_Ima.Name="Plano imagen"




    A=[P_Obj, P_ImageSlicer, P_Ima]
    configuracion_1 = Kos.Setup()
    ImageSlicer = Kos.system(A,configuracion_1)
    Rayos=Kos.raykeeper(ImageSlicer)



    outfile="savedRays.npy"
    R=np.load(outfile)
    print(np.shape(R))
    X,Y,Z,L,M,N,W=R[:,0],R[:,1],R[:,2],R[:,3],R[:,4],R[:,5],R[:,6]



    nrays=2000
    Xr=np.zeros(nrays)
    Yr=np.zeros(nrays)
    Zr=np.zeros(nrays)
    Lr=np.zeros(nrays)
```

```
    Mr=np.zeros(nrays)
    Nr=np.zeros(nrays)
    NM=np.zeros(nrays)

    con=0
    con2=0


    for i in range(0,nrays):
        if con2==10:
            print(100.*i/nrays)
            con2=0

        pSource_0 = [X[i], Y[i], Z[i]*0]
        dCos = [L[i], M[i], N[i]]
        ImageSlicer.NsTrace(pSource_0, dCos, W[i])

        x,y,z=ImageSlicer.XYZ[-1]
        l,m,n=ImageSlicer.LMN[-1]
        Xr[con]=x
        Yr[con]=y
        Zr[con]=z
        Lr[con]=l
        Mr[con]=m
        Nr[con]=n

        AA=ImageSlicer.SURFACE
        AA=np.asarray(AA)
        AW=np.argwhere(AA==1)
        if ImageSlicer.NAME[-1]=="Plano imagen" and len(AW)<10 and ImageSlicer.TT<0.9:



            # and ImageSlicer.TT<0.9 and ImageSlicer.TT>0.4


            NM[con]=i
            Rayos.push()
        else:
            NM[con]=-1

        con=con+1
        con2=con2+1



    args=np.argwhere(NM!=-1)

    X=Xr[args]
    Y=Yr[args]
    Z=Zr[args]

    L=Lr[args]
    M=Mr[args]
    N=Nr[args]
    W=W*np.ones_like(N)


####################
    plt.plot(X, Y, '.', c="r", markersize=1)

    # axis labeling
    plt.xlabel('x')
```

```
plt.ylabel('y')

# figure name
plt.title('Dot Plot')
plt.axis('square')
plt.show()


#              Rays.push()
Kos.display3d(ImageSlicer, Rayos, 0)
```

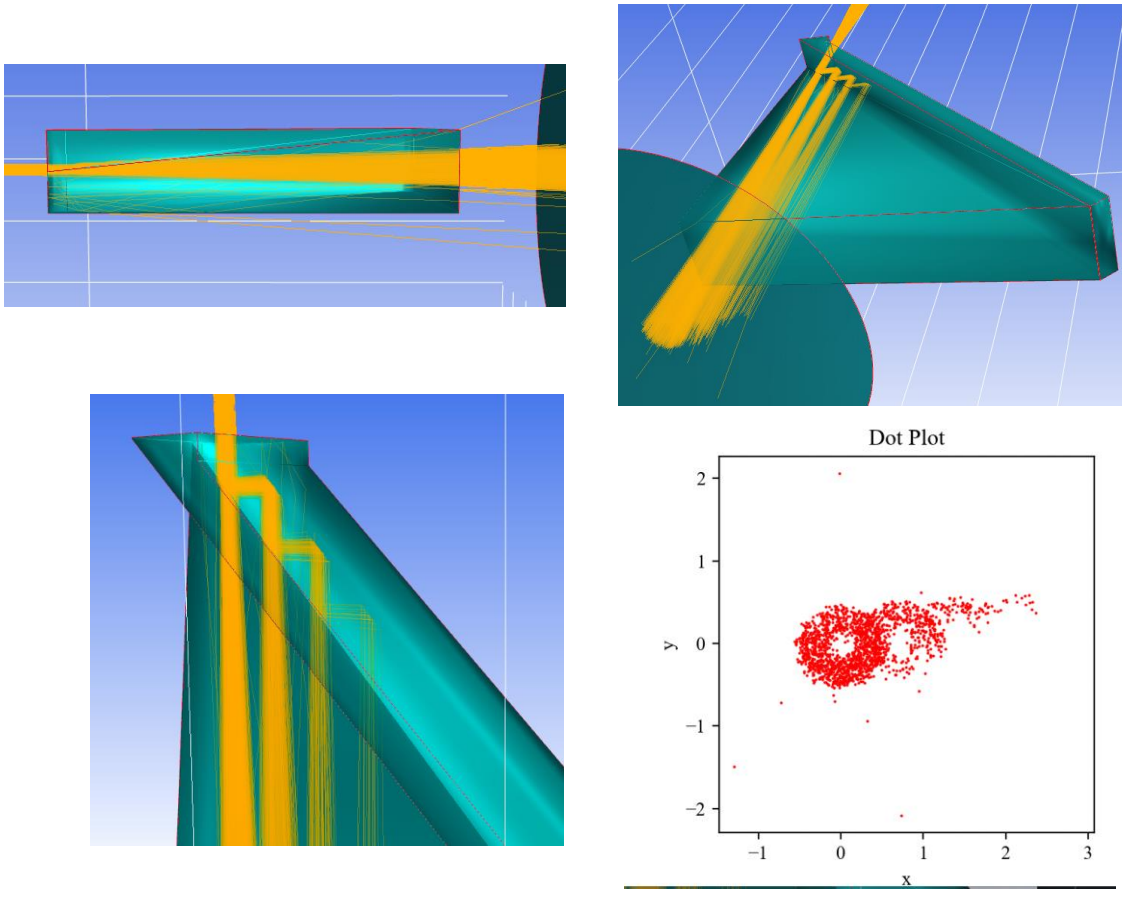


Figura A24: *Ejemplo de Image Slicer Bowel Walraven a partir de un modelo STL generado en OpenScad.*

APÉNDICE A.26 EJEMPLO- TEL 2M_ATMOSPHERIC_REFRACTION_CORRECTOR.PY

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Aug  2 12:04:14 2020
Ejemplo - -Tel_2M_Atmospheric_Refraction_Corrector.py

@author: joelherreravazquez
"""
import KrakenOS as Kos
import numpy as np
import matplotlib.pyplot as plt
import time


P_Obj=Kos.surf()
P_Obj.Rc=0
P_Obj.Thickness=1000+3.452200000000000E+003
P_Obj.Glass="AIR"
P_Obj.Diameter=1.059E+003*2.0

Thickness=3.452200000000000E+003
M1=Kos.surf()
M1.Rc=-9.638000000004009E+003
M1.Thickness=-Thickness
M1.k=-1.077310000000000E+000
M1.Glass="MIRROR"
M1.Diameter=1.059E+003*2.0
M1.InDiameter=250*2.0

M2=Kos.surf()
M2.Rc=-3.93E+003
M2.Thickness=Thickness+1.037525880125084E+003
M2.k=-4.328100000000000E+000
M2.Glass="MIRROR"
M2.Diameter=3.365E+002*2.0
M2.AxisMove=0

P_Ima=Kos.surf()
P_Ima.Diameter=1000.0
P_Ima.Glass="AIR"
P_Ima.Name="Plano imagen"
A=[P_Obj,M1,M2,P_Ima]

configuracion_1=Kos.Setup()
Telescopio=Kos.system(A,configuracion_1)
Rayos1=Kos.raykeeper(Telescopio)
Rayos2=Kos.raykeeper(Telescopio)
Rayos3=Kos.raykeeper(Telescopio)

W = 0.4
Surf= 1
AperVal = 2010
AperType = "EPD" # "STOP"
Pup = Kos.PupilCalc(Telescopio, sup, W, AperType, AperVal)
Pup.Samp=11
Pup.FieldType = "angle"
Pup.AtmosRef = 1
Pup.T    = 283.15     # k
Pup.P    = 101300     # Pa
Pup.H    = 0.5        # Humidity ratio 1 to 0
```

```
Pup.xc  = 400        # ppm
Pup.lat = 31         # degrees
Pup.h   = 2800       # meters
Pup.l1  = 0.60169    # micron
Pup.l2  = 0.50169    # micron
Pup.z0  = 55.0       # degrees
Pup.Ptype = "hexapolar"
Pup.FieldX = 0.0
W1 = 0.50169
Pup.l2  = W1
xa,ya,za,La,Ma,Na=Pup.Pattern2Field()
W2 = 0.60169
Pup.l2  = W2
xb,yb,zb,Lb,Mb,Nb=Pup.Pattern2Field()
W3 = 0.70169
Pup.l2  = W3
xc,yc,zc,Lc,Mc,Nc=Pup.Pattern2Field()
##############################################
for i in range(0,len(xa)):
    pSource_0 = [xa[i], ya[i], za[i]]
    dCos=[La[i], Ma[i], Na[i]]
    Telescopio.Trace(pSource_0, dCos, W1)
    Rayos1.push()
for i in range(0,len(xb)):
    pSource_0 = [xb[i], yb[i], zb[i]]
    dCos=[Lb[i], Mb[i], Nb[i]]
    Telescopio.Trace(pSource_0, dCos, W2)
    Rayos2.push()
for i in range(0,len(xc)):
    pSource_0 = [xc[i], yc[i], zc[i]]
    dCos=[Lc[i], Mc[i], Nc[i]]
    Telescopio.Trace(pSource_0, dCos, W3)
    Rayos3.push()


##############################################

# Kos.display3d(Telescopio,Rayos,2)
X,Y,Z,L,M,N=Rayos1.pick(-1)
plt.plot(X*1000.0,Y*1000.0, 'x', c="b")
X,Y,Z,L,M,N=Rayos2.pick(-1)
plt.plot(X*1000.0,Y*1000.0, 'x', c="r")
X,Y,Z,L,M,N=Rayos3.pick(-1)
plt.plot(X*1000.0,Y*1000.0, 'x', c="g")
# axis labeling
plt.xlabel('x')
plt.ylabel('y')
# figure name
plt.title('Spot Diagram')
plt.axis('square')
plt.ylim(-np.pi, np.pi)
plt.show()
```

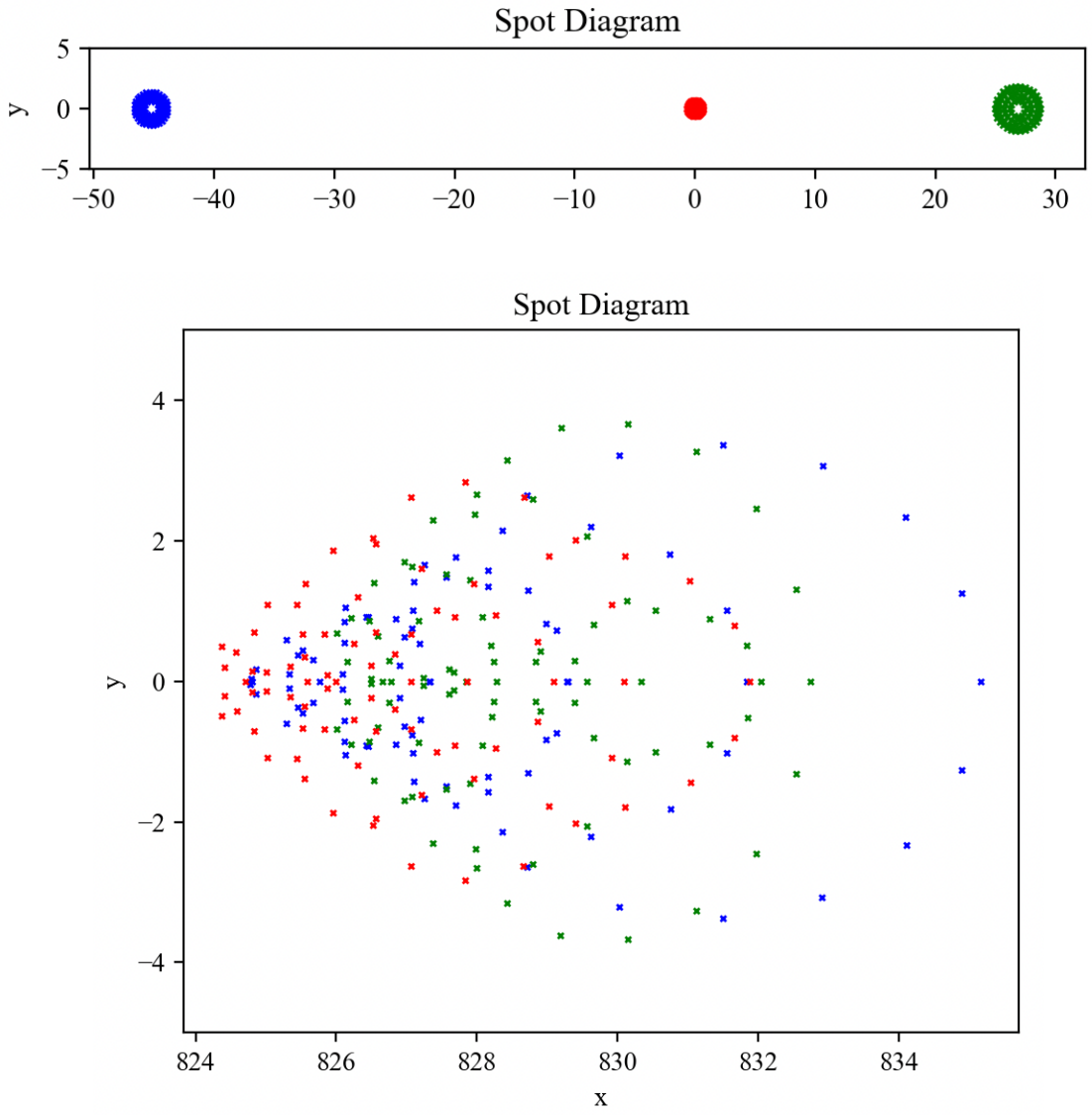


Figura A25. *Arriba: ejemplo de el efecto de la atmósfera en tres longitudes de onda distintas. Abajo: la implementación de un corrector de dispersión atmosférica.*

APÉNDICE A.27 EJEMPLO- EXTRA SHAPE MICRO LENS ARRAY

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Extra Shape Micro Lens Array"""
import KrakenOS as Kos
import numpy as np
import matplotlib.pyplot as plt

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 55.134*0
L1a.Thickness = 2.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0

L1c = Kos.surf()
L1c.Thickness = 40
L1c.Glass = "AIR"
L1c.Diameter = 30

def f(x,y,E):
    DeltaX=E[0]*np.rint(x/E[0])
    DeltaY=E[0]*np.rint(y/E[0])
    x=x-DeltaX
    y=y-DeltaY
    s = np.sqrt((x * x) + (y * y))
    c = 1.0 / E[1]
    InRoot = 1 - (E[2] + 1.0) * c * c * s * s
    z = (c * s * s / (1.0 + np.sqrt(InRoot)))
    return z

coef=[3.0, -3, 0]
L1c.ExtraData=[f, coef]
L1c.Res=2

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 300.0
P_Ima.Name = "Image plane"

A = [P_Obj, L1a,  L1c, P_Ima]
Config_1 = Kos.Setup()

Lens = Kos.system(A, Config_1)
Rays = Kos.raykeeper(Lens)

Wav = 0.45
for i in range(-100, 100+1):
    pSource = [0.0, i/10., 0.0]
    dCos = [0.0, 0.0, 1.0]
    Lens.Trace(pSource, dCos, Wav)
    Rays.push()
```

```
Kos.display3d(Lens, Rays, 1)
Kos.display2d(Lens, Rays, 0)
```

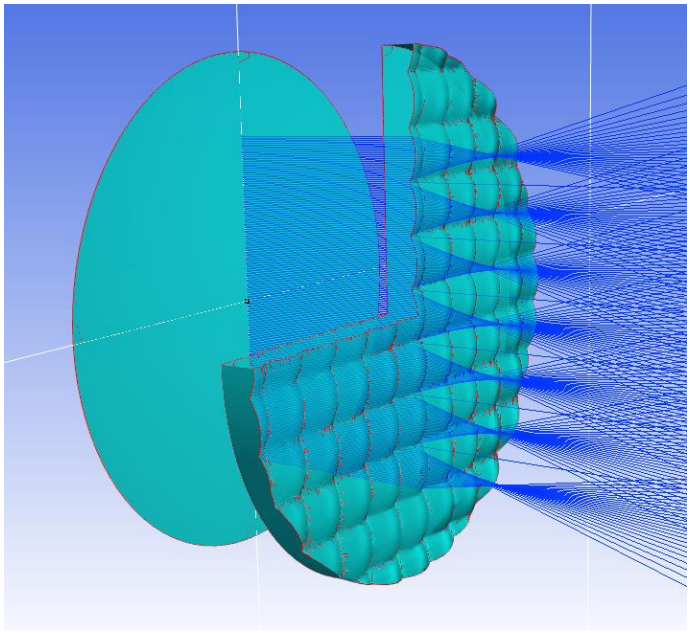


Figura A26: *Ejemplo de arreglo de micro lentes definida por el usuario en una superficie del tipo ExtraShape.*

APÉNDICE A.28 EJEMPLO- EXTRA SHAPE RADIAL SINE

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Extra Shape Radial Sine"""
import KrakenOS as Kos
import numpy as np
import matplotlib.pyplot as plt

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0
P_Obj.Drawing = 0

L1a = Kos.surf()
L1a.Rc = 55.134
L1a.Thickness = 9.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0

L1c = Kos.surf()
L1c.Rc = -224.69
L1c.Thickness = 40
L1c.Glass = "AIR"
L1c.Diameter = 30

def f(x,y,E):
    r=np.sqrt(x*x+y*y)
    r=np.asarray(r)
    H=2.0*np.pi*r/E[0]
    z=np.sin(H) * E[1]
    return z

coef = np.zeros(36)
coef[0]=5
coef[1]=.5
ES = [f, coef]
L1c.ExtraData=ES

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 200.0
P_Ima.Name = "Image plane"

A = [P_Obj, L1a,  L1c, P_Ima]
Config_1 = Kos.Setup()

Lens = Kos.system(A, Config_1)
Rays = Kos.raykeeper(Lens)

Wav = 0.45
for i in range(-100, 100+1):
    pSource = [0.0, i/10., 0.0]
    dCos = [0.0, 0.0, 1.0]
    Lens.Trace(pSource, dCos, Wav)
    Rays.push()
```

```
Kos.display3d(Lens, Rays, 2)
Kos.display2d(Lens, Rays, 0)
```

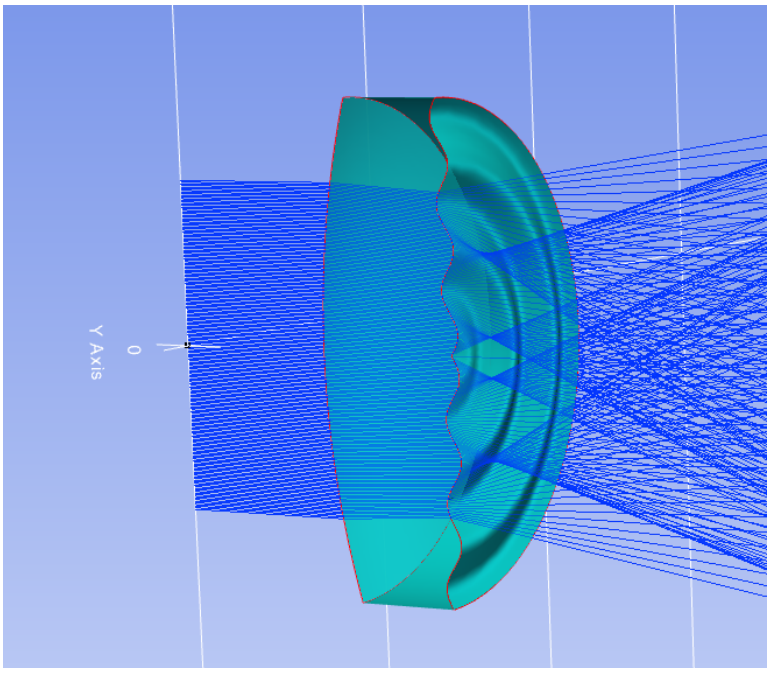


Figura A27: *Vista de una lente con una superficie definida por el usuario por medio de una función radial.*

APÉNDICE A.29 EJEMPLO- EXTRA SHAPE XY COSINES

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Extra Shape XY Cosines"""
import KrakenOS as Kos
import numpy as np
import matplotlib.pyplot as plt

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 55.134*0
L1a.Thickness = 9.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0

L1c = Kos.surf()
L1c.Thickness = 40
L1c.Glass = "AIR"
L1c.Diameter = 30

def f(x,y,E):
    r = np.sqrt((x * x) + (y * y * 0))
    H=2.0*np.pi*r/E[0]
    zx=np.abs(np.cos(H) * E[1])
    r = np.sqrt((x * x * 0) + (y * y))
    H=2.0*np.pi*r/E[0]
    zy=np.abs(np.cos(H) * E[1])
    return zx+zy

coef=[10.0,1.]
L1c.ExtraData=[f, coef]
L1c.Res=1

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 300.0
P_Ima.Name = "Image plane"

A = [P_Obj, L1a,  L1c, P_Ima]
Config_1 = Kos.Setup()

Lens = Kos.system(A, Config_1)
Rays = Kos.raykeeper(Lens)

Wav = 0.45
for i in range(-100, 100+1):
    pSource = [0.0, i/10., 0.0]
    dCos = [0.0, 0.0, 1.0]
    Lens.Trace(pSource, dCos, Wav)
    Rays.push()


Kos.display3d(Lens, Rays, 2)
Kos.display2d(Lens, Rays, 0)
```

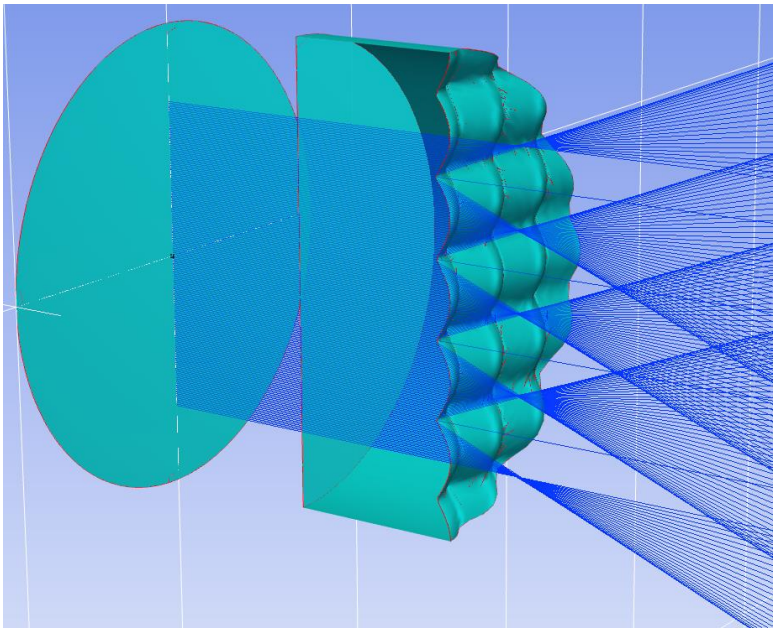


Figura A28: *Vista del corte de una lente definida por el usuario con una función de sagita con componentes en X y Y.*

APÉNDICE A.30 EJEMPLO- MULTICORE

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Multicore"""
import multiprocessing
import time
import numpy as np
import KrakenOS as Kos

start_time = time.time()

P_Obj = Kos.surf()
P_Obj.Rc = 0.0
P_Obj.Thickness = 10
P_Obj.Glass = "AIR"
P_Obj.Diameter = 30.0

L1a = Kos.surf()
L1a.Rc = 9.284706570002484E+001
L1a.Thickness = 6.0
L1a.Glass = "BK7"
L1a.Diameter = 30.0
L1a.Axicon = 0

L1b = Kos.surf()
L1b.Rc = -3.071608670000159E+001
L1b.Thickness = 3.0
L1b.Glass = "F2"
L1b.Diameter = 30

L1c = Kos.surf()
L1c.Rc = -7.819730726078505E+001
L1c.Thickness = 9.737604742910693E+001
L1c.Glass = "AIR"
L1c.Diameter = 30

P_Ima = Kos.surf()
P_Ima.Rc = 0.0
P_Ima.Thickness = 0.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 3.0
P_Ima.Name = "Plano imagen"

A = [P_Obj, L1a, L1b, L1c, P_Ima]
config_1 = Kos.Setup()

Doblete1 = Kos.system(A, config_1)

def trax1(xyz, lmn, w, q):
    Rayos = Kos.raykeeper(Doblete1)
    start_time = time.time()
    for i in range(0, 700):
        Doblete1.Trace(xyz, lmn, w)
        Rayos.push()
    A = Rayos.pick(-1)
    Rayos.clean()
    q.put(A[0])
    print("--- %s seconds ---" % (time.time() - start_time))

if __name__ == '__main__':
    start_time = time.time()
```

```
pSource_0 = [1, 0, 0.0]
dCos = [0.0, np.sin(np.deg2rad(0)), np.cos(np.deg2rad(0))]
w = 0.5
q = multiprocessing.Queue()
p1 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.1, q))
p2 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.2, q))
p3 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.3, q))
p4 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.4, q))
p5 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.5, q))
p6 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.6, q))
p7 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.7, q))
p8 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.8, q))
p9 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.1, q))
p10 = multiprocessing.Process(target=trax1, args=(pSource_0, dCos, w + 0.2, q))

p1.start()
p2.start()
p3.start()
p4.start()
p5.start()
p6.start()
p7.start()
p8.start()
p9.start()
p10.start()

p1.join()
p2.join()
p3.join()
p4.join()
p5.join()
p6.join()
p7.join()
p8.join()
p9.join()
p10.join()
print(".................................................")
print("Total time :")
print("--- %s seconds ---" % (time.time() - start_time))

for i in range(0,10):

    A = q.get()
    print(len(A))
```

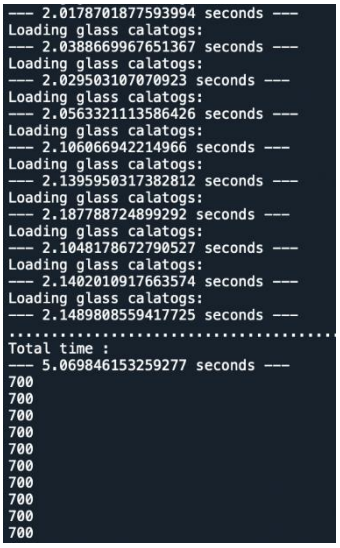


Figura A29: *Salida de consola al ejecutar el mismo trazado de rayos con 8 núcleos del procesador simultáneamente.*

APÉNDICE A.31 EJEMPLO- SOLID OBJECTS STL ARRAY

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
Examp Solid Objects STL ARRAY"""
import matplotlib.pyplot as plt
import numpy as np
import pyvista as pv
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Thickness = 5000.0
P_Obj.Glass = "AIR"
P_Obj.Diameter = 6.796727741707513E+002 * 2.0
P_Obj.Drawing = 0

FOV = 0.5
Ref_esp = 0.8
Tf = 200  # 40
A1 = Tf * Tf
Conc = 800
Conc = Conc / 0.8
fpl = int(np.round(np.sqrt(Conc) / 2.0))
print("Numero de facetas por lado (Calculo 1)", (fpl * 2.0) + 1.0)
print("Tamanio por lado(mm) ", Tf * ((fpl * 2.0) + 1.0))

n = fpl
FN = 1
focal = Tf * (fpl * 2.0 + 1.0) * FN
print("Distancia focal(mm) ", focal)

sobredim = 2.0 * focal * np.tan(np.deg2rad(FOV / 2.0))
print("Sobredim (mm): ", sobredim)

Tf = Tf - sobredim
A2 = Tf * Tf
RA = A1 / A2
Conc = Conc * RA
print("Nuevo tamaño de las facetas: ", Tf)

fpl = int(np.round(np.sqrt(Conc) / 2.0))
print("Nuevo numero de facetas por lado (Calculo 2)", (fpl * 2.0) + 1.0)
print("Tamanio por lado 2a (mm) ", Tf * ((fpl * 2.0) + 1.0))

n = fpl
Cx = Tf
Cy = Tf
Cz = 0
Lx = Tf
Ly = Tf
Lz = 1.0

element0 = pv.Cube(center=(0.0, 0.0, 0.0), x_length=0.1, y_length=0.1, z_length=0.1,
bounds=None)
for A in range(-n, n + 1):
    for B in range(-n, n + 1):
        Ty = 0.5 * np.rad2deg(np.arctan2(Cx * A, focal))
        Tx = -0.5 * np.rad2deg(np.arctan2(Cy * B, focal))
        element1 = pv.Cube(center=(0.0, 0.0, 0.0), x_length=Lx, y_length=Ly, z_length=Lz,
bounds=None)
        element1.rotate_x(Tx)
        element1.rotate_y(Ty)
```

```
        v = [-Cx / 2.0, -Cy / 2.0, 0]
        element1.translate(v)
        v = [Cx * A, Cy * B, Cz]
        element1.translate(v)
        element0 = element0 + element1
element0.save("salida.stl")
direc = r"salida.stl"

objeto = Kos.surf()
objeto.Diameter = 118.0 * 2.0
objeto.Solid_3d_stl = direc
objeto.Thickness = -6000
objeto.Glass = "MIRROR"
objeto.TiltX = 0
objeto.TiltY = 0
objeto.DespX = 0
objeto.DespY = 0
objeto.AxisMove = 0

P_Ima = Kos.surf()
P_Ima.Rc = 0
P_Ima.Thickness = -1.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 2000.0
P_Ima.Drawing = 1
P_Ima.Name = "Plano imagen"

A = [P_Obj, objeto, P_Ima]
configur = Kos.Setup()

Telescope = Kos.system(A, configur)
Rays = Kos.raykeeper(Telescope)

W = 0.633
tam = 25
rad = 5500.0
tsis = len(A) + 2
for j in range(-tam, tam + 1):
    for i in range(-tam, tam + 1):
        x_0 = (i / tam) * rad
        y_0 = (j / tam) * rad
        r = np.sqrt((x_0 * x_0) + (y_0 * y_0))
        if r < rad:
            tet = 0.0
            pSource_0 = [x_0, y_0, 0.0]
            dCos = [0.0, np.sin(np.deg2rad(tet)), np.cos(np.deg2rad(tet))]
            Telescope.NsTrace(pSource_0, dCos, W)
            if np.shape(Telescope.NAME)[0] != 0:
                if Telescope.NAME[-1] == "Plano imagen":
                    plt.plot(Telescope.Hit_x[-1], Telescope.Hit_y[-1], '.', c="g")
                    Rays.push()

plt.axis('square')
plt.show()
Kos.display3d(Telescope, Rays, 0)
```

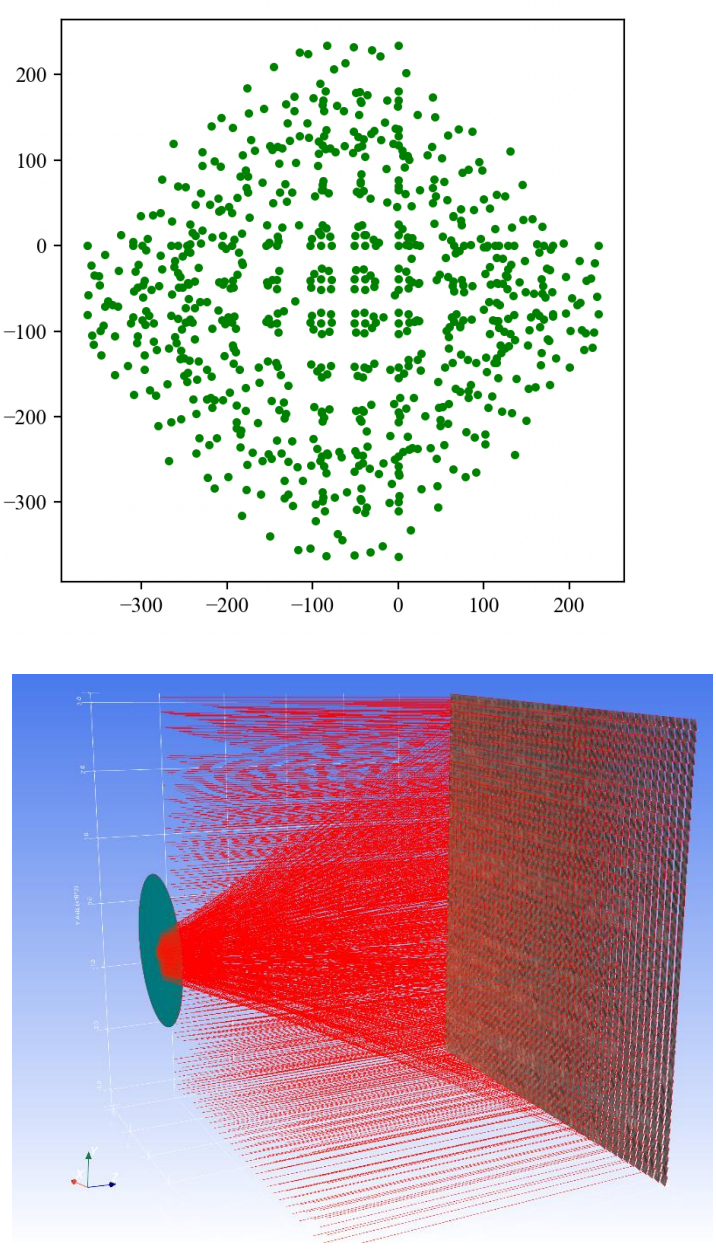


Figura A30. *Superior: diagrama de manchas generado por un arreglo de espejos planos mostrados en la figura inferior.*

APÉNDICE A.32 EJEMPLO- SOURCE DISTRIBUTION FUNCTION

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Aug  2 12:04:14 2020
Ejemplo - -Source_Distribution_Function.py
"""

import matplotlib.pyplot as plt
import numpy as np
import pyvista as pv
import scipy
import KrakenOS as Kos

P_Obj = Kos.surf()
P_Obj.Thickness = 5000.0
P_Obj.Glass = "AIR"
P_Obj.Diameter = 6.796727741707513E+002 * 2.0
P_Obj.Drawing = 0
#############################################################################################
objeto = Kos.surf()
objeto.Rc=-12000
objeto.k=-1
objeto.Diameter=2500
objeto.Thickness = -6000
objeto.Glass = "MIRROR"
P_Ima = Kos.surf()
P_Ima.Rc = 0
P_Ima.Thickness = -1.0
P_Ima.Glass = "AIR"
P_Ima.Diameter = 6000.0
P_Ima.Drawing = 1
P_Ima.Name = "Plano imagen"
A = [P_Obj, objeto, P_Ima]
configur = Kos.Setup()
Telescope = Kos.system(A, configur)
Rays = Kos.raykeeper(Telescope)
W = 0.633
Sun = Kos.SourceRnd()
Ejemplo - =4
if Ejemplo -  == 0:
    # Sun distribution
    def f(x):
        teta=x
        FI=np.zeros_like(teta)
        Arg2=np.argwhere(teta>(4.65/1000.0))
        FI=np.cos(0.326 * teta)/np.cos(0.308*teta)

        Chi2=.03
        k=0.9* np.log(13.5*Chi2)*np.power(Chi2,-0.3)
        r=(2.2* np.log(0.52*Chi2)*np.power(Chi2,0.43))-1.0
        FI[Arg2]= np.exp(k)*np.power(teta[Arg2] * 1.0e3 , r)
        return FI
    Sun.field =20*np.rad2deg((4.65/1000.0))
if Ejemplo -  == 1:
    # Sinc cunction
    def f(x):
        Wh=0.025
        r=(x*90.0/0.025)*np.pi
        res=np.sin(r)/r
        return res
```

```
    Sun.field =0.025*3
if Ejemplo -  == 2:
    #Flat
    def f(x):
        res=1
        return res
    Sun.field =1.2/(2.*3600.)
if Ejemplo -  == 3:
    # Parabolic
    def f(x):
        r=(x*90.0/0.025)
        res=r**2
        return res
    Sun.field =1.2/(2.*3600.)

if Ejemplo -  == 4:
    # Gaussian (Seeing)
    def f(x):
        x=np.rad2deg(x)
        seing=1.2/3600.0
        sigma=seing/2.3548
        mean = 0
        standard_deviation = sigma
        y=scipy.stats.norm(mean, standard_deviation)
        res=y.pdf(x)
        return res
    Sun.field=4*1.2/(2.0*3600.0)

Sun.fun = f
Sun.dim = 3000
Sun.num = 100000
L, M, N, X, Y, Z = Sun.rays()

Xr=np.zeros_like(L)
Yr=np.zeros_like(L)
Nr=np.zeros_like(L)
con=0
con2=0
for i in range(0,Sun.num):
    if con2==10:
        print(100.*i/Sun.num)
        con2=0

    pSource_0 = [X[i], Y[i], Z[i]]
    dCos = [L[i], M[i], N[i]]
    Telescope.Trace(pSource_0, dCos, W)
    Xr[con]=Telescope.Hit_x[-1]
    Yr[con]=Telescope.Hit_y[-1]
    Nr[con]=Telescope.SURFACE[-1]
    con=con+1
    con2=con2+1
    #Rays.push()
args=np.argwhere(Nr==2)
plt.plot(Xr[args], Yr[args], '.', c="g", markersize=1)
# axis labeling
plt.xlabel('x')
plt.ylabel('y')
# figure name
plt.title('Dot Plot')
plt.axis('square')
plt.show()
```

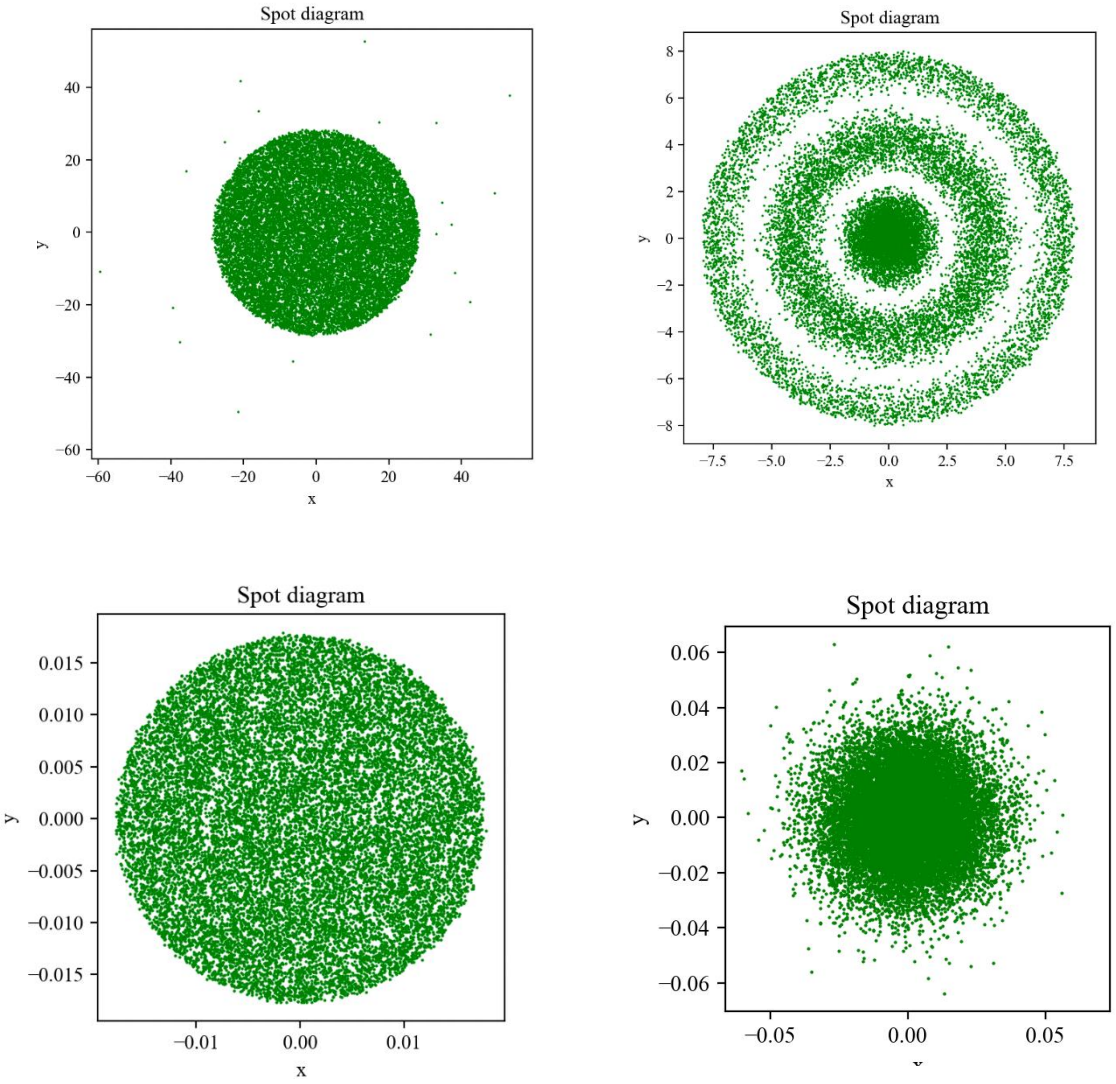


Figura A31: *Ejemplos de imágenes generadas con los rayos que provienen de 4 fuentes distintas cuya función de distribución es una función matemática.*
Arriba izquierda: distribución definida por el sol. Arriba derecha: función Sinc.
Abajo izquierda: constante. Abajo derecha: distribución Gaussiana.