

# Corrigé épreuve informatique - Centrale - 2021

C. Antonini

mail : [cantonini@stanislas-cannes.com](mailto:cantonini@stanislas-cannes.com)

Les commentaires constructifs (notamment sur les questions de complexité) seront les bienvenus

On commence par les importations citées dans l'énoncé.

```
[1]: import numpy as np
import math
```

## I/ Géométrie

### Question 1

```
[2]: def vec(A, B):
return B-A
```

### Question 2

```
[3]: def ps(v1, v2):
s = 0
for i in range(len(v1)):
s += v1[i]*v2[i]
return s
# On pourrait aussi voir le résultat comme la somme des coefficients
# du produits de Hadamard :
# def ps(v1, v2):
#     v = v1*v2
#     s = 0
#     for x in v:
#         s += x
#     return s
# Vu l'annexe p8, on pouvait même se permettre de taper
# def ps(v1, v2):
#     v = v1*v2
#     return v.sum()
# Ou carrément :
# def ps(v1, v2):
#     return np.inner(v1, v2)
# Car (je cite) "calcule la somme des produits terme à terme etc." :
# c'est le produit scalaire canonique
```

### Question 3

```
[4]: def norme(v):  
      return math.sqrt(ps(v, v))
```

### Question 4

```
[5]: def unitaire(v):  
      return (1/norme(v))*v  
      # On peut aussi directement diviser !  
      # def unitaire(v):  
      #     return v/norme(v)
```

### Question 5

Tapons déjà les fonctions !

```
[6]: def pt(r, t):  
      # assert t >= 0  
      (S, u) = r  
      return S + t * u
```

Renvoie le point  $M$  du rayon  $r$  de paramètre  $t$  ( $t \geq 0$ ), i.e. le vecteur défini par  $\vec{SM} = t.\vec{u}$ .

Une erreur est renvoyée si  $t < 0$  (c'est-à-dire si "l'assertion" (= la supposition)  $t \geq 0$  n'est pas vérifiée).

```
[7]: def dir(A, B):  
      return unitaire(vec(A, B))
```

Renvoie le vecteur unitaire dirigeant la droite  $(AB)$ , qui est aussi le vecteur du rayon issu de  $A$  passant par  $B$ .

```
[8]: def ra(A, B):  
      return A, dir(A, B)
```

Renvoie le rayon issu de  $A$  et passant par  $B$ .

### Question 6

Le rayon de la sphère de centre  $A$  et passant par  $B$  est la distance  $AB = \|\vec{AB}\|$ , d'où :

```
[9]: def sp(A, B):  
      r=norme(vec(A, B))  
      return (A,r) # Parenthèses optionnelles ici
```

### Question 8

Il faut choisir la plus petite des 2 racines de l'équation s'il y a des solutions, les deux sont de même signe car on suppose que l'origine du rayon n'est pas dans la boule ( $c$ =produit des racines est positif). Ainsi il faut prendre la solution avec  $-\sqrt{\Delta}$  et pas  $+\sqrt{\Delta}$ ...

Normalement, si on lance la fonction, les racines (s'il y en a) sont positives. J'ai néanmoins précisé `if t<0: return None` pour le cas où l'on n'atteint pas la sphère dans la 1/2 droite du rayon. Il faudrait modifier la réponse à la Q25 pour éviter cela.

```
[10]: def intersection(r, s):
    A, u = r
    C, R = s # ATTENTION, PB avec l'énoncé : le nom du rayon lumineux
            # = r = nom du rayon de la sphère
            # J'appelle donc R le rayon de la sphère !
    # définition des coeffs de l'équation (a=1, je ne le nomme pas)
    CA = vec(C, A)
    b = 2*ps(u, CA)
    c = ps(CA, CA) - R**2
    Delta = b**2 - 4*c
    if Delta < 0:
        return None
    else:
        t = (-b-math.sqrt(Delta))/2
        # Pour Q25 j'évite le cas t<0, mais c'est
        # car il me manque un appel à la fonction au_dessus ou visible
        # dans cette question 25
        if t<0:
            return None
        else:
            return pt(r, t), t
```

## II/ Optique

### II.A - Visibilité

#### Question 9

le plan tangent à la sphère au point P est le plan passant par P et orthogonal au vecteur CP, donc M appartient à ce plan ssi les vecteurs  $\vec{PM}$  et  $\vec{CP}$  sont orthogonaux Ce plan est donc d'équation  $\vec{PM} \cdot \vec{CP} = 0$  (. pour produit scalaire !) Les demi-espaces stricts séparés par ce plan sont donc d'(in)équations  $\vec{PM} \cdot \vec{CP} > 0$  et  $\vec{PM} \cdot \vec{CP} < 0$  Comme  $\vec{PC} \cdot \vec{CP} = -\|\vec{PC}\|^2 < 0$ , le demi-espace au-dessus de la sphère, ne contient pas C, son inéquation est donc  $\vec{PM} \cdot \vec{CP} > 0$ .

#### Question 10

On en déduit la fonction :

```
[11]: def au_dessus(s, P, src):
    C, R = s
    PM = vec(P, src)
    CP = vec(C, P)
    return ps(PM, CP)>0 # On pourrait aussi mettre du large...
```

## Question 11

```
[12]: def visible(obj, j, P, src):
    if au_dessus(obj[j], P, src) == False:
        return False # Ainsi, on ne va pas plus loin
                        # si la sphère n'est pas visible.
    # On parcourt les autres sphères, et
    # si on en rencontre une plus proche, on renvoie False
    for i in range(len(obj)):
        if i!=j:
            r = ra(src, P)
            inter=intersection(r, obj[i])
            if inter != None:
                if inter[1] < norme(vec(src, P)):
                    return False
    # Si on arrive ici, c'est que tout est bon !
    return True
```

## II.B - Diffusion

### Question 12

```
[13]: def couleur_diffusee(r, Cs, N, kd):
    S, u = r
    co = ps(-u, N) # calcul de cos(theta) ATTENTION, l'orientation de u
                  # (vecteur directeur du rayon)
                  # opposée au vecteur qui permet de calculer le cos
    return co*(Cs*kd)
```

## II.C - Réflexion

### Question 13

En notant  $\vec{v}$  le projeté orthogonal de  $-\vec{u}$  (même pb que ci-dessus) sur la droite dirigée par  $\vec{N}$ ,  
on a  $\vec{w} + -\vec{u} = 2\vec{v}$ .

D'où la fonction

```
[14]: def rayon_reflechi(s, P, src):
    u = dir(src, P)
    C, r = s
    N = dir(C, P) # Vecteur normal (sortant) à la sphère en P
    v = ps(N, -u)*N
    w = u+2*v
    return (P,w)
```

### III/ Enregistrement des scènes - requêtes SQL

#### Question 14

```
SELECT sc_nom FROM Scene WHERE EXTRACT(year FROM sc_creation)="2021"
```

#### Question 15

```
SELECT sc_id, COUNT(so_id) FROM Source GROUP BY sc_id
```

#### Question 16

```
SELECT ob_id, ob_x, ob_y, ob_z, sp_id FROM Scene JOIN Objet ON Scene.sc_id=Objet.sc_id  
JOIN Sphere ON ob_id=sp_id WHERE sc_nom="woodbox"
```

#### Question 17

```
SELECT s1.ob_id, so_id, s2.ob id FROM Objet as s1 JOIN Objet as s2 ON s1.sc_id=s2.sc_id  
JOIN Scene ON s1.sc_id=Scene.sc_id JOIN Source ON Scene.sc_id=Source.sc_id WHERE  
sc_nom="woodbox" AND Occulte(Scene.sc_id, s1.ob_id, so_id, s2.ob_id)
```

### IV/ Lancer de rayons

#### IV.A - Ecran

#### Question 18

```
[15]: def grille(i, j):  
    x = (Delta/N)*(j-N/2)  
    # Attention, sur fig 4, lorsque i augmente, y diminue.  
    y = (Delta/N)*(N/2-i)  
    return np.array([x, y, 0])
```

#### Question 19

```
[16]: def rayon_ecran(omega, i, j):  
    u = dir(omega, grille(i, j))  
    return (omega, u)
```

## IV.B - Couleur d'un pixel

### Question 20

```
[17]: def interception(r, eviter=-1):
    # J'ai rajouté le 2ème paramètre pour la Question 25,
    # on ne cherche pas l'intersection avec l'objet d'indice eviter...
    x = None
    minim = np.inf
    for i in range(len(Objet)):
        if i != eviter:
            t = intersection(r, Objet[i])
            if t != None:
                if t[1] < minim:
                    x = t[0], i
                    minim = t[1]
    return x
```

### Question 21

```
[18]: def couleur_diffusion(P, j):
    coul = np.array([0,0,0])
    for i in range(len(Source)):
        if visible(Objet, j, P, Source[i]):
            coul = coul + couleur_diffusee(ra(Source[i], Objet[j][0]),
            →ColSrc[i], dir(Objet[j][0], P), KdObj[j])
    return coul # si aucune source visible, ça renvoie le [0,0,0] initial,
                # qui correspond bien au noir
```

## IV.C - Constitution de l'image

### Question 22

```
[19]: def lancer(omega, fond):
    im = np.ndarray((N,N,3))
    for i in range(N):
        for j in range(N):
            r = rayon_ecran(omega, i, j)
            x = interception(r)
            if x == None:
                im[i, j] = fond
            else:
                im[i, j] = couleur_diffusion(x[0], x[1])
    return im
```

Je trouve que ça vaut la peine d'essayer sur un exemple concret !

```
[20]: # Définition des objets (un par un, avec .append)
Objet = []
Objet.append((np.array([0,5.5,-6]), 3))
Objet.append((np.array([0.5,1.5,-2]), 1))
Objet.append((np.array([2,-3,-3]), 3))
Objet.append((np.array([-4,0,-3]), 2))
# Définition des couleurs de diffusion,
# objet par objet
KdObj = []
KdObj.append(np.array([0,1,0]))
KdObj.append(np.array([1,1,1]))
KdObj.append(np.array([0,0.8,1]))
KdObj.append(np.array([1,1,0]))
# Définition des sources lumineuses (positions)
Source = []
Source.append(np.array([-2,0,3]))
Source.append(np.array([3,0,3]))
# Définition des couleurs émises,
# source par source
ColSrc = []
ColSrc.append(np.array([1,1,1]))
ColSrc.append(np.array([0,0,1]))
# Résolution de l'image et dimension de la grille
N = 1024
Delta = 10
```

Je crée une fonction (non demandée) pour sauvegarder l'image dans un fichier dont je passe aussi le nom en paramètre :

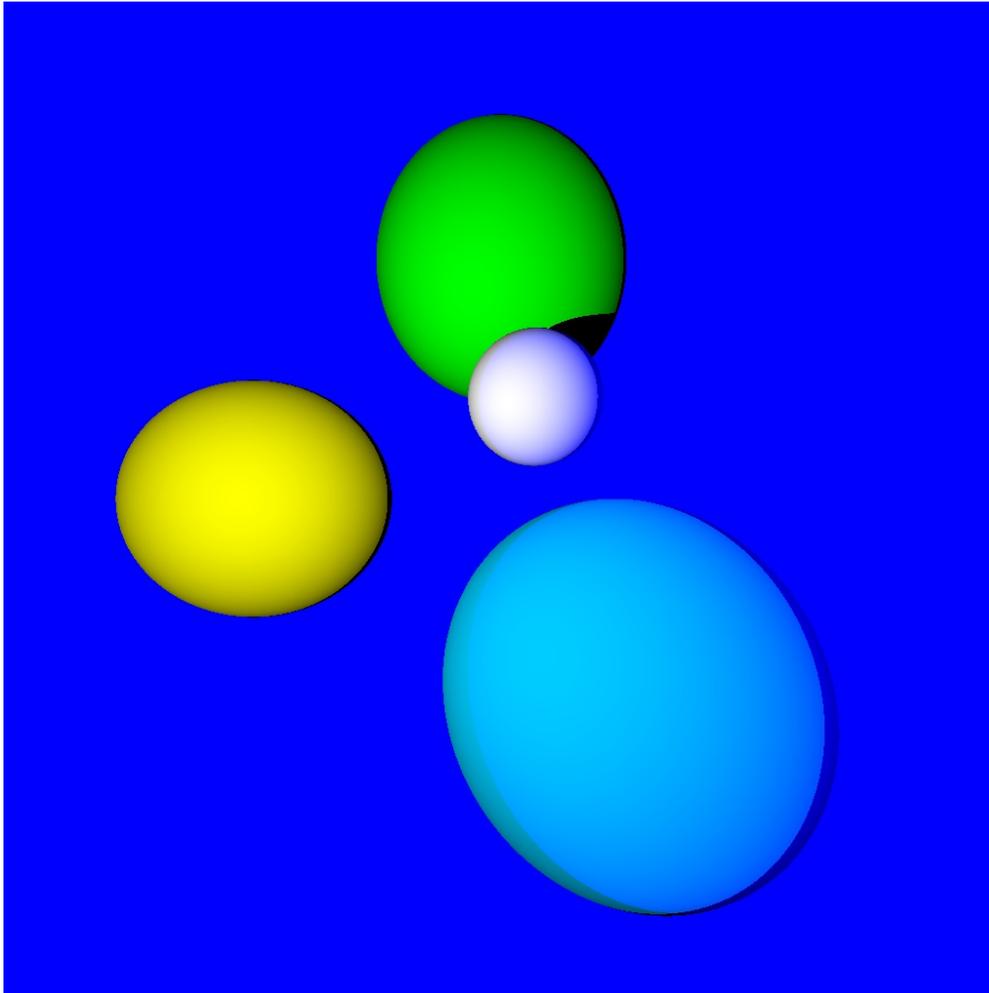
```
[21]: import matplotlib.pyplot as plt

def sauvegarde(im, nom_fichier):
    # J'applique un traitement pour que les valeurs (des niveaux r, v, b)
    # soient des entier entre 0 et 255 plutôt que des flottants entre 0 et 1
    # (il était aussi possible que l'on sorte de l'intervalle)
    #
    # Enfin, le type uint8 de numpy correspond justement
    # à un entier dans [[0,255]] codé sur un octet...
    def f(a):
        return max(0,min(255,round(255*a)))
    im2 = np.zeros((N,N,3),dtype=np.uint8)
    for i in range(N):
        for j in range(N):
            for c in range(3):
                im2[i,j,c]=f(im[i,j,c])
    plt.imsave(nom_fichier, im2)
    # Je pourrais visualiser, mais je ne l'active pas :
    # plt.imshow(im2)
```

Et je lance mes procédures :

```
[22]: im1 = lancer(np.array([0,0,4]), np.array([0,0,1]))  
sauvegarde(im1, "4Billes_2sources_sans_reflexion.jpg")
```

Voici mon résultat :



## IV.D - Complexités

### Questions 23 et 24

On lance un rayon pour chacun des  $N^2$  pixels de l'image.

Pour chaque rayon, on recherche l'objet éventuellement intercepté, ce qui nécessite  $n_o$  passages. Il faut alors calculer la couleur diffusée en ce point (s'il est sur un objet) ce qui nécessite de calculer la couleur reçue de chaque source ( $n_s$  itérations) mais de vérifier si aucun autre objet ne le cache, donc  $n_o$  itérations).

La complexité dans le pire des cas est  $O(N^2 \cdot n_o^2 \cdot n_s)$ .

Si un objet cache tous les autres, la complexité dans le meilleur des cas diminue alors en  $N^2 \cdot n_o \cdot n_s$ .

## V/ Améliorations

### V.A - Prise en compte de la réflexion

#### Question 25

Notons qu'une version récursive sur rmax serait aussi très intéressante à envisager !

```
[23]: # Une version récursive sur rmax serait aussi intéressante à envisager !
def reflexions(r, rmax):
    R = []
    I = interception(r)
    while (len(R)<rmax+1) and (I != None):
        R.append(I)
        r=rayon_reflechi(Objet[I[1]], I[0], r[0])
        # J'ai modifié interception pour éviter (à cause des erreurs d'arrondis)
        # que l'on retombe sur l'objet lui-même !
        I = interception(r, I[1])
    if R != []:
        return R
    else:
        return None
# Le else... est inutile, en fait...
# Il faudrait que j'utilise visible ou au_dessus dans cette fonction,
# j'ai préféré rajouter une condition dans interception
```

#### Question 26

```
[24]: def couleur_percue(r, rmax, fond):
    R = reflexions(r, rmax)
    if R == None:
        return fond
    else:
        C = np.array([0,0,0]) # C'est le Cr+1=0 ...
        for I in R[::-1]: # On commence par la fin
            # puisque la relation de récurrence
            # donne Ck en fonction de Ck+1...
            C = couleur_diffusion(I[0], I[1])+KrObj[I[1]]*C
        return C
```

### Question 27

```
[25]: def lancer_complet(omega, fond, rmax):  
    im = np.ndarray((N,N,3))  
    for i in range(N):  
        for j in range(N):  
            r = rayon_ecran(omega, i, j)  
            im[i, j] = couleur_percue(r, rmax, fond)  
    return im
```

Je reprends le même exemple que précédemment, en rajoutant les coefficients de réflexion :

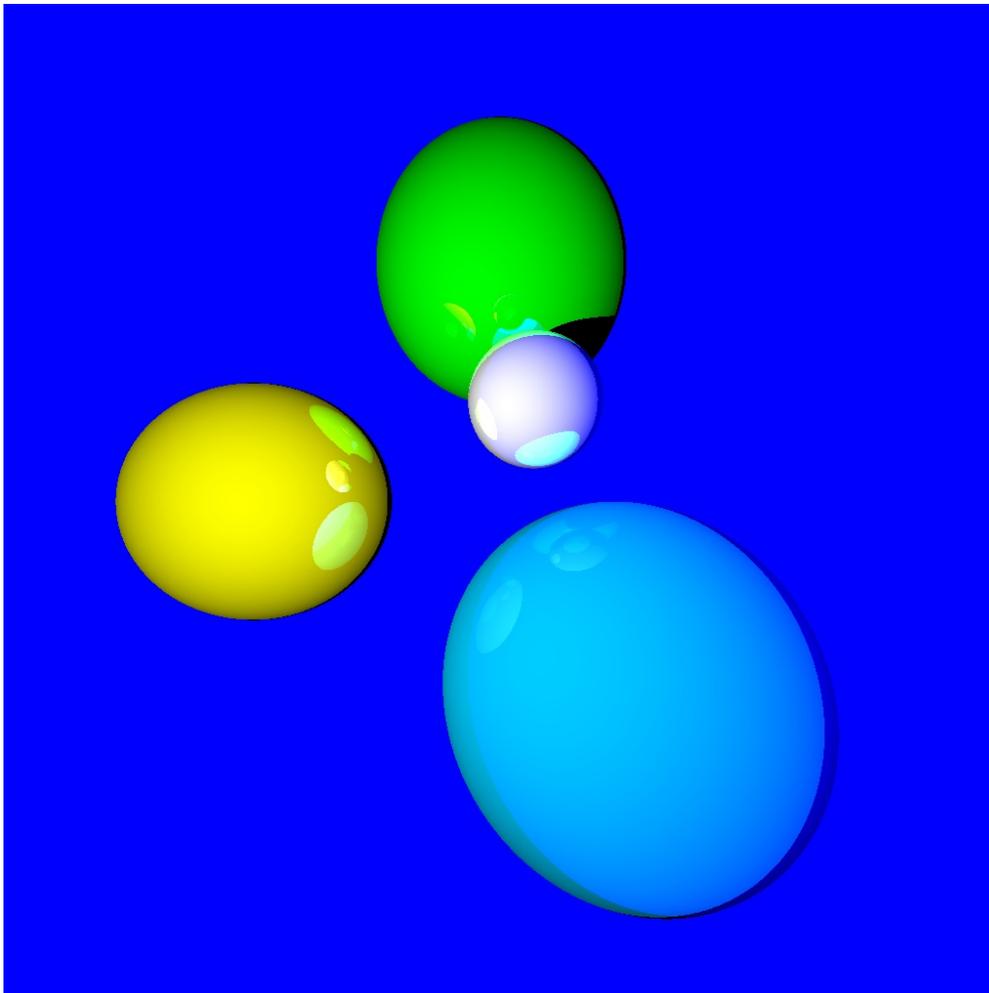
```
[26]: KrObj=[0.8, 1, 0.1, 0.4]
```

Je peux alors utiliser la procédure lancer\_complet...

J'ai choisi d'aller jusqu'à 10 réflexions !

```
[27]: im_reflet1 = lancer_complet(np.array([0,0,4]), np.array([0,0,1]), 10)  
  
sauvegarde(im_reflet1, "4Billes_2sources_avec_10_reflexions_max.jpg")
```

Voici le nouveau résultat :



Je fais un 2ème exemple plus simple, avec lumière à la verticale du centre, et uniquement 2 objets (un vert et un rouge), dont un qui reflète à 100% et l'autre pas du tout.

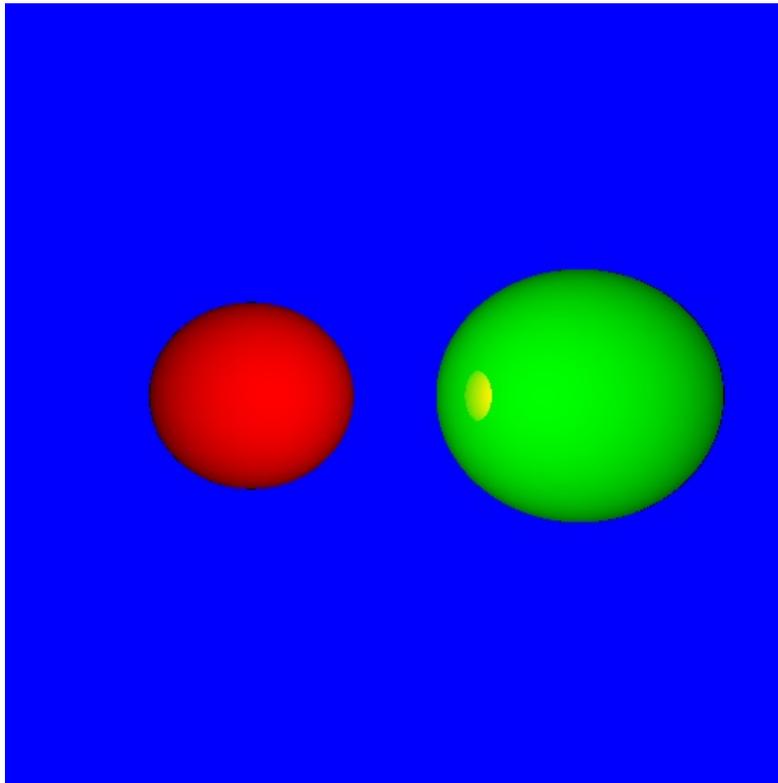
```
[28]: Objet = []
Objet.append((np.array([4,0,-4]), 3))
Objet.append((np.array([-3,0,-3]), 2))
KdObj = []
KdObj.append(np.array([0,1,0]))
KdObj.append(np.array([1,0,0]))
KrObj = [1, 0]
Source = []
Source.append(np.array([0,0,3]))
ColSrc = []
ColSrc.append(np.array([1,1,1]))

N = 512
Delta = 10

im_simple = lancer_complet(np.array([0,0,4]), np.array([0,0,1]), 1)

sauvegarde(im_simple, "2Billes_1source_avec_1_reflexion.jpg")
```

Et le résultat est le suivant :



Le reflet de la bille rouge sur la bille verte donne une zone jaune (synthèse additive des couleurs)

Un dernier exemple pour sourire (à vous de le lancer pour voir !)

```
[29]: Objet = []
Objet.append((np.array([0,-1,-3]), 2))
Objet.append((np.array([-1.5,2,-3]), 1.3))
Objet.append((np.array([1.5,2,-3]), 1.3))
Objet.append((np.array([0,-1.1,-0.8]), 0.2))
KdObj = []
KdObj.append(np.array([1,1,1]))
KdObj.append(np.array([.2,.2,.2]))
KdObj.append(np.array([.2,.2,.2]))
KdObj.append(np.array([0,0,0]))
KrObj = [1, 0, 0, 0]
Source = []
Source.append(np.array([2,2,3]))
ColSrc = []
ColSrc.append(np.array([1,1,1]))

N = 512
Delta = 6

#im_simple2 = lancer_complet(np.array([0,0,4]), np.array([0,0,1]), 1)

#sauvegarde(im_simple2, "souris.jpg")
```

### Question 28

La complexité obtenue questions 23-24 est multipliée par  $r_{max}$ , ou plus précisément par  $(1 + r_{max})$ .

## V.B - Une optimisation

### Question 29

```
[30]: def table_risque(risque):
    no = len(Objet)
    ns = len(Source)
    res = []
    for i in range(no):
        L = []
        for j in range(ns):
            L2 = []
            for i2 in range(no):
                if [IdObj[i], IdSrc[j], IdSrc[i2]] in risque:
                    L2.append(i2)
            L.append(L2)
        res.append(L)
    return res
```

### Question 30

```
[31]: def visible_opt(j, k, P):
    if au_dessus(Objet[k], P, Source[j]) == False:
        return False # Ainsi, on ne va pas plus loin
                       # si la sphère n'est pas visible.
    # On parcourt les autres sphères de la table à risque
    # et si l'intersection de la demi-droite P, source est non vide,
    # on renvoie False
    for i in range(TableRisque[k][j]):
        r = ra(P, Source[j])
        if intersection(r, obj[i]) != None:
            return False
    # Si on arrive ici, c'est que tout est bon !
    return True
```

**Fin !**