teaching sciences for innovation

Proposition de corrigé

Concours : Concours Centrale-Supélec

Année : 2021

Filière : MP - PC - PSI - TSI

Épreuve : Informatique

Ceci est une proposition de corrigé des concours de CPGE, réalisée bénévolement par des enseignants de Sciences Industrielles de l'Ingénieur et d'Informatique, membres de l'UPSTI (Union des Professeurs de Sciences et Techniques Industrielles), et publiée sur le site de l'association :

https://www.upsti.fr/espace-etudiants/annales-de-concours

A l'attention des étudiants

Ce document vous apportera des éléments de corrections pour le sujet traité, mais n'est ni un corrigé officiel du concours, ni un corrigé détaillé ou exhaustif de l'épreuve en question.

L'UPSTI ne répondra pas directement aux questions que peuvent soulever ces corrigés : nous vous invitons à vous rapprocher de vos enseignants si vous souhaitez des compléments d'information, et à vous adresser à eux pour nous faire remonter vos éventuelles remarques.

Licence et Copyright

En cas de doute, n'hésitez pas à nous contacter à : corrigesconcours@upsti.fr.

Informez-vous !

Retrouvez plus d'information sur les Sciences de l'Ingénieur, l'orientation, les Grandes Ecoles ainsi que sur les Olympiades de Sciences de l'Ingénieur et sur les Sciences de l'Ingénieur au Féminin sur notre site : www.upsti.fr

L'équipe UPSTI

Lancer de rayons

Corrigé UPSTI

I Géométrie

Q 1. Écrire une fonction d'entête

```
def vec(A:point, B:point) -> vecteur:
```

Solution basée sur les propriétés arithmétiques des numpy array :

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

Q 2. Écrire une fonction d'entête

```
def ps(v1:vecteur, v2:vecteur) -> float:
```

Solution utilisant la fonction numpy `np.inner(a,b)` donnée en fin de sujet :

```
1 def ps(v1, v2):
2     return np.inner(v1,v2)
```

Q 3. Écrire une fonction d'entête

```
def norme(v:vecteur) -> float:
```

Solution utilisant le carré scalaire pour définir la norme (avec la fonction `sqrt()` du module `numpy`, mais peut aussi bien se faire avec celle du module `math` puisqu'appliquée à un scalaire) :

```
1 def norme(v):
2     return np.sqrt(ps(v,v))
```

Q 4. Écrire une fonction d'entête

```
def unitaire(v:vecteur) -> vecteur:
```

Solution :

```
1 def unitaire(v):
2     return v/norme(v)
```

Q 5. Que font les fonctions `pt`, `dir` et `ra` ci-dessous ?

```
1 def pt(r:rayon, t:float) -> point:
2     assert t >= 0
3     (S, u) = r
4     return S + t * u
```

La fonction `pt` retourne le point du rayon `(S,u)` de mesure algébrique `t` devant être positive ou nulle.

```
1 def dir(A:point, B:point) -> vecteur:
2     return unitaire(vec(A, B))
```

La fonction `dir` retourne le vecteur unitaire support du vecteur $\overrightarrow{AB}$, et donc le vecteur directeur de la droite (AB).

```
1 def ra(A:point, B:point) -> rayon:
2     return (A, dir(A, B))
```

La fonction `ra` retourne le rayon d'origine `A` et de vecteur unitaire le vecteur directeur de (AB) : elle retourne donc le rayon d'origine `A` et passant par `B`.

Q 6. Écrire une fonction d'entête

```
def sp(A:point, B:point) -> sphère:
```

qui renvoie la sphère de centre A passant par B.

Solution renvoyant la sphère de centre A et de rayon $r = \| \overrightarrow{AB} \|$:

```
1 def sp(A, B):
2     return (A, norme(vec(A,B)))
```

Q 7. Montrer qu'une droite passant par le point A de vecteur directeur $\vec{u}$ et une sphère (C, r) sont sécantes si et seulement si l'équation d'inconnue t

$$t^2 + 2t(\vec{u} \cdot \overrightarrow{CA}) + \| \overrightarrow{CA} \|^2 - r^2 = 0 \tag{1}$$

possède deux solutions réelles, éventuellement confondues.

Le rayon d'origine A et de vecteur directeur $\vec{u}$ intersecte la sphère de centre C et de rayon r si il existe un point M (au moins un, rayon tangent à la sphère ; au plus deux) appartenant au rayon, de mesure t réelle et positive avec $\overrightarrow{AM} = t \cdot \vec{u}$, tel que $\| \overrightarrow{CM} \| = r$ (voir figure 1).

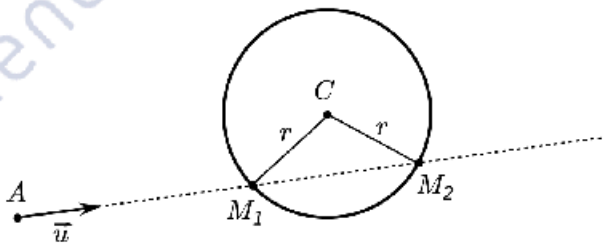

FIGURE 1 – Configuration d'une intersection entre un rayon et une sphère

Cela revient donc à chercher les solutions à valeurs réelles de l'équation suivante :

$$\begin{aligned}
\| \overrightarrow{CM} \|^2 = r^2 &\Leftrightarrow \overrightarrow{CM} \cdot \overrightarrow{CM} = r^2 \\
&\Leftrightarrow (\overrightarrow{CA} + \overrightarrow{AM}) \cdot (\overrightarrow{CA} + \overrightarrow{AM}) = r^2 \\
&\Leftrightarrow \overrightarrow{CA} \cdot \overrightarrow{CA} + 2\overrightarrow{AM} \cdot \overrightarrow{CA} + \overrightarrow{AM} \cdot \overrightarrow{AM} - r^2 = 0 \\
&\Leftrightarrow \| \overrightarrow{CA} \|^2 + 2t \cdot \vec{u} \cdot \overrightarrow{CA} + \| \overrightarrow{AM} \|^2 - r^2 = 0 \\
&\Leftrightarrow \| \overrightarrow{CA} \|^2 + 2t \cdot \vec{u} \cdot \overrightarrow{CA} + t^2 - r^2 = 0 \qquad \textbf{CQFD}
\end{aligned}$$

Q 8. Écrire une fonction d'entête

```
def intersection(r:rayon, s:sphère) -> (point, float) or None:
```

qui renvoie le premier point de la sphère `s` frappé par le rayon lumineux `r` et la distance entre ce point et l'origine du rayon. La fonction renvoie `None` si le rayon ne coupe pas la sphère. On suppose que l'origine du rayon n'est pas située à l'intérieur de la sphère.

Cela revient à résoudre l'équation $a \cdot t^2 + b \cdot t + c = 0$, avec $a = 1$, $b = 2\vec{u} \cdot \overrightarrow{CA}$ et $c = \| \overrightarrow{CA} \|^2 - r^2$, et à ne garder que la solution minimale des deux, en considérant que le point en question est de mesure algébrique $t > 0$ sur le rayon.

```python
def intersection(r, s):
    (C, rc) = s # sphere de centre C et de rayon rc
    (A, u) = r  # rayon d'origine A et de vecteur directeur u
    a = 1
    b = 2*ps(u, vec(C,A))
    c = ps(vec(C,A),vec(C,A))-rc**2
    delta = b**2-4*a*c   # Calcul du discriminant
    if delta < 0:
        return None
    else:
        # premier point est celui de distance minimale
        # soit donc pour la solution avec - racine de delta
        t = (-b-np.sqrt(delta))/(2*a)
        # Résultat valide seulement pour t > 0
        if t > 0:
            return (pt(r,t), t)
        else:
            return None
```

II Optique

II-A - Visibilité

Q 9. Une source lumineuse ponctuelle S ne peut être vue d'un point P d'une sphère (C, r) que si la source est au-dessus de l'horizon de P, défini ici comme le plan tangent à la sphère en P. Donner une condition géométrique pour que la source soit au-dessus de l'horizon.

Il suffit de remarquer que le plan d'horizon du point P est de vecteur normal $\vec{N}$, vecteur unitaire obtenu à partir du vecteur $\overrightarrow{CP}$, et que selon la position du point S, l'angle θ entre le vecteur $\vec{N}$ et le vecteur $\overrightarrow{PS}$ sera tel que :

- $0 \leq \theta < \frac{\pi}{2}$, si S est situé au-dessus de l'horizon ;
- $\theta = \frac{\pi}{2}$, si S appartient à l'horizon ;
- $\frac{\pi}{2} < \theta \leq \pi$, si S est situé au-dessous de l'horizon.

On peut alors exploiter avantageusement les propriétés du produit scalaire, sachant que le signe du résultat sera le même que l'on utilise le vecteur normal $\vec{N}$ ou le vecteur $\overrightarrow{CP}$, pour en déduire que

S sera situé au-dessus de l'horizon de P si $\overrightarrow{CP} \cdot \overrightarrow{PS} > 0$.

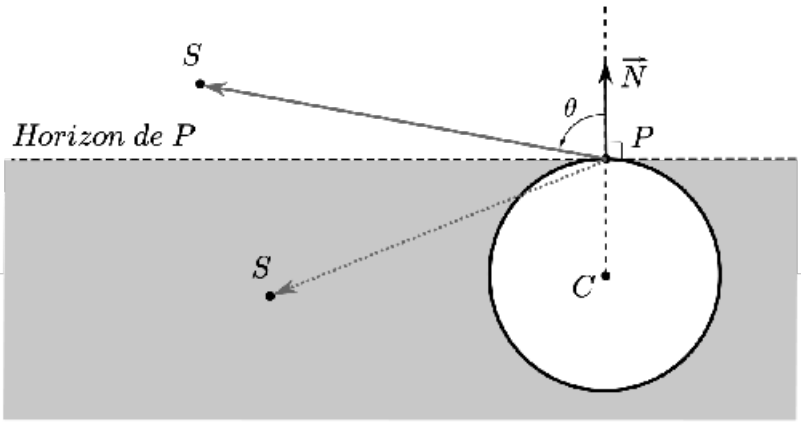

FIGURE 2 – Positions de la source lumineuse S selon l'horizon du point P.

Q 10. Écrire une fonction d'entête

```
def au_dessus(s:sphère, P:point, src:point) -> bool:
```

qui détermine si la source située en `src` est au-dessus de l'horizon du point `P` de la sphère `s`.

Il suffit d'exploiter le résultat précédent :

```
1 def au_dessus(s, P, src):
2     (C, rc) = s
3     return ps(vec(C,P),vec(P,src)) > 0
```

Q 11. On considère une scène contenant plusieurs sphères et une source lumineuse. Pour que la source soit visible d'un point P d'une sphère particulière, il faut que cette source soit au-dessus de l'horizon et qu'aucune autre sphère ne la cache. Écrire une fonction booléenne d'entête

```
def visible(obj:[sphère], j:int, P:point, src:point) -> bool:
```

où le paramètre `obj` est une liste contenant les sphères de la scène et `src` l'emplacement de la source lumineuse. Cette fonction détermine si la source est visible du point `P`, appartenant à la sphère `obj[j]`.

Avant tout il faut tester si la source lumineuse est visible. Si oui, il suffit alors de tester si un point d'intersection existe entre le rayon d'origine `src` passant par `P` et chaque sphère de `obj` (différente de la sphère `j` !), et si celui-ci est de distance inférieure à la distance entre `src` et `P` (voir figure 3).

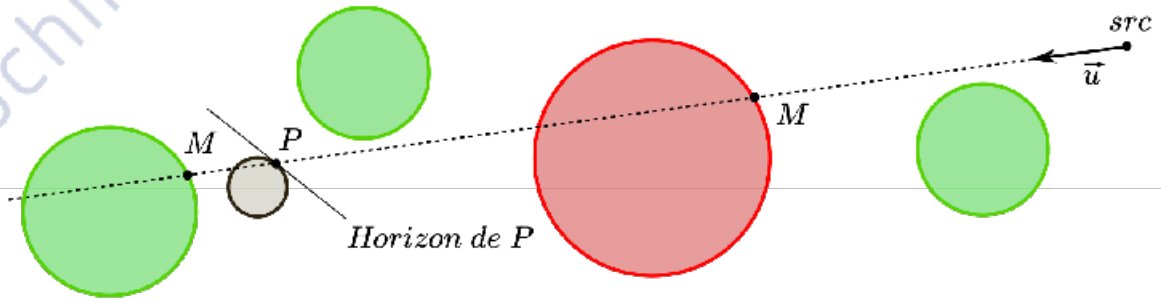

FIGURE 3 – Différentes positions de sphères, occultant (en rouge) ou non (en vert) le rayon.

```
def visible(obj, j, P, src):
    if au_dessus(obj[j], P, src): # Source lumineuse potentiellement visible
        r = ra(src, P)      # création du rayon d'origine src passant par P
        d = norme(vec(src,P))    # Distance entre le point P et la source
        for i in range(len(obj)):       # Boucle sur chaque sphere de obj
            if i != j: # On teste toutes les sphères sauf celle d'indice j
                M = intersection(r, obj[i])
                # Si intersection existe et occultante, alors non visible
                if M != None and M[1] < d:
                    return False
        # Arrivé ici, la source lumineuse est visible
        return True
    else:
        return False
```

II-B - Diffusion

Q 12. Écrire une fonction d'entête

```
def couleur_diffusée(r:rayon, Cs:couleur, N:vecteur, kd:couleur) -> couleur:
```

qui renvoie la couleur de la lumière diffusée par le point `P` éclairé par un rayon lumineux `r` en provenance d'une source ponctuelle de couleur `Cs`. Les paramètres `N` et `kd` représentent respectivement le vecteur unitaire normal à l'objet en `P` et les coefficients de diffusion de l'objet (figure 2). La source `S` est supposée visible de `P`.

Remarque : il est à déplorer un caractère accentué dans le prototype de la fonction... À éviter absolument !

Le produit de Hadamard est simplement obtenu avec les numpy array par un simple produit entre deux vecteurs, générant un vecteur égal au produit termes à termes des deux vecteurs. Le cosinus de l'angle θ est obtenu par le produit scalaire des vecteurs unitaires $\vec{N}$ et $-\vec{u}$ en remarquant que $\vec{N}\cdot(-\vec{u}) = \cos\theta$.

```
def couleur_diffusée(r, Cs, N, kd):
    (S, u) = r
    cos_theta = ps(N, -u)
    return kd*Cs*cos_theta
```

II-C - Réflexion

Q 13. Écrire une fonction d'entête

```
def rayon_réfléchi(s:sphère, P:point, src:point ) -> rayon:
```

qui renvoie le rayon réfléchi par le point `P` de la sphère `s` en provenance de la source S placée en `src`. Le résultat est le couple $(P, \vec{w})$ représentant le rayon émergent. La source S est supposée visible de P.

Le rayon réfléchi est donc celui illustré figure 4, où $\theta' = \theta$.

On sait de plus que $-\vec{u}\cdot(\vec{N}) = \vec{w}\cdot(\vec{N}) = \cos\theta$, dont on déduit que

$$\begin{aligned}(\vec{w}-\vec{u})\cdot\vec{N} &= 2\cos\theta\\ \vec{w}-\vec{u} &= 2\cos\theta\cdot\vec{N}\\ \vec{w} &= \vec{u}+2\cos\theta\cdot\vec{N}\\ \vec{w} &= \vec{u}-2(\vec{u}\cdot\vec{N})\cdot\vec{N}\end{aligned}$$

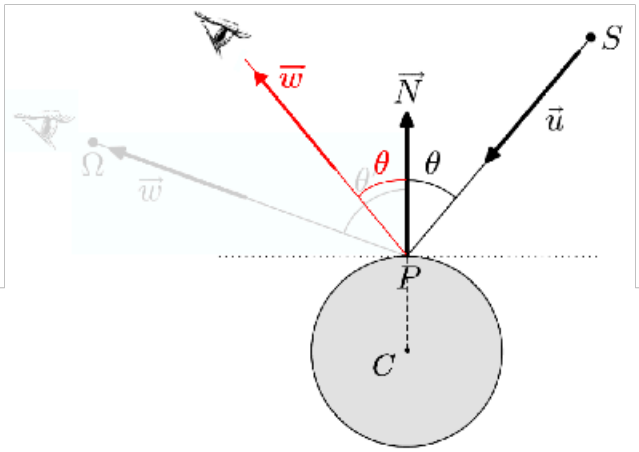

FIGURE 4 – Rayon réfléchi en P issu de la source lumineuse S en respect des lois de Descartes.

```
def rayon_reflechi(s, P, src):
    (C, rc) = s          # Récupération du centre de la sphère
    N = dir(C,P)         # Calcul du vecteur unitaire N
    u = dir(src,P)       # Calcul du vecteur unitaire u
    w = u-2*ps(u,N)*N    # Calcul du vecteur unitaire w
    return (P,w)         # Retour du rayon (P, w)
```

III Enregistrement des scènes

Q 14. Écrire une requête SQL qui donne le nom des scènes créées au cours de l'année 2021.

Il faut utiliser pour cela la fonction EXTRACT(*part* FROM t) donnée en fin d'énoncé, pour récupérer la valeur `year` d'un type `timestamp`.

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

Q 15. Écrire une requête SQL qui donne, pour chaque scène, son identifiant et le nombre de sources qui l'éclairent.

```
SELECT sc_id, COUNT(*) as nombre_sources
FROM Source
GROUP BY sc_id ;
```

Remarque : si aucune source n'éclaire une scène, celle-ci n'apparaitra pas dans le résultat de la requête, même avec une jointure sur la table Scene.

Solution : réaliser une union (avec le mot-clé ALL pour garder tous les doublons) entre tous les sc_id présents dans la table Scene (clé primaire donc un sc_id unique par scène) et ceux présents dans la table Source (autant de sc_id que de sources). Il suffit alors de regrouper la table résultat par sc_id, le nombre de sources sera égal au nombre de ligne -1 (et si aucune source n'éclaire une scène, il y aura au minimum la ligne issue de la table Scene) :

```
SELECT sc_id, COUNT(*)-1 as nombre_sources
FROM
(SELECT sc_id FROM Scene
UNION ALL
SELECT sc_id FROM Source)
GROUP BY sc_id;
```

Q 16. Écrire une requête SQL qui liste l'identifiant, les coordonnées du centre et le rayon de toutes les sphères contenues dans la scène dont le nom est woodbox. On suppose qu'une seule scène possède ce nom.

Il suffit d'effectuer une requête sur la jointure entre les tables Scene, Objet et Sphere, pour un même sc_id entre les deux premières et un ob_id = sp_id pour les deux dernières, en utilisant un renommage pour éviter toute ambigüité.

```
SELECT sp.sp_id as sp_id, ob.ob_x as xc, ob.ob_y as yc, ob.ob_z as zc, sp.sp_rayon as rayon
FROM Scene as sc JOIN Objet as ob JOIN Sphere as sp
ON sc.sc_id = ob.sc_id AND ob.ob_id = sp.sp_id
WHERE sc.sc_nom = "woodbox";
```

Q 17. Écrire une requête qui, pour la scène woodbox, renvoie tous les triplets objr_id, so_id, objo_id pour lesquels l'objet d'identifiant objo_id occulte la source d'identifiant so_id pour l'objet d'identifiant objr_id.

Pour répondre à cette question il faut générer toutes les combinaisons d'objets pris deux à deux pour chaque source présente dans la scène "woodbox" : consiste à réaliser un produit cartésien de la table Objet par elle-même en excluant les cas de même ob_id (un objet ne peut pas s'occulter lui-même), pour chaque objet et chaque source lumineuse présente dans la scene "woodbox" (jointure avec les tables Scene et Source sur mêmes sc_id). La sélection s'effectue sur le nom de la scene et sur le booléen renvoyé par la fonction `OCCULTE()`.

```
SELECT o1.ob_id as ojbr_id, so.so_id as so_id , o2.ob_id as objo_id
FROM Scene as sc JOIN Objet as o1 JOIN Objet as o2 JOIN Source as so
ON so.sc_id = sc.sc_id AND o1.sc_id = sc.sc_id AND o2.sc_id = sc.sc_id AND o1.ob_id != o2.ob_id
WHERE sc.sc_nom = "woodbox" AND OCCULTE(sc.sc_id, o1.ob_id, so.so_id, o2.ob_id);
```

IV Lancer de rayons

IV-A - Écran

Q 18. Écrire une fonction d'entête

```
def grille(i:int, j:int) -> point:
```

qui renvoie les coordonnées cartésiennes du point E, centre de la case repérée par les indices (i, j) de la grille (figure 4).

On remarquera que le côté d'une case est de longueur $\frac{\delta}{N}$, que l'écran est à la profondeur $z = 0$, et que les cooordonnées des points d'indices $(0, 0)$ et $(N-1, N-1)$ sont celles exprimées sur la figure 5, ce qui nous donne comme relations :

$$\begin{cases} x &= \frac{\delta}{N}\left[j - \frac{N-1}{2}\right] \\ y &= \frac{\delta}{N}\left[-i + \frac{N-1}{2}\right] \\ z &= 0 \end{cases}$$

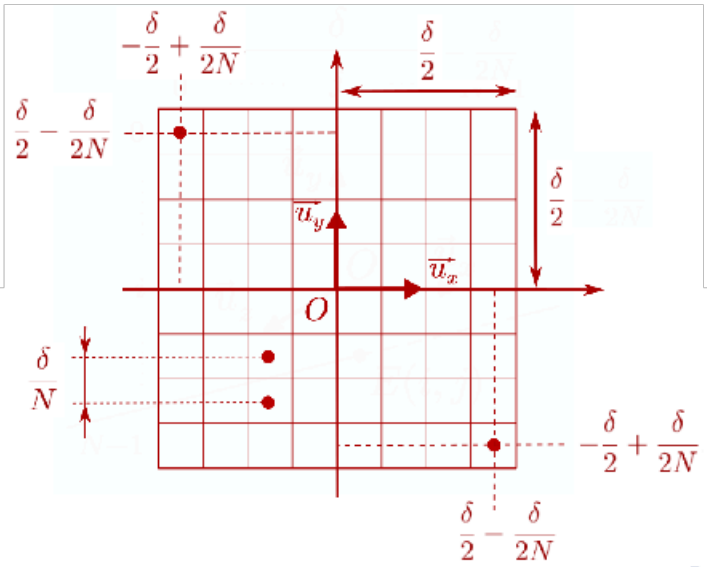

FIGURE 5 – Coordonnées dans le plan $(O, \overrightarrow{u_x}, \overrightarrow{u_y})$ des cases Top-Left et Bottom-Right.

```
def grille(i, j):
    x = Delta*(2*j-N+1)/(2*N)
    y = Delta*(-2*i+N-1)/(2*N)
    return np.array([x,y,0])
```

Q 19. Écrire une fonction d'entête

```
def rayon_écran(omega:point, i:int, j:int) -> rayon:
```

qui renvoie le rayon issu du point `omega` et passant par $E(i, j)$.

```
def rayon_ecran(omega, i, j):
    E = grille(i, j)
    return ra(omega, E)
```

IV-B - Couleur d'un pixel

Q 20. Écrire une fonction d'entête

```
def interception(r:rayon) -> (point, int) or None:
```

qui prend en paramètre un rayon `r` et qui renvoie le premier point matériel de la scène atteint par ce rayon ainsi que l'indice de la sphère concernée dans la liste Objet. Si le rayon n'intercepte aucune sphère, la fonction renvoie `None`.

Il suffit de s'inspirer de la fonction `visible`, en ne gardant que le point de distance minimale, si interception il y a. C'est donc une recherche de minimum, qui peut ne pas exister.

On peut alors décider d'utiliser une variable booléenne `found` permettant de gérer l'existence ou non du point, ainsi que l'initialisation du minimum.

```python
def interception(r):
    found = False
    M_min, i_min = 0, 0
    # Boucle sur chaque sphere de Objet, avec récupération de l'indice
    for i in range(len(Objet)):
        M = intersection(r, Objet[i])
        # Si intersection existe
        if M != None:
            # Initialisation minimum sur premier point trouvé
            if not found:
                M_min = M
                i_min = i
                found = True # On a au moins une interception
            elif M[1] < M_min[1]: # Point de distance plus courte trouvé
                M_min = M
                i_min = i
    # Test si interception et renvoi du résultat
    if found:
        return (M_min[0], i_min)
    # Sinon renvoi de None
    return None
```

Q 21. Écrire une fonction d'entête

```
def couleur_diffusion(P:point, j:int) -> couleur:
```

qui renvoie la couleur diffusée par le point `P` appartenant à la sphère `Objet[j]`. Si aucune source n'éclaire `P`, la fonction renvoie `noir`.

```python
def couleur_diffusion(P, j):
    # Initialisation de la couleur diffusée totale à la couleur noire
    Cd = noir
    # Récupération du centre et du rayon de l'objet sphere d'indice j
    (C, rc) = Objet[j]
    # Création du vecteur unitaire normal à la sphère en P
    N = dir(C, P)
    # Boucle sur chaque source avec récupération de l'indice
    for i in range(len(Source)):
        # Test si source lumineuse visible par P
        if visible(Objet, j, P, Source[i]):
            # Si oui, accumulation de la couleur diffusée par celle-ci
            r = ra(Source[i], P)    # Rayon d'origine Source[i] passant par P
            Cd = Cd + couleur_diffusée(r, colSrc[i], N, KdObj[j])
    # Renvoi couleur diffusée totale
    return Cd
```

IV-C - Constitution de l'image

Q 22. Écrire une fonction d'entête

```
def lancer(omega:point, fond:couleur) -> image:
```

qui génère l'image associée à la scène. Si un rayon n'intercepte aucun objet, le pixel correspondant est de couleur `fond`.

Il suffit d'exploiter toutes les fonctions précédemment écrites, en effectuant un balayage de l'image à générer.

```
def lancer(omega, fond):
    img = np.zeros((N,N,3))
    # Balayage de l'image pour chaque pixel
    for i in range(N):
        for j in range(N):
            # calcul du rayon à lancer de source omega et passant par E(i,j)
            r = rayon_ecran(omega, i, j)
            # Détermination du point d'interception
            I = interception(r)
            if I != None:
                # Si point d'interception calcul de la couleur diffusée
                (P, k) = I
                c = couleur_diffusion(P, k)
                img[i][j] = c
            else:
                #... Sinon attribution de la couleur de fond
                img[i][j] = fond
    # Renvoi de l'image générée
    return img
```

IV-D - Complexités

Q 23. Calculer la complexité temporelle de la fonction `lancer` dans le meilleur des cas en caractérisant la situation correspondante.

Il faut pour cela évaluer les complexités temporelles des différentes fonctions appelées :

Fonction	Meilleur des cas	Pire des cas
`rayon_ecran()`	Algorithme linéaire, à coût constant : complexité en $\mathcal{O}(1)$	
`intersection()`	Algorithme linéaire, à coût constant : complexité en $\mathcal{O}(1)$	
`visible()`	1 itération (source lumineuse pas visible par P) à coût constant : complexité en $\mathcal{O}(1)$	Boucle sur les n_o objets (source lumineuse visible par P), à coût constant : complexité en $\mathcal{O}(n_o)$
`interception()`	Boucle sur n_o objets à coût constant : complexité en $\mathcal{O}(n_o)$	
`couleur_diffusion()`	Aucune source lumineuse visible par P, soit boucle sur n_s sources à coût constant : complexité en $\mathcal{O}(n_s)$	Toutes les sources lumineuses visibles par P, soit n_s itérations de coût en $\mathcal{O}(n_o)$: complexité en $\mathcal{O}(n_o \cdot n_s)$
`lancer()`	Balayage sur les N^2 pixels, aucune interception d'objets (image de fond pur) à coût en $\mathcal{O}(n_o)$: complexité en $\mathcal{O}(N^2 \cdot n_o)$	Balayage sur les N^2 pixels, interception de chaque rayon par un objet de la scène à coût dans le pire des cas en $\mathcal{O}(n_o \cdot n_s)$ (pire des cas de `couleur_diffusion`) : complexité en $\mathcal{O}(N^2 \cdot n_o \cdot n_s)$

Donc ici le meilleur des cas serait celui où aucun objet n'est présent dans l'image générée (image de fond pur), donc une complexité temporelle en $\mathcal{O}(N^2 \cdot n_o)$.

Q 24. Calculer la complexité temporelle de la fonction `lancer` dans le pire des cas en caractérisant la situation correspondante.

Pire des cas = image remplie intégralement d'objets visibles en tous points par toutes les sources lumineuses (signifie que toutes les sources lumineuses soient situées en `omega`...), donc une complexité temporelle en $\mathcal{O}(N^2 \cdot n_o \cdot n_s)$.

V Améliorations

V-A - Prise en compte de la réflexion

Q 25. Écrire une fonction d'entête

```
def réflexions(r:rayon, rmax:int) -> [(point, int)]:
```

qui renvoie une liste de couples (P_k, i_k) correspondant aux points successivement rencontrés par le rayon `r` au fur et à mesure de ses réflexions. Dans chaque couple, P_k est le point où a lieu la $(k+1)^{ème}$ réflexion et i_k est l'indice de l'objet contenant P_k. Le résultat comporte au plus `rmax` éléments, les éventuelles réflexions ultérieures ne sont pas prises en compte.

Il suffit d'exploiter les fonctions `rayon_reflechi()` et `interception()` soit :

- dans une boucle conditionnée à l'existence d'intersections successives (en repartant du rayon réfléchi) et à au plus `rmax` couples résultats ;
- de manière récursive, de la même manière, avec comme cas de base la non existence d'interception et comme cas limite l'atteinte de `rmax` récursivités.

```
""" Version itérative"""
def reflexions(r, rmax):
    # Initialisations
    i = 0
    L = []
    rf = r

    # Calcul première interception
    I = interception(rf)

    # Boucle tant qu'interception ou rmax fois au maximum
    while I != None and i < rmax:
        # Ajout du résultat dans la liste L
        L.append(I)
        # récupération du point P et de l'indice k de l'objet
        (P, k) = I
        # Calcul du nouveau rayon à partir du point P
        rf = rayon_reflechi(Objet[k], P, rf[0])
        # Incrément compteur de boucle
        i = i + 1
        # Calcul nouvelle interception
        I = interception(rf)

    # Renvoi liste résultat
    return L
```

```python
""" Version récursive """
def reflexions(r, rmax):
    # Cas limite
    if rmax <= 0:
        return []
    # Cas de base
    I = interception(r)
    if I == None:
        return []

    # Une interception existe : récursivité
    # récupération du point P et de l'indice k de l'objet
    (P, k) = I
    # Calcul du nouveau rayon à partir du point P
    rf = rayon_reflechi(Objet[k], P, r[0])
    # Récupération de la liste résultat récursive
    L = reflexions(rf, rmax-1)
    # Insertion du résultat en début de liste (k-ème réflexion)
    L.insert(0, I)

    # Renvoi liste résultat
    return L
```

Q 26. Écrire une fonction d'entête

```
def couleur_perçue(r:rayon, rmax:int, fond:couleur) -> couleur:
```

qui renvoie la couleur du premier point de la scène rencontré par le rayon `r` en tenant compte d'au maximum `rmax` réflexions de ce rayon. Si le rayon ne rencontre aucun objet, la fonction renvoie la couleur `fond`.

Le plus simple est de construire la couleur percue en respect de la relation donnée, en bouclant sur la liste des reflexions par ordre décroissant des indices de `rmax-1` à `0` inclus.

```python
def couleur_percue(r, rmax, fond):
    # Calcul des reflexions successives
    L = reflexions(r, rmax)

    # Couleur de fond si aucune reflexion
    if len(L) == 0:
        return fond

    Ck = noir
    # Boucle de rmax-1 à 0 inclus par pas de -1
    for i in range(len(L)-1, -1, -1):
        # recupération point P et indice de sphere k
        (P, k) = L[i]
        # Calcul récurrent de couleur percue
        Cdk = couleur_diffusion(P, k)
        kr = KrObj[k]
        Ck = Cdk + kr*Ck
    # Renvoi de la couleur percue
    return Ck
```

Q 27. Écrire une fonction d'entête

```
def lancer_complet(omega:point, fond:couleur, rmax:int) -> image:
```

qui construit l'image de la scène en tenant compte des diffusions et des réflexions.

Il suffit de modifier la fonction `lancer` pour maintenant affecter la couleur percue à chaque pixel, sachant que l'interception du rayon ou non (et ses réflexions successives) est maintenant inclus dans l'appel de la fonction `couleur_percue`.

```
def lancer_complet(omega, fond, rmax):
    img = np.zeros((N,N,3))
    # Balayage de l'image pour chaque pixel
    for i in range(N):
        for j in range(N):
            # calcul du rayon à lancer de source omega et passant par E(i,j)
            r = rayon_ecran(omega, i, j)
            # affectation de la couleur percue
            img[i][j] = couleur_percue(r, rmax, fond)
    # Renvoi de l'image générée
    return img
```

Q 28. Exprimer la nouvelle complexité dans le pire des cas.

Dans le pire des cas on obtient :

- fonction `reflexions` : r_{max} itérations à coût constant $\Rightarrow$ complexité en $\mathcal{O}(r_{max})$;
- appel de `couleur_percue` : appel de `reflexions` + r_{max} appels de la fonction `couleur_diffusion` de complexité en $\mathcal{O}(n_o \cdot n_s) \Rightarrow$ complexité en $\mathcal{O}(r_{max} \cdot n_o \cdot n_s)$;
- appel de `lancer_complet` : N^2 appels de `couleur_percue` $\Rightarrow$ complexité en $\mathcal{O}(N^2 \cdot r_{max} \cdot n_o \cdot n_s)$.

Donc la complexité temporelle est multipliée du nombre maximal de réflexion successives r_{max} par rapport à la fonction `lancer` (ce à quoi on pouvait logiquement s'attendre...).

V-B - Une optimisation

Q 29. Écrire une fonction d'entête

```
def table_risque(risque:[[int, int, int]]) -> [[[int]]]:
```

qui prend en paramètre une liste de triplets correspondant au résultat de la requête SQL de la question 17 et construit une liste de listes de listes d'entiers telle que, si `res` est le résultat de la fonction, `res[i][j]` donne la liste (éventuellement vide) des indices des objets susceptibles de masquer la source `Source[j]` pour un point de l'objet `Objet[i]`.

On peut ici avantageusement utiliser la propriété `L.index(e)` d'une liste `L`, puisqu'il va falloir retrouver l'index de chaque élément de chaque triplet à partir de l'identifiant stocké dans les listes `IdObj` et `IdSrc`.

```
def table_risque(risque):
    no = len(Objet)
    ns = len(Source)
    # Creation du tableau de listes vides de dimensions no x ns
    res = [[[]]*ns]*no

    # Boucle pour chaque triplet du résultat de la requête SQL
    for objr_id, so_id, objo_id in risque:
        # récupération des index des objets et de la source
```

```
        i_objr = IdObj.index(objr_id)
        j_src =  IdSrc.index(so_id)
        i_objo = IdObj.index(objo_id)
        # Ajout de l'objet occultant dans le tableau
        res[i_objr][j_src].append(i_objo)

    # renvoi du tableau résultat
    return res
```

Q 30. Le résultat de la fonction `table_risque` est conservé dans la variable globale `TableRisque`. Écrire une fonction d'entête

```
def visible_opt(j:int, k:int, P:point) -> bool:
```

qui prend en paramètres l'indice `j` d'une source, l'indice `k` d'une sphère et un point `P` de cette sphère et qui, comme la fonction visible de la question 11, détermine si la source `Source[j]` est visible à partir du point `P`.

On peut de manière simple utiliser la fonction `visible` dont on construit la liste des sphères susceptibles de masquer la source j pour un point de la sphere k à partir des données contenues dans `TableRisque`.

```
def visible_opt(j, k, P):
    # Creation de la liste des sphères susceptibles de masquer Source[j]
    # Pour un point de l'objet Objet[k]
    obj = [Objet[i] for i in TableRisque[k][j]]
    # Appel de la fonction visible pour la liste des sphères créée
    return visible(obj, k, P, Source[j])
```

Néanmoins, la construction de la liste d'objets potentiellement occultants génère un coût supplémentaire non négligeable, et la solution optimale est donc de réécrire entièrement la fonction sur la base de la fonction `visible`, où la sphère `j` est maintenant assurée de ne pas être une sphère potentiellement occultante (puisque le cas de non occultation d'une sphère par elle-même est déjà géré dans l'appel de la requête SQL de la question 17) :

```
def visible_opt(j, k, P):
    if au_dessus(Objet[k], P, Source[j]): # Source j potentiellement visible
        r = ra(Source[j], P)  # création du rayon (Source[j], P)
        d = norme(vec(Source[j],P)) # Distance entre le point P et la source
        for i in TableRisque[k][j]: # Pour chaque sphere susceptible de masquer P
            M = intersection(r, Objet[i])
            # Si intersection existe et occultante, alors non visible
            if M != None and M[1] < d:
                return False
        # Arrivé ici, la source lumineuse est visible
        return True
    else:
        return False
```

Résultats obtenus avec les solutions proposées dans ce corrigé.

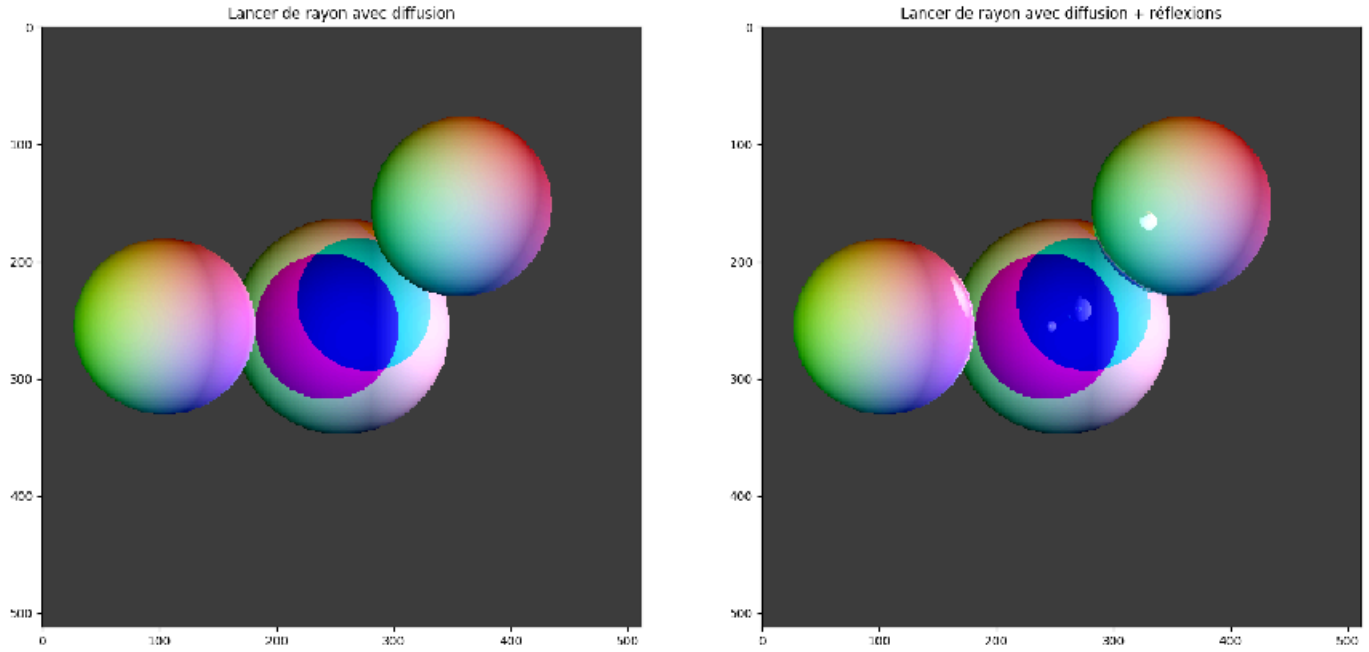

FIGURE 6 – Résultats pour 3 sphères, 4 sources lumineuses avec $N = 512$.

```
# Quelques couleurs
noir = np.array([0., 0., 0.])
gris = np.array([.5, .5, .5])
blanc = np.array([1., 1., 1.])
rouge = np.array([1., 0., 0.])
bleu = np.array([0., 0., 1.])
vert = np.array([0., 1., 0.])

# Créations objets
Objet = [(np.array([0., 0., -10.]), 1.5), \
         (np.array([-1., 0., -4.]), 0.5), \
         (np.array([1., 1., -6.]), 0.75)]
KdObj = [.9, .9, .9]
KrObj = [0.5, 0.2, 0.2]

# Création sources lumineuses
Source = [np.array([6., 0., -10.]),\
          np.array([-2., 0., 2.]),\
          np.array([1., -1., -1.]),\
          np.array([6., 6., 14.])]
colSrc = [blanc, vert, bleu, rouge]

# Configuration de l'image résultat
Delta = .1
omega = np.array([0., 0., .12])
N = 512
fond = noir

# Génération des images et tracé
img = lancer(omega, fond)
img2 = lancer_complet(omega, fond,10)
```

```
32 plt.subplot(1,2,1)
33 plt.imshow(img)
34 plt.title("Lancer de rayon avec diffusion")
35 plt.subplot(1,2,2)
36 plt.imshow(img2)
37 plt.title("Lancer de rayon avec diffusion + réflexions")
38 plt.show()
```