
Programmation orientée objet

Version 2024

mars 12, 2025

1 Vecteur

Créer une classe Vecteur3D. Chaque vecteur aura 3 attributs : x, y, z

- Écrire une méthode norme qui renvoie la norme.
- Écrire la méthode `__add__` pour faire la somme entre deux vecteurs
- Écrire la méthode `__mul__` pour faire soit le produit par un scalaire ($2\vec{u}$) ou le produit scalaire ($\vec{u} \cdot \vec{v}$).

2 Bibliographie

Un livre est décrit par son titre, auteur et année de publication (pour faire les choses simplement). Écrire une classe Livre qui enregistre ces informations. Écrire la méthode `__repr__` et `__str__`.

Une bibliographie est une liste de livre. Écrire la classe Bibliographie qui enregistre une liste de livre (on stockera la liste de livre sous forme d'une liste qui sera un attribut de la bibliographie).

L'objectif final est de pouvoir faire ceci

```
livre1 = Livre("A very nice book", "F. Dupont", 2014)
livre2 = Livre("A very smart book", "A. Einstein", 1923)
livre3 = Livre("A very stupid comic", "D. Duck", 1937)

bibliographie = Bibliographie([book1, book2, book3])
```

Maintenant que tout est fait sous forme d'objet, on peut imaginer écrire plusieurs méthode :

- Écrire une méthode `save_to_json` pour sauvegarder la bibliographie et une fonction (ou une méthode de class) `load_from_json`.
- Écrire une méthode `filter_by_year` qui fait une nouvelle bibliographie ne contenant que les livres d'une année.
- Écrire une méthode `to_latex` ou `to_html` qui formate correctement la bibliographie. La méthode de la classe Bibliographie devra appeler une méthode pour chaque Livre.

En latex cela devra donner

```
\begin{thebibliography}{9}
\bibitem{Dupont2014}
F. Dupont (2014) \emph{A very nice book}

\bibitem{Einstein1923}
A. Einstein (1923) \emph{A very smart book}

\bibitem{Duck1937}
D. Duck (1937) \emph{A very stupid comic}
\end{thebibliography}
```

Et en HTML

```
<table>
  <thead>
    <tr> <th>Auteur</th><th>Titre</th><th>Année</th></tr>
  </thead>
  <tbody>
    <tr><td>F. Dupont</td><td>2014</td><td>A very nice book</td></tr>
    <tr><td>A. Einstein</td><td>1923</td><td>A very smart book</td></tr>
    <tr><td>D. Duck</td><td>1937</td><td>A very stupid comic</td></tr>
  </tbody>
</table>
```

Remarque : si un objet possède une méthode `_repr_html_`, alors le jupyter notebook utilisera automatiquement la représentation en HTML. Rajouter cette méthode (qui appellera `to_html`).

3 Analyse de données

Un dispositif composé d'un générateur basse fréquence et d'un oscilloscope deux voies a été utilisée pour enregistrer des données afin d'effectuer un diagramme de Bode. Ces données sont enregistrées dans un fichier au format Json : il s'agit d'une liste de dictionnaire, chaque dictionnaire correspondant à un point de mesure (paramètre du GBF et trace sur l'oscilloscope). Les données sont disponibles [ici](#)

Pour pouvoir traiter les données, nous allons créer deux classes : la première correspondra à un point de mesure (BodePoint) et la seconde à un diagramme BodeDiagram

```
class BodePoint(object):
    def __init__(self, freq, t, inp, out):
        self.freq = freq
        self.t = np.array(t)
        self.inp = np.array(inp)
        self.out = np.array(out)

class BodeDiagram(object):
    def __init__(self, list_of_bode_point):
        self._list_of_bode_point = list_of_bode_point

    @classmethod
    def from_json(cls, list_):
        return cls([BodePoint.from_json(dct) for dct in list_])
```

Nous utiliserons les fonctions suivantes pour ajuster les courbes par une sinusoïde

```
def fit_function(t, freq, a, b, offset):
    phi = 2*np.pi*freq*t
    return a*np.sin(phi) + b*np.cos(phi) + offset

def fit_sinusoid(t, y, freq):
    amp = np.std(y)
    p0 = [freq, amp, amp, 0]
    popt, _ = curve_fit(fit_function, t, y, p0=p0)
    return {'freq':popt[0], 'a':popt[1], 'b':popt[2], 'offset':popt[3]}
```

1. Écrire une méthode de class `from_json(self, dct)` de `BodePoint` afin de charger les données depuis un dictionnaire.
2. Écrire une méthode `plot_data(self, ax1, ax2)` afin de tracer les courbes sur deux axes (éventuellement identiques).

3. Écrire une méthode `fit_data` qui renvoie les paramètres de fit pour les deux courbes. Modifier la méthode `plot_data` pour afficher le fit.
4. Écrire une méthode `get_complex_gain` qui renvoie le gain complexe du circuit (i.e. le rapport $(a_i + jb_i)/(a_o + jb_o)$).
5. On voit que l'on effectue deux fois le fit si jamais on appelle `plot_data` puis `get_complex_gain`. Comment faire pour ne calculer le fit qu'une seule fois ?
6. Ecrire enfin les méthodes `get_complex_gain` et `get_frequencies` de la classe `BodeDiagram`.

4 Impédance complexe d'un circuit bibolaire

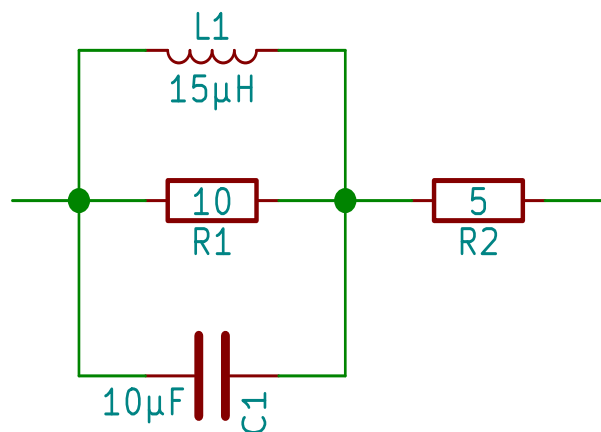


FIG. 1 – Exemple de circuit

Objectif : faire comprendre à Python ce type de circuit pour pouvoir ensuite faire des calculs. Ici, on demandera de calculer l'impédance complexe à une fréquence donnée.

Il y a plusieurs objets de nature différente donc de classe différente (résistance, condensateur, circuit parallèle, ...). Mais ces objets sont tous des circuits bibolaires. Tous ces objets devront mettre en oeuvre une méthode pour calculer leur impédance à une fréquence donnée.

Code final en Python (objectif à atteindre pour que l'objet soit le plus simple à utiliser)

```
R1 = Resistor(10)
R2 = Resistor(5)
L1 = Inductor(15E-6)
C1 = Capacitor(10E-6)

circuit = Serial(Parallel(L1, R1, C1))
circuit = R2 + (L1|R1|C1) # Notation simple

print(circuit.impedance(50))
```

Ici, le `|` représente une structure parallèle et le `+` une structure série.

La structure de classe sera la suivante

```
class BipolarCircuit(object):
    pass

class Combination(BipolarCircuit):
    pass

class Serial(Combination):
```

(suite sur la page suivante)

```

pass

# idem for parallel

class Device(BipolarCircuit):
    pass

class Resistor(Device):
    pass

```

1. Créer les méthode `__init__` des classes ci dessus and que le `__repr__`. Rajouter la classe `Capacitor`, `Inductor`, `Parallel`.
2. Définir les méthodes `__or__` et `__add__` afin que l'on puisse utiliser la notation simplifiée.
3. Ecrire les méthodes `def impedance(self, frequency)` afin que l'on puisse connaitre l'impédance de chaque circuit. Attention : il y aura autant de méthodes que de cas spécifiques.

5 Système de calcul formel

Note : Cet exercice est à but purement pédagogique. Pour utiliser un système de calcul formel sous Python, la librairie `sympy` existe et fonctionnera bien mieux que ce que l'on va faire !

L'objectif de ce TD est de réaliser un système de calcul formel qui permettra de manipuler des expressions algébriques simples et de réaliser des opérations simples. Par exemple, on souhaite pouvoir effectuer

```

x = Symbol('x')
y = Symbol('y')

s = 2*x*y + sin(x)*y

print(s.diff(x)) # Dérivée par rapport à x

```

Chaque expression sera représentée par un arbre. Les feuilles de l'arbre seront soit les symboles soit les constantes numériques. Les noeuds seront des fonctions à un ou plusieurs argument (sinus, somme, opposé, ...). Le nom de la classe du noeud désignera la fonction. Les « enfants » du noeud seront les arguments de la fonction. Par exemple l'expression ci dessus correspondra à l'objet suivant

```

# sA : 2*x*y
sA = Prod(Prod(Number(2), Symbol('x')), Symbol('y'))
# sB : sin(x)*y
sB = Prod(Sin(Symbol('x')), Symbol('y'))

s = Sum(sA, sB)

```

5.1 Structure du programme

Voici la structure de base

```
class Expr(object):  
    pass  
  
class Node(Expr):  
    pass  
  
class Leave(Expr):  
    pass
```

Pour les feuilles

```
class Symbol(Leave):  
    pass  
  
class Number(Leave):  
    pass
```

Ensuite on définit les fonctions

```
class Function(Node):  
    """ Function with an arbitrary number of arguments """  
    pass
```

Les opérateurs sont des fonctions comme les autres, mais elle seront simplement affichées différemment

```
class BinaryOperator(Function):  
    pass  
  
class Sum(BinaryOperator):  
    pass  
# Idem pour Sub, Div, Prod, Pow  
  
class UnitaryOperator(Function):  
    pass  
  
class Neg(UnitaryOperator):  
    pass
```

Les fonction mathématiques, qui prennent un seul argument

```
class MathFunction(Function):  
    pass  
  
class Sin(MathFunction):  
    pass
```

5.2 Questions

On va procéder étape par étape. Il sera plus facile de commencer par les feuilles avant d'écrire la structure globale.

1. Ecrire le `__init__` de la classe `Symbol` et `Number`
2. Ecrire une méthode `display` sur ces classes afin de renvoyer une chaîne de caractère contenant le symbole ou le nombre
3. Ecrire le `__init__` de la class `Sin` ainsi que le `display`. Le `display` devra appeler le `display` de l'argument. Par exemple ceci devra fonctionner

```
>>> x = Symbol('x')
>>> Sin(x).display()
sin(x)
>>> Sin(Sin(x)).display()
sin(sin(x))
```

4. Généraliser le `__init__` et le `display` de `Sin` afin de le mettre dans la class `MathFunction`. On rajoutera un attribut de classe à chaque sous classe de `MathFunction`

```
class Sin(MathFunction):
    fonction_name = 'sin'
```

5. Faire de même pour les opérateurs binaires. On pourra commencer par simplement le faire pour `Sum`, puis généraliser avec un attribut de classe

```
class Sum(BinaryOperator):
    operator_name = '+'
```

6. A ce stade quelque chose comme ceci devrait fonctionner

```
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> Sum(x, Sin(Prod(x, y)))
```

Rajouter les méthodes `__add__`, `__mul__`, etc à la classe `Expr` afin de pouvoir écrire :

```
>>> x + Sin(x*y)
```

7. Ecrire les méthodes `evaluate` afin de calculer la valeur numérique d'une expression. Cette méthode fonctionnera de la sorte :

```
>>> expr = x + Sin(x*y)
>>> expr.evaluate(x=1, y=3)
```

On aura donc le protocole suivant

```
def evaluate(self, **kwd):
    pass
```

Le dictionnaire `kwd` sera passé récursivement jusqu'aux feuilles et sera utilisé pour évaluer les symboles.

Les opérateurs binaires numériques sont définis dans le module `operator` et les fonctions dans le module `math`. Afin de factoriser le code, on rajoutera donc simplement un attribut de classe du type `operator_function = operator.add` pour les opérateurs binaires et `math_function = math.sin` pour les fonctions.

8. Maintenant que vous avez compris le principe, il devrait être facile d'écrire une méthode `diff` qui effectue la dérivée par rapport à une variable !
9. Reste à simplifier les expressions. Une technique consiste à créer des règles de simplifications sous forme de méthode que l'on regroupe ensuite dans une liste

```

class Sum(BinaryOperator):
    operator_name = '+'
    operator_function = operator.add

    def simplification_de_deux_nombres(self):
        if isinstance(self.arg1, Number) and
            isinstance(self.arg2, Number):
            return Number(self.arg1.value + self.arg2.value)

    def simplification_addition_avec_zero(self):
        pass

liste_simplification = ['simplification_de_deux_nombres',
                        'simplification_addition_avec_zero']

```

Ensuite, il faut réussir à appeler correctement et de façon recursive ces méthodes...

10. Pour l’affichage des opérateurs binaires, les règles de priorité peuvent être utilisées pour éviter de mettre trop de parenthèses. Par exemple, dans le cas $a*(b+c)$, la multiplication appelle le display de l’addition. Comme elle est prioritaire, l’addition va renvoyer le résultat avec des parenthèses. Dans le cas inverse $a + b*c$, c’est inutile. Il faut donc que le display d’un opérateur passe sa priorité à ses enfants lors de l’appel de display. Implémenter ce principe.

6 Module and package

Nous souhaitons enregistrer les constantes fondamentales de la physique dans un package.

- Créer le package `constants` avec deux modules : le premier `fundamental` et le second `atomic_mass`. Ils s’utiliseront de la façon suivante

```

from constants.fundamental import mu_0, hbar, e, c
from constants.atomic_mass import rubidium_87

```

Voici les valeurs (en SI) :

$$c = 299792458; e = 1.60217663 \times 10^{-19}; h = 6.62607015 \times 10^{-34}; \hbar = h/2\pi;$$

$$\mu_0 = 1.25663706212 \times 10^{-6}; \epsilon_0 = 1/\mu_0 c^2; G = 6.67430 \times 10^{-11};$$

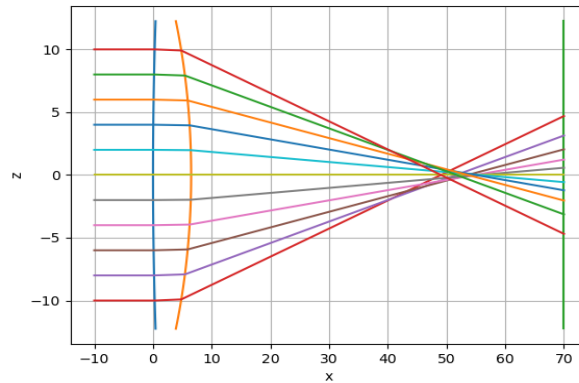
Pour le masses atomique :

$$M(^{87}\text{Rb}) = 86.909180527 m_u; M(^{85}\text{Rb}) = 86.909180527 m_u;$$

- Modifier le `__init__.py` pour que `from constants import mu_0, h, e` fonctionne.
- Créer une fichier `setup.py` et installer le (en utilisant `pip install -e . --user`). Tester depuis un autre repertoire.

7 Traceur de rayons

Un traceur de rayon est un programme qui permet de calculer la propagation d’un faisceau lumineux à travers un système optique. Des programmes commerciaux (tels que Code V, Zemax ou OSLO) permettent de faire de tels calculs. Dans ce TP nous allons programmer un traceur de rayons avec Python. Identifions dans un premier temps les objets élémentaires dans ce problème. Il faut pouvoir décrire des dioptries et des rayons. Nous allons donc commencer par écrire une classe `Diotre`, et une classe `Ray`. Puis nous introduirons des méthodes permettant l’interaction entre ces classes. Dans la suite les variables à utiliser dans Python seront notées en italique. Nous utiliserons les millimètres pour mesurer les longueurs.



7.1 Dioptries sphériques

Un dioptré est une interface entre deux milieux d'indices différents. Dans ce TP nous allons nous limiter aux dioptries sphériques. Un dioptré est caractérisé par son intersection avec l'axe optique : `position`, par son rayon de courbure : `radius_of_curvature`, par les indices de réfraction de part et d'autre : `refractive_index` (tuple à deux éléments, le premier étant l'indice à gauche et le second l'indice à droite), par son diamètre (extension suivant z) : `diameter`.

- Créer la classe `Dioptré`, avec comme argument obligatoire les variables `position`, `radius_of_curvature`, `refractive_index`, et comme argument optionnel `diameter`, dont vous fixerez la valeur par défaut à 25.4 mm (1 pouce). On pourra utiliser une `dataclass`.
- Créer une propriété `center` qui renvoie la position du centre de la sphère du dioptré.

Le dioptré étant sphérique, il a pour équation $z^2 + (x - c)^2 = R^2$, où R est le rayon de courbure et c le centre. La position du dioptré est donc $x_0 = c + R$, le rayon de courbure pouvant être positif ou négatif, suivant la concavité du dioptré. Avec cette convention, on a

$$x = c - \text{sgn}(R) \sqrt{R^2 - z^2}.$$

- Ajouter une méthode `shape` qui prend en entrée une valeur de z et renvoie la valeur x donnée par l'équation précédente. Cette méthode devra fonctionner avec des tableaux `numpy`.
- Ajouter une méthode `plot` qui trace le dioptré sur un axe.

7.2 Rayon lumineux

Un rayon lumineux peut être décrit par un point \vec{p} (vecteur à deux composantes) et par un vecteur directeur \vec{k} (à deux composantes également).

- Créer la classe `Ray`, ayant pour argument obligatoire `starting_point` (le point \vec{p}), `direction` (le vecteur \vec{k}), et pour argument optionnel `refractive_index` dont la valeur par défaut sera l'objet `None`. On s'assurera que le point \vec{p} et le vecteur \vec{k} sont enregistré sous forme de tableaux `numpy`.
- Comme tous les vecteurs d'ondes sont proportionnels à l'indice de réfraction du milieu, on peut choisir de normaliser les vecteurs d'ondes par l'indice du milieu (c'est à dire que l'on prend $2\pi/\lambda = 1$). Ajouter une méthode `normalize_direction_to_refractive_index` qui permet de normaliser \vec{k} par l'indice de réfraction.
- Normaliser \vec{k} par l'indice de réfraction dans l'initialiseur, uniquement si l'indice de réfraction est passé comme argument.

7.3 Loi de Snell et Descartes

Nous avons maintenant deux classes pour décrire les objets élémentaire du problème considéré. Il faut écrire des méthodes de ces classes qui décrivent la réfraction des rayons par les dioptrés. Il y a plusieurs manières de procéder. Ici, on propose d'ajouter ces méthodes à la classe `Dioptré`.

- Ajouter une méthode `get_refracted_ray` à la classe `Dioptré`. Cette méthode prendra comme argument la variable `incident_ray`, qui devra être le rayon incident (une instance de la classe `Ray`). Dans un premier temps, ajouter l'unique instruction `pass` à cette méthode.

L'objectif est que cette méthode renvoie le rayon réfracté. Pour commencer il faut d'abord déterminer le point d'intersection entre le rayon incident et le dioptré. Nous allons écrire une méthode spécifique pour cela. L'équation paramétrique du rayon incident est $\vec{p}(0) + t\vec{k}_i$. On cherche le paramètre t_i , tel que $\vec{p}(t_i) = \vec{p}(0) + t_i\vec{k}_i$ soit le point d'intersection entre le rayon incident et le dioptré. Dans le triangle grisé, on doit donc résoudre :

$$\begin{aligned}\|\vec{p}(t) - \vec{c}\|^2 &= R^2 \\ \|\vec{p}(0) - \vec{c} + t\vec{k}_i\|^2 &= R^2 \\ \alpha t^2 + \beta t + \gamma &= 0,\end{aligned}$$

avec

$$\begin{aligned}\alpha &= \|\vec{k}_i\|^2 \\ \beta &= 2\vec{k}_i \cdot (\vec{p}(0) - \vec{c}) \\ \gamma &= \|\vec{p}(0) - \vec{c}\|^2 - R^2.\end{aligned}$$

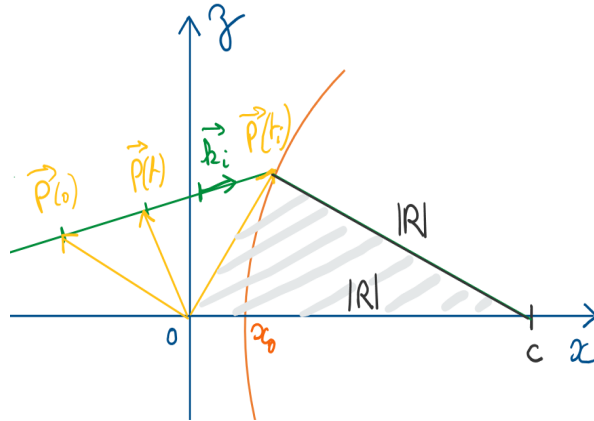


FIG. 2 – Intersection entre un rayon et un dioptré sphérique. Le dioptré est représenté en orange, le rayon incident en vert.

- Ajouter une méthode `intersection` à la classe `Dioptré`, qui prend en argument un rayon, et renvoie un tableau numpy à deux composantes contenant les coordonnées de $\vec{p}(t_i)$. Dans cette méthode, il faudra donc déterminer les racines du trinôme du second degré ci-dessus.

On utilise ensuite la loi de Snell et Descartes : la composante parallèle du vecteur d'onde (projection du vecteur d'onde sur la tangente au dioptré, au point d'incidence) est conservée lors de la réfraction. On peut déterminer cette composante parallèle en soustrayant à \vec{k}_i sa composante perpendiculaire :

$$\vec{k}_{\parallel} = \vec{k}_i - \vec{k}_{i,\perp} = \vec{k}_i - (\vec{k}_i \cdot \vec{n}) \cdot \vec{n}$$

On a introduit \vec{n} , le vecteur normal à la tangente au point d'incidence (cf. figure ci-contre). Par ailleurs, sur la figure précédente on voit que

$$\vec{n} \propto \vec{p}(t_i) - \vec{c}.$$

La composante perpendiculaire du vecteur d'onde du rayon réfracté s'obtient ensuite en utilisant le fait que $\|\vec{k}_r\| = n_2$, où n_2 est l'indice de réfraction du côté droit du dioptré. On a donc

$$\vec{k}_{r,\perp} = \sqrt{n_2^2 - \|\vec{k}_{\parallel}\|^2} \vec{n}.$$

On en déduit le vecteur d'onde réfracté

$$\vec{k}_r = \vec{k}_{\parallel} - \text{sgn}(R)\vec{k}_{r,\perp}.$$

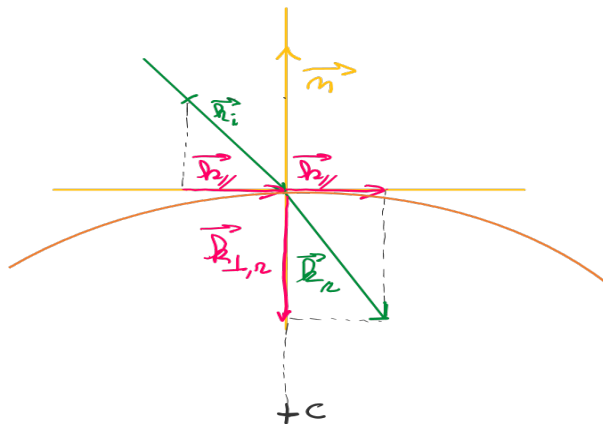


FIG. 3 – Réfraction du rayon incident sur le dioptré sphérique. Par rapport à la figure précédente, une rotation a été effectuée pour faciliter le tracé et la visualisation.

- **Compléter la méthode `get_refracted_ray` :**
 - Déterminer $\vec{p}(t_i)$ en appelant la méthode `col{intersection}`.
 - Déterminer le vecteur unitaire \vec{n} .
 - Calculer \vec{k}_{\parallel} .
 - Calculer $\vec{k}_{r,\perp}$.
 - En déduire \vec{k}_r , et retourner l'objet rayon réfracté.

7.4 Trajet complet d'un rayon et système optique

L'essentiel du travail est fait. Pour faciliter la programmation du tracé de rayon nous pouvons écrire une classe `col{OpticalSystem}` pour décrire un système optique comprenant plusieurs dioptrés. Cette classe aura pour attribut une liste de Dioptré. De même afin de décrire un trajet complet de rayons à travers les différents dioptrés, on peut regrouper les différents rayons dans une liste. }

- Créer la classe `OpticalSystem`.
- Ajouter à la classe `OpticalSystem` une méthode `plot` qui trace les dioptrés de la liste.
- Créer la classe `RayList`.
- Ajouter à la classe `RayList` une méthode `plot` qui trace des segments entre les points de départ des rayons de la liste.
- Créer une class `Screen` qui est un dioptré plan ne faisant rien. On changera la class `Dipotre` en `SphericalDioptré` et on fera hériter `Screen` et `SphericalDioptré` d'une classe vide `Dipotre`.
- Ajouter à la classe `OpticalSystem` une méthode `ray_tracing` qui prend comme argument un rayon (instance de la classe `Ray`) incident sur le système optique, et qui renvoie une liste de rayons (instance de la classe `Raylist`) correspondant au trajet de la lumière à travers les différents dioptrés du système optique.
- Tester le programme en générant une figure similaire à celle donnée en introduction.
- (Bonus) Ecrire les méthodes spéciales afin que le code suivant puisse fonctionner (méthode `__add__` pour assembler ensemble dioptrés ou systèmes optiques, `__matmul__` pour déplacer un dioptré ou un système optique, `__neg__` pour inverser le sens d'un dioptré ou système optique) :

```
n_LAH64 = 1.77694
n_SF11 = 1.76583
n_BK7 = 1.5112
n_air = 1.0002992
```

(suite sur la page suivante)

```
#Reference from https://www.thorlabs.com/
s1 = Dioptre(0, -4.7 , (n_air, n_SF11), diameter=3)
s2 = Dioptre(1.5, 1E10, (n_SF11, n_air), diameter=3)
LC2969 = s1 + s2 # Return an OpticalSystem

s1 = Dioptre(0, 1E10 , (n_air, n_BK7))
s2 = Dioptre(4.1, -38.6, (n_BK7, n_air))
LA1608 = s1 + s2 # Return an OpticalSystem

system = -LC2969 + LA1608@50
```