Programmes Python & Annexes

Programme d'affichage du tapis

Bibliothèque :

```
import numpy as np

import matplotlib.pyplot as plt
```

Constantes :

```
# caractéristiques du billard :

D = 0.7115 # demi-largeur du billard

D2 = 0.1815 # distance caractérisant le positionnement des boules

Long_bi = 4 * D

epaisseur_bi = 0.127 # épaisseur du contour en bois

epaisseur_bande = (1 / 5) * epaisseur_bi

Larg_bi = 2 * D

# caractéristique de la boule :

Rb = 0.0615 # rayon d'une boule
```

Affichage :

```
P0_blanche = np.array([D + D2, D]) # position initiale de la boule blanche

P0_rouge = np.array([D, 3 * D]) # position initiale de la boule rouge

P0_jaune = np.array([D, D]) # position initiale de la boule jaune

# affichage du contour de la table sans épaisseur :

plt.plot([0, 0],[0, Long_bi], color = 'g')

plt.plot([0, Larg_bi],[0, 0], color = 'g')

plt.plot([Larg_bi, Larg_bi],[0, Long_bi], color = 'g')

plt.plot([0, Larg_bi],[Long_bi, Long_bi], color = 'g')

# affichage du contour de la table avec épaisseur :

plt.plot([- epaisseur_bi, - epaisseur_bi],[- epaisseur_bi, Long_bi + epaisseur_bi], color = 'brown')

plt.plot([- epaisseur_bi, Larg_bi + epaisseur_bi],[- epaisseur_bi, - epaisseur_bi], color = 'brown')

plt.plot([Larg_bi + epaisseur_bi, Larg_bi + epaisseur_bi],[- epaisseur_bi, Long_bi + epaisseur_bi], color = 'brown')

plt.plot([- epaisseur_bi, Larg_bi + epaisseur_bi],[Long_bi + epaisseur_bi, Long_bi + epaisseur_bi], color = 'brown')

# remplissages bois :

plt.fill_between([- epaisseur_bi, - epaisseur_bande], - epaisseur_bi, Long_bi + epaisseur_bi, color = 'brown')

plt.fill_between([- epaisseur_bi, Larg_bi + epaisseur_bi], Long_bi + epaisseur_bande, Long_bi + epaisseur_bi, color = 'brown')

plt.fill_between([- epaisseur_bi, Larg_bi + epaisseur_bi], - epaisseur_bi, - epaisseur_bande, color = 'brown')
```

```
plt.fill_between([Larg_bi + epaisseur_bande, Larg_bi + epaisseur_bi], - epaisseur_bi, Long_bi + epaisseur_bi, color = 'brown')


# remplissage tapis :
plt.fill_between([0, Larg_bi], 0, Long_bi, color = (0.4, 0.4, 0.99))
# remplissage bandes :
plt.fill_between([- epaisseur_bande, 0], - epaisseur_bande, Long_bi + epaisseur_bande, color = 'g')
plt.fill_between([- epaisseur_bande, Larg_bi + epaisseur_bande], Long_bi, Long_bi + epaisseur_bande, color = 'g')
plt.fill_between([- epaisseur_bande, Larg_bi + epaisseur_bande], - epaisseur_bande, 0, color = 'g')
plt.fill_between([Larg_bi, Larg_bi + epaisseur_bande], - epaisseur_bande, Long_bi + epaisseur_bande, color = 'g')
# points de départ :
plt.plot([P0_blanche[0]],[P0_blanche[1]], marker="o", color = 'white')
plt.plot([P0_rouge[0]],[P0_rouge[1]], marker="o", color = 'r')
plt.plot([P0_jaune[0]],[P0_jaune[1]], marker="o", color = 'yellow')
# tracé des cercles autour des boules
t = np.linspace(0, 2* np.pi, 1000) # permet de tracer des cercles autour des positions des boules
plt.plot(P0_blanche[0] + Rb*np.cos(t), P0_blanche[1] + Rb*np.sin(t), color = 'white')
plt.plot(P0_rouge[0] + Rb*np.cos(t), P0_rouge[1] + Rb*np.sin(t), color = 'r')
plt.plot(P0_jaune[0] + Rb*np.cos(t), P0_jaune[1] + Rb*np.sin(t), color = 'yellow')
# détails :
plt.xlabel('Position en x')
plt.ylabel('Position en y')
plt.title("Positions initiales")
plt.axis('equal') # permet de se placer dans un repère orthonormé
plt.show()
```

Programme final de simulation et d'optimisation

Bibliothèques :

```
import numpy as np # permet de travailler avec des vecteurs
import random
import sqlite3 # offre une passerelle avec les bases de données
import cmath # permet de travailler avec des nombres complexes
import time # permet de mesurer le temps d'exécution du programme

start_time = time.time() # début de la prise de mesure du temps
```

Constantes :

```
# caractéristiques du billard :
D = 0.7115 # demi-largeur du billard
D2 = 0.1815 # distance caractérisant le positionnement des boules
Long_bi = 4 * D
epaisseur_bi = 0.127
epaisseur_bande = (1/5)*epaisseur_bi
Larg_bi = 2 * D
e = 0.7 # coefficient de restitution des bandes
# caractéristiques de la boule :
Rb = 0.0615 # rayon de la boule
m = 0.209 # masse de la boule
# coefficients généraux :
mu_r = 0.018 # coefficient de frottement de roulement
g = 9.81 # accélération de pesanteur
```

Fonctions auxiliaires :

```
def vitesse(V0, t):
        return - mu_r * g * t * (V0 / (np.linalg.norm(V0))) + V0
def avance(V0, t, P0):
        return P0 + V0 * t - (1/2) * mu_r * g * t**2 * (V0/(np.linalg.norm(V0)))
```

```
## fonction : 'collision'

def collision(P1, P2, V1):
        V1_c = cmath.polar(V1[0] + 1j*V1[1])
        v1 = V1_c[0]
        theta = V1_c[1]
        dist = np.array([P2[0] - P1[0], P2[1] - P1[1]])
        u = dist / np.linalg.norm(dist)
        U_c = cmath.polar(u[0] + 1j * u[1])
        theta_u = U_c[1]
        V1prime_c = 0
        V2prime_c = 0
        if P1[0] < P2[0] and P1[1] < P2[1] and V1[1] > 0:
                theta1 = np.pi / 2 - (theta - theta_u)
                theta2 = -(theta - theta_u)
                V2prime_c = v1 * np.cos(theta2) * cmath.exp(1j * U_c[1])
                V1prime_c = v1 * np.cos(theta1) * cmath.exp(1j * (theta + theta1))
        elif P1[0] < P2[0] and P1[1] < P2[1] and V1[1] < 0:
                theta1 = - ( np.pi / 2 -theta_u -(2 * np.pi - theta))
                theta2 = np.pi / 2 + theta1
                V2prime_c = v1 * np.cos(theta2) * cmath.exp(1j*U_c[1]) # vitesse complexe de la boule à l'arrêt après le choc
                V1prime_c = v1 * np.cos(theta1) * cmath.exp(1j*(theta + theta1))
        elif P1[0] < P2[0] and P1[1] > P2[1] and V1[1] > 0:
                theta1 = np.pi / 2 - theta -( 2 * np.pi - theta_u)
                theta2 = - np.pi / 2 + theta1
                V2prime_c = v1 * np.cos(theta2) * cmath.exp(1j*U_c[1]) # vitesse complexe de la boule à l'arrêt après le choc
                V1prime_c = v1 * np.cos(theta1) * cmath.exp(1j*(theta + theta1))
        elif P1[0] < P2[0] and P1[1] > P2[1] and V1[1] < 0:
                theta1 = - ( np.pi / 2 - (theta_u - theta))
                theta2 =  np.pi / 2 + theta1
                V2prime_c = v1 * np.cos(theta2) * cmath.exp(1j*U_c[1]) # vitesse complexe de la boule à l'arrêt après le choc
                V1prime_c = v1 * np.cos(theta1) * cmath.exp(1j*(theta + theta1))
        elif P1[0] > P2[0] and P1[1] > P2[1] and V1[1] > 0:
                theta1 = - ( np.pi / 2 -(theta_u - theta))
                theta2 =  np.pi / 2 + theta1
                V2prime_c = v1 * np.cos(theta2) * cmath.exp(1j*U_c[1]) # vitesse complexe de la boule à l'arrêt après le choc
                V1prime_c = v1 * np.cos(theta1) * cmath.exp(1j*(theta + theta1))
        elif P1[0] > P2[0] and P1[1] > P2[1] and V1[1] < 0:
                theta1 = np.pi / 2 -(theta - theta_u)
```

```
                theta2 = - np.pi / 2 + theta1
                V2prime_c = v1 * np.cos(theta2) * cmath.exp(1j*U_c[1]) # vitesse complexe de la boule à l'arrêt après le choc
                V1prime_c = v1 * np.cos(theta1) * cmath.exp(1j*(theta + theta1))
        elif P1[0] > P2[0] and P1[1] < P2[1] and V1[1] > 0:
                theta1 = - ( np.pi / 2 - (theta_u - theta))
                theta2 =  np.pi / 2 + theta1
                V2prime_c = v1 * np.cos(theta2) * cmath.exp(1j*U_c[1]) # vitesse complexe de la boule à l'arrêt après le choc
                V1prime_c = v1 * np.cos(theta1) * cmath.exp(1j*(theta + theta1))
        elif P1[0] > P2[0] and P1[1] < P2[1] and V1[1] < 0:
                theta1 = np.pi / 2 - (theta - theta_u)
                theta2 = - np.pi / 2 + theta1
                V2prime_c = v1 * np.cos(theta2) * cmath.exp(1j*U_c[1]) # vitesse complexe de la boule à l'arrêt après le choc
                V1prime_c = v1 * np.cos(theta1) * cmath.exp(1j*(theta + theta1))
        return (np.array([V1prime_c.real, V1prime_c.imag]), np.array([V2prime_c.real, V2prime_c.imag]))


## Fonction principale : 'trajectoires'
dt = 1e-5 # temps d'échantillonnage
def trajectoires(P0_blanche, V0_blanche, P0_rouge, P0_jaune):
        global Long_bi, Larg_bi, e, dt
        t = 0
        # dictionnaire intermédiaire contenant les informations des boules entre deux collisions :
        boules = {
                'blanche': {'position': P0_blanche, 'vitesse': V0_blanche, 'temps': t},
                'rouge': {'position': P0_rouge, 'vitesse': np.array([0, 0]), 'temps': t},
                'jaune': {'position': P0_jaune, 'vitesse': np.array([0, 0]), 'temps': t}
        }
        # dictionnaire contenant toutes les informations sur les trajectoires des boules :
        trajectoires_boules = {
                'blanche': {'positions': [P0_blanche], 'vitesses': [V0_blanche], 'collisions' : 0},
                'rouge': {'positions': [P0_rouge], 'vitesses': [np.array([0, 0])], 'collisions' : 0},
                'jaune': {'positions': [P0_jaune], 'vitesses': [np.array([0, 0])], 'collisions' : 0}
        }
        while any(np.linalg.norm(boules[key]['vitesse']) > 1e-2 for key in boules.keys()): # au moins une boule est en mouvement
                for key in boules.keys():
                        if np.linalg.norm(boules[key]['vitesse']) > 1e-2:
                                # on actualise la position et la vitesse de la boule :
                                P1 = avance(boules[key]['vitesse'], boules[key]['temps'], boules[key]['position'])
                                V1 = vitesse(boules[key]['vitesse'], boules[key]['temps'])
```

```
                boules[key]['vitesse'] = V1

                # Gestion des collisions entre boules

                for other_key in boules.keys():

                        if key != other_key and np.linalg.norm(boules[other_key]['vitesse']) < 1e-2:

                                if np.linalg.norm(P1 - boules[other_key]['position']) <= 2 * Rb:

                                        # on actualise les vitesses des deux boules :

                                        boules[key]['vitesse'], boules[other_key]['vitesse'] = collision(P1, boules[other_key]['position'],    V1)

                                        trajectoires_boules[key]['positions'].append(boules[key]['position'])

                                        trajectoires_boules[key]['vitesses'].append(boules[key]['vitesse'])

                                        trajectoires_boules[other_key]['vitesses'].append(boules[other_key]['vitesse'])

                                        # on réinitialise l'échelle de temps des deux boules :

                                        boules[key]['temps'] = 0

                                        boules[other_key]['temps'] = 0

                                        # On relève cette collision :

                                        trajectoires_boules[key]['collisions'] += 1

                                        trajectoires_boules[other_key]['collisions'] += 1

                # Gestion des collisions avec les parois

                if np.size(P1) >= 4 :

                        P1 = P1[0,:] # slicing pour garder uniquement des vecteurs en cas de problème

                if P1[0] <= Rb or P1[0] >= Larg_bi - Rb : # collision avec les parois latérales

                        boules[key]['vitesse'] = np.array([-boules[key]['vitesse'][0] * e, boules[key]['vitesse'][1] * e])

                        boules[key]['position'] = boules[key]['position']

                        trajectoires_boules[key]['positions'].append(boules[key]['position'])

                        trajectoires_boules[key]['vitesses'].append(boules[key]['vitesse'])

                        boules[key]['temps'] = 0

                elif P1[1] <= Rb or P1[1] >= Long_bi - Rb: # collision avec les parois frontales

                        boules[key]['vitesse'] = np.array([boules[key]['vitesse'][0] * e, -boules[key]['vitesse'][1] * e])

                        boules[key]['position'] = boules[key]['position']

                        trajectoires_boules[key]['positions'].append(boules[key]['position'])

                        trajectoires_boules[key]['vitesses'].append(boules[key]['vitesse'])

                        boules[key]['temps'] = 0

                else : # pas de collisions avec les parois

                        # Mise à jour de la position

                        boules[key]['position'] = P1

                        # Ajout des données pour la trajectoire

                        trajectoires_boules[key]['positions'].append(P1)

                        trajectoires_boules[key]['vitesses'].append(V1)

                        boules[key]['temps'] += dt

return trajectoires_boules
```

```
## base de données : création, ajout, récupération

# nom de la base de données :
nom_database = str(input("Entrez un nom pour la base de données : ")) + '.db'

def creer_bdd(database):
        # Connexion à la base de données (ou création si elle n'existe pas)
        conn = sqlite3.connect(database)
        cursor = conn.cursor()
        # Création de la table 'positions'
        cursor.execute('''
                CREATE TABLE IF NOT EXISTS positions (
                        id_pos INTEGER PRIMARY KEY AUTOINCREMENT,
                        pos_ini_blanche TEXT,
                        pos_ini_rouge TEXT,
                        pos_ini_jaune TEXT
                )
        ''')
        # Création de la table 'coups'
        cursor.execute('''
                CREATE TABLE IF NOT EXISTS coups (
                        id_coup INTEGER PRIMARY KEY AUTOINCREMENT,
                        id_pos INTEGER, # clef étrangère qui permet de savoir à quelle position initiale correspond le coup
                        V0_blanche TEXT,
                        FOREIGN KEY(id_pos) REFERENCES positions(id_pos)
                )
        ''')
        # Création de la table 'verif'
        cursor.execute('''
                CREATE TABLE IF NOT EXISTS verif (
                        id_verif INTEGER PRIMARY KEY AUTOINCREMENT,
                        id_pos INTEGER, # clef étrangère
                        id_coup INTEGER, clef étrangère
                        collisions_blanche INTEGER, # relève le nombre de collisions de la boule blanche
                        collisions_rouge INTEGER,
                        collisions_jaune INTEGER,
                        coup_valide BOOLEAN, # indicateur de validité du coup
                        FOREIGN KEY(id_pos) REFERENCES positions(id_pos),
                        FOREIGN KEY(id_coup) REFERENCES coups(id_coup)
                )
```

```
        ''')
        conn.commit()
        conn.close()


# fonction qui permet d'ajouter des coups à la base de données :
def ajouter_position_et_coup(database, pos_ini_blanche, pos_ini_rouge, pos_ini_jaune, V0_blanche):
        conn = sqlite3.connect(database)
        cursor = conn.cursor()
        # relever le nombre de collisions de chaque boule :
        collisions_blanche = trajectoires(pos_ini_blanche, V0_blanche, pos_ini_rouge, pos_ini_jaune)['blanche']['collisions']
        collisions_rouge = trajectoires(pos_ini_blanche, V0_blanche, pos_ini_rouge, pos_ini_jaune)['rouge']['collisions']
        collisions_jaune = trajectoires(pos_ini_blanche, V0_blanche, pos_ini_rouge, pos_ini_jaune)['jaune']['collisions']
        # validité du coup :
        if collisions_blanche != collisions_rouge + collisions_jaune : # les boules rouges et jaunes se sont touchées
                coup_valide = 0
        elif collisions_blanche == 0 or collisions_jaune == 0 or collisions_rouge == 0 : # une boule n'est pas touchée
                coup_valide = 0
        else :
                coup_valide = 1
        # Insertion dans la table 'positions'
        cursor.execute('''
                INSERT INTO positions (pos_ini_blanche, pos_ini_rouge, pos_ini_jaune)
                VALUES (?, ?, ?)
        ''', (str(pos_ini_blanche), str(pos_ini_rouge), str(pos_ini_jaune))) # insertion sous forme de chaine de caractères
        id_pos = cursor.lastrowid  # Récupérer l'ID de la position ajoutée pour pouvoir l'insérer dans les autres tables
        # Insertion dans la table 'coups'
        cursor.execute('''
                INSERT INTO coups (id_pos, V0_blanche)
                VALUES (?, ?)
        ''', (id_pos, str(V0_blanche)))
        id_coup = cursor.lastrowid  # récupérer l'ID du coup inséré
        # Insertion dans la table 'verif'
        cursor.execute('''
                INSERT INTO verif (id_pos, id_coup, collisions_blanche, collisions_rouge, collisions_jaune, coup_valide)
                VALUES (?, ?, ?, ?, ?, ?)
        ''', (id_pos, id_coup, collisions_blanche, collisions_rouge, collisions_jaune, coup_valide))
        conn.commit()
        conn.close()
```

```
creer_bdd(nom_database) # créer la base de données


# Fonction pour convertir une chaine de caractères en tableau NumPy
def string_to_array(c):
    return np.fromstring(c.strip('[]'), sep=' ')


def recuperer_coups(database):
    conn = sqlite3.connect(database)
    cursor = conn.cursor()
    # Requête pour récupérer les coups, leurs positions et la vérification
    cursor.execute('''
        SELECT c.id_coup, c.V0_blanche, p.pos_ini_blanche, p.pos_ini_rouge, p.pos_ini_jaune, v.collisions_blanche, v.collisions_rouge, v.collisions_jaune
        FROM coups c
        JOIN positions p ON c.id_pos = p.id_pos
        JOIN verif v ON c.id_coup = v.id_coup WHERE v.coup_valide = 1 ;
    ''')
    coups_valides = cursor.fetchall()
    conn.close()
    # Convertir les chaines de caractères en tableaux NumPy
    coups_valides = [(coup[0], string_to_array(coup[1]), string_to_array(coup[2]), string_to_array(coup[3]), string_to_array(coup[4]), coup[5], coup[6], coup[7]) for coup in coups_valides]
    return coups_valides
```

Positions initiales aléatoires :

```
# Vérifier que les positions initiales sont bien deux à deux distinctes :
def verifier_positions_initiales(pos_blanche, pos_rouge, pos_jaune):
    if np.linalg.norm(pos_blanche - pos_rouge) < 2 * Rb:
        return False
    if np.linalg.norm(pos_blanche - pos_jaune) < 2 * Rb:
        return False
    if np.linalg.norm(pos_rouge - pos_jaune) < 2 * Rb:
        return False
    return True
```

```
# Générer des positions aléatoires
while True:
        rand_bx = random.uniform(0.0615, 1.3615) # abscisse boule blanche
        rand_rx = random.uniform(0.0615, 1.3615) # abscisse boule rouge
        rand_jx = random.uniform(0.0615, 1.3615) # abscisse boule jaune
        rand_by = random.uniform(0.0615, 2.7845) # ordonnée boule blanche
        rand_ry = random.uniform(0.0615, 2.7845) # ordonnée boule rouge
        rand_jy = random.uniform(0.0615, 2.7845) # ordonnée boule jaune
        pos_ini_blanche = np.array([rand_bx, rand_by])
        pos_ini_rouge = np.array([rand_rx, rand_ry])
        pos_ini_jaune = np.array([rand_jx, rand_jy])
        if verifier_positions_initiales(pos_ini_blanche, pos_ini_rouge, pos_ini_jaune):
                break
        else:
                print("Positions invalides, génération de nouvelles positions...")

# afficher les positions initiales aléatoires retenues :
print("Positions initiales validées:", pos_ini_blanche, pos_ini_rouge, pos_ini_jaune)
```

Échantillonnage des vitesses tests :

```
V = [] # liste des vitesses échantillonnées
Lambda = 15 # valeur de la norme de la vitesse
for k in range(60):
        V.append(np.array([coef * np.cos( (k*np.pi)/30), coef * np.sin( (k*np.pi)/30)]))
# remplissage de la base de données :
for v0 in V :
        ajouter_position_et_coup(nom_database, pos_ini_blanche, pos_ini_rouge, pos_ini_jaune, v0)

# récupération des coups valides :
coups_valides = recuperer_coups(nom_database)
print("Les coups valides sont : ", coups_valides)
```

```
## Traitement des sous coups valides : permet d'avoir un coup d'avance

Liste = []

# fonction qui permet de récupérer le maximum ainsi que son indice dans la liste :

def recup_max_et_ind(L):
        L2 = []
        max = L[0][1]
        for i in range(1, len(L)):
                if L[i][1] >= max :
                        max = L[i][1]
        for k in range(len(L)):
                if L[k][1] == max :
                        L2.append(L[k][0])
        return max, L2


for coup in coups_valides :
        trajectoires_results = trajectoires(coup[2], coup[1], coup[3], coup[4])
        dico_pos_finale = {'blanche' : [], 'rouge' : [], 'jaune' : []}
        for key in trajectoires_results.keys():
                positions = np.array(trajectoires_results[key]['positions'])
                dico_pos_finale[key] = positions[-1] #  on récupère les positions finales des boules à l'issue du coup
        # création d'une base de données pour chaque coup valide
        creer = creer_bdd(str(coup[0])+'.db')
        # remplir la sous base de données
         for v0 in V :
                ajouter_position_et_coup(str(coup[0])+'.db', dico_pos_finale['blanche'], dico_pos_finale['rouge'], dico_pos_finale['jaune'], v0)
        # récupérer le nombre de sous coups valide (de la sous base)
        nombre_sous_coups_valides = len(recuperer_coups(str(coup[0])+'.db'))
        Liste.append([coup[0], nombre_sous_coups_valides])
        print([coup[0], nombre_sous_coups_valides])


max, L_indices_valides = recup_max_et_ind(Liste)
print("Le maximum de coups valides au tour suivant est :", max)
print("Les indices des coups valides qui admettent un maximum de sous coups valides sont :", L_indices_valides)
# temps d'exécution :
end_time = time.time() # fin de la mesure du temps
execution_time = end_time - start_time
print(f"Le programme a mis {execution_time:.2f} secondes à s'exécuter.")
```

Annexe : Collision entre deux boules

On note $\mathcal{S}$ le système formé par les deux boules.

- Dans le référentiel du laboratoire $\mathcal{R}_0$ supposé Galiléen, on a d'après le théorème du centre d'inertie :

$$\frac{d\overrightarrow{p_S}}{dt} = \sum \overrightarrow{F_{ext}} = \vec{0} \Rightarrow \overrightarrow{p_{S,i}} = \overrightarrow{p_{S,f}}$$

- L'énergie totale du système est :

$$\mathcal{E} = \mathcal{E}_{c,S} + \mathcal{E}_{p,S}^{int} + U_S = \mathcal{E}_{c,S}$$

  Comme le système est isolé de l'extérieur alors l'énergie est conservée :

$$\mathcal{E}_{c,S,i} = \mathcal{E}_{c,S,f}$$

En supposant que les boules ont la même masse, on a :

$$\begin{cases} \overrightarrow{v_{1,i}} = \overrightarrow{v_{1,f}} + \overrightarrow{v_{2,f}} \\ v_{1,i}^2 = v_{1,f}^2 + v_{2,f}^2 \end{cases}$$

La première équation élevée au carré donne :

$$v_{1,i}^2 = v_{1,f}^2 + v_{2,f}^2 + 2\overrightarrow{v_{1,f}} \cdot \overrightarrow{v_{2,f}}$$

D'où en soustrayant avec l'autre équation du système :

$$\boldsymbol{\overrightarrow{v_{1,f}} \cdot \overrightarrow{v_{2,f}} = 0}$$

➢ Les deux vecteurs vitesses forment un angle droit : l'angle à l'issue d'une collision entre deux boules vaut **90°**

On multiplie maintenant la première équation par $\overrightarrow{v_{1,f}}$ et on obtient :

$$\boldsymbol{v_{1,f} = v_{1,i}\cos(\theta_1)}$$

De même :

$$\boldsymbol{v_{2,f} = v_{1,i}\cos(\theta_2)}$$