Étude et optimisation des coups au billard français

Thomas CANISARES–MORÈRE

N°SCEI 23838
Positionnements thématiques
PHYSIQUE (*Mécanique*)
INFORMATIQUE (*Informatique pratique*)

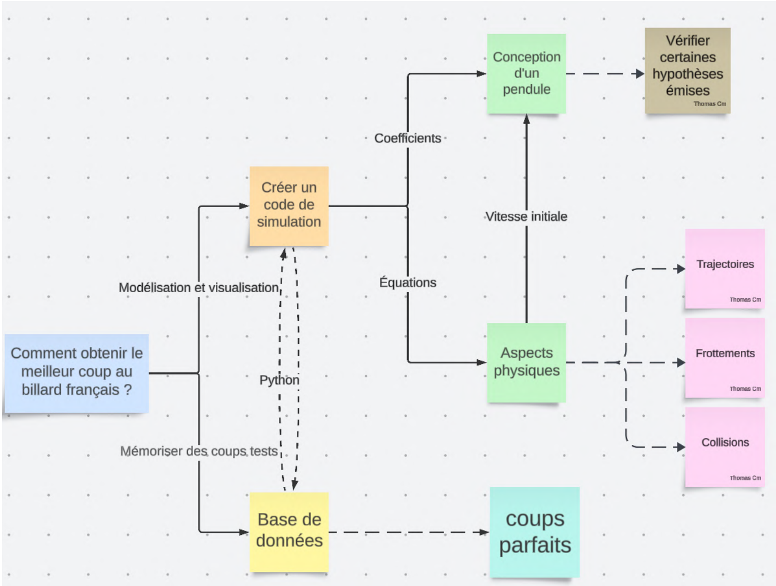
2023-2024

Plan

Présentation du billard français

Figure 1.1 – Photographie d'une table de billard français (sans poches)

problématique et objectifs



Définition du coup parfait

Définition

Le coup parfait dépend des positions des boules sur la table ainsi que des règles spécifiques au jeu. Le joueur doit :

- frapper la bille blanche avec **la bonne force**, **la bonne direction** et **le bon angle**,
- obtenir **une position idéale pour le prochain coup**,
- **gêner l'adversaire dans son jeu**.

Intérêts d'un programme de simulation et d'optimisation du billard français

1. **Pour le grand public :**
   - Apprendre à jouer.
   - Un outil d'entrainement précieux.
   - Développer des compétences cognitives.
   - Accessible à tous.

2. **Pour les joueurs expérimentés :**
   - Perfectionnement des compétences.
   - Référence rapide.
   - Validation des choix des coups.
   - Exploration des nouvelles stratégies.

Les coups de base

- **Le carreau direct.**
- **Le carreau indirect.**
- Le coup de masse.
- Le coup de trois ou quatre bandes.
- Le coup de défense.

Coups non pris en compte dans notre étude

Le coup de masse ainsi que le coup de défense ne seront pas pris en compte.

Plan

Hypothèses

<u>**Pour les collisions des boules avec les bandes :**</u>

- Les boules vérifient **la loi de réflexion de Descartes**,
- on considère un **coefficient de restitution**.

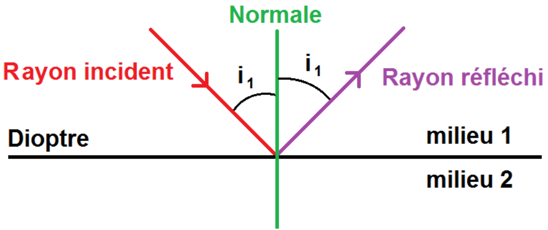
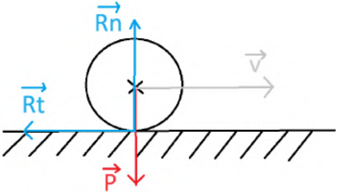


Figure 2.1 – Illustration de la loi de réflexion de Descartes

On néglige la phase de glissement de la boule

trajectoire de la boule



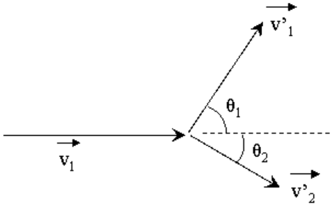
Principe fondamental de la dynamique : $m\vec{a} = -mg\,\vec{u_z} - R_t \frac{\vec{v_0}}{||\vec{v_0}||} + R_n \vec{u_z}$

On obtient le système : $\begin{cases} m\vec{a} = -R_t \frac{\vec{v_0}}{||\vec{v_0}||} \\ R_n = mg \\ R_t = f \times R_n \end{cases}$

Équation de la trajectoire d'une boule

$$\vec{OM}(t) = -\frac{1}{2}gft^2 \frac{\vec{v_0}}{||\vec{v_0}||} + \vec{v_0}t + \vec{OM}(t=0)$$

Physique des collisions



Résultat 1

Les vecteurs vitesse des deux boules à l'issue de la collision forment un angle droit : $\theta_1 - \theta_2 = \frac{\pi}{2}$

Résultat 2

$$\begin{cases} v_{1,f} = v_{1,i} \times cos(\theta_1) \\ v_{2,f} = -v_{1,i} \times sin(\theta_1) \end{cases}$$

Plan

Le maillet : centre d'inertie

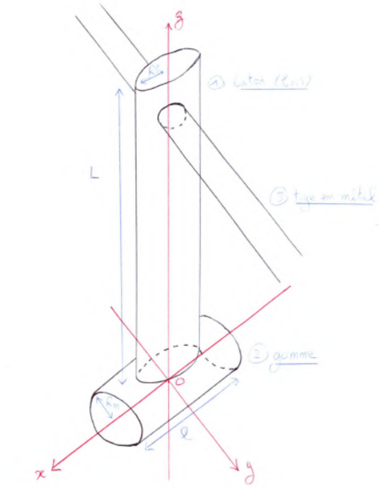


Figure 3.1 – dessin du maillet percé

On note $G(0,0,z_G)$ le centre d'inertie du maillet :

$$z_G = \frac{m_{baton} \times (R_m + \frac{L}{2})}{m_{baton} + m_{gomme}}$$

$$z_G = \frac{\rho_{baton} \pi {R_b}^2 L (R_m + \frac{L}{2})}{\rho_{baton} \pi {R_b}^2 L + \rho_{gomme} \pi {R_m}^2 l}$$

Altitude du centre d'inertie

$$z_G = 4,735 \text{ cm}$$

Conception du pendule

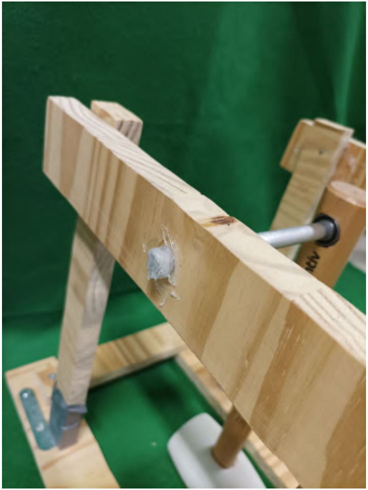


Figure 3.2 – tige en métal utilisée

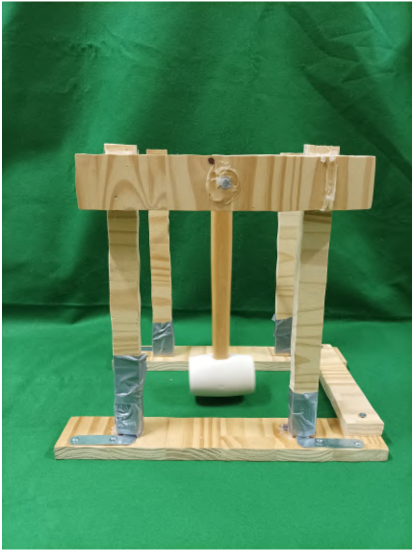
Figure 3.3 – Roulement à bille

Conception du pendule

Figure 3.4 – Fixation des pieds

Figure 3.5 – Cales en bois

Conception du pendule



Relation liant l'angle du maillet et la vitesse initiale de la boule

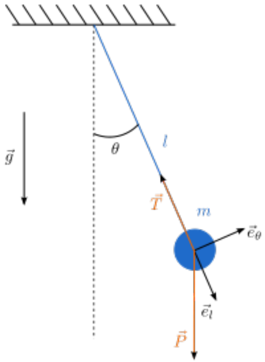


Figure 3.6 – Schéma du pendule

conservation de l'énergie mécanique

$$V_0 = K \times \sqrt{1 - \cos(\theta)}$$

où $K = \sqrt{2g(L + \frac{R_m}{2} - z_G)} = 2.26\ \text{m/s}$

Relation liant l'angle du maillet et la vitesse initiale de la boule

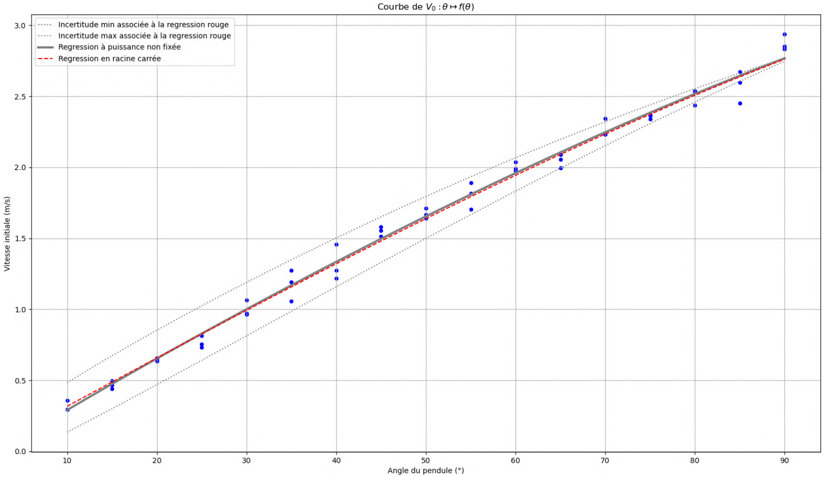


Figure 3.7 – Résultats expérimentaux

Relation liant l'angle du maillet et la vitesse initiale de la boule

Modèle regressi avec le plus faible écart type :

$$V_0 = K \times (1 - \cos(\theta))^p$$

où : $\begin{cases} K = 2.82 m.s^{-1} \\ p = (475 \pm 51) \times 10^{-3} \end{cases}$

Conclusion

On obtient une courbe expérimentale très proche du modèle théorique.

Plan

Angle de 90 degrés à l'issue d'une collisions entre boules

Numéro de la tentative	1	2	3	4	5	6	7
Angle à l'issue de la collision (degrés)	78	71	84	88	79	77	89

Figure 4.1 – Pointage

Collisions des boules avec les bandes

Angle du pendule (degrés)	coefficient de restitution
20	0.539
30	0.486
40	0.757
50	0.514
60	0.863
70	0.692
80	0.812
90	0.779

Angle d'incidence de la boule (degrés)	10	20	40	50	60	70
coefficient de restitution	0.814	0.708	0.566	0.691	0.68	0.797

Coefficient de restitution

$$\epsilon = 0.7$$

Coeffecient de frottement de la feutrine

Angle du pendule (degrés)	**coefficient de frottement**
15	0.0163
30	0.0183
45	0.0173
60	0.0183
75	0.0204

Coefficient de frottement

f = 0.018

Figure 4.2 – Vitesse en fonction du temps

Vérification de la loi de Descartes (réflexion des boules)

Angle d'incidence (degrés)	Angle réfléchi (degrés)
17.5	5
30	15
45	33
62	57

Figure 4.3 – Angle d'incidence : 17.5 degrés (petit angle)

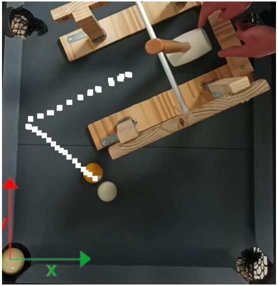


Figure 4.4 – Angle d'incidence : 62 degrés (grand angle)

Résumé des vérifications et expérimentations

Critère	Conclusion
Angle de 90 degrés après une collision entre boules	Vérifié
Coefficient de frottement de la feutrine f	**0.018**
Coefficient de restitution ϵ	**0.7**
Réflexion selon la loi de Descartes	Vérifié uniquement pour les grands angles

Coefficients utiles

On retiendra en particulier $f = 0.018$ et $\epsilon = 0.7$

Loi de Descartes

On ne tiendra pas compte du caractère non vérifiable de la loi de Descartes pour les petits angles.

Plan

Préliminaires

```python
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)))
```

Figure 5.1 – fonctions utiles

On doit coder :

1. une fonction "**collision**" qui décrira les collisions entre boule,
2. une fonction "**trajectoires**" qui décrira l'évolution des boules.

Bibliothèques utilisées

numpy, cmath et matplolib.pyplot

Affichage

```python
## Affichage :
P0_blanche = np.array([D + D2, D])
P0_rouge = np.array([D, 3 * D])
P0_jaune = np.array([D, D])

# contour de la table sans epaisseur :
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')
# contour de la table avec epaisseur :
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')

t = np.linspace(0, 2* np.pi, 1000)
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 aléatoires")
plt.axis('equal')

plt.show()
```

Figure 5.2 – Code de l'affichage

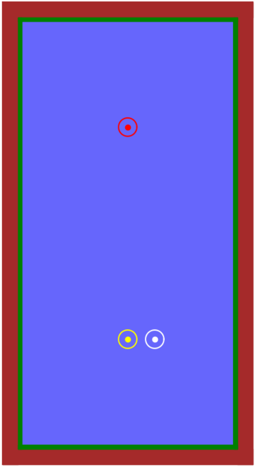


Figure 5.3 – Affichage de la position initiale

Fonction "boule"

Utilisation des listes

Critère n°1

La fonction principale de notre code doit décrire :

- le comportement en ligne droite de la boule,
- les collisions avec les bandes.

```python
def boule(P0, V0):
    global Long_bi, Larg_bi, e, t, dt, Rb
    V, P = [], [] # liste des positions et vitesses
    P.append(P0)
    V.append(V0)
    V_norme = np.linalg.norm(V0)
    t = dt
    i = 0
    T = [0]
    Liste_V_norme = [V_norme]
    while V_norme > 1e-2 :
        P1 = avance(V0, t, P0)
        V1 = vitesse(V0, t)
        if P1[0] <= Rb or P1[0] >= Larg_bi - Rb :
            V0 = np.array([-V[i-1][0]*e, V[i-1][1]*e])
            P0 = P[i-1]
            P.append(P0)
            V.append(V0)
            T.append(T[-1]+dt)
            t = 0
            V_norme = np.linalg.norm(V0)
        elif P1[1] <= Rb or P1[1] >= Long_bi - Rb :
            V0 = np.array([V[i-1][0]*e, -V[i-1][1]*e])
            P0 = P[i-1]
            P.append(P0)
            V.append(V0)
            T.append(T[-1]+dt)
            t = 0
            V_norme = np.linalg.norm(V0)
        else :
            P.append(P1)
            V.append(V1)
            V_norme = np.linalg.norm(V1)
            T.append(T[-1]+dt)
        i += 1
        t += dt
        Liste_V_norme.append(V_norme)
    return P, V, T, Liste_V_norme
```

Figure 5.4 – Code fonction "boule"

Fonction "boule"

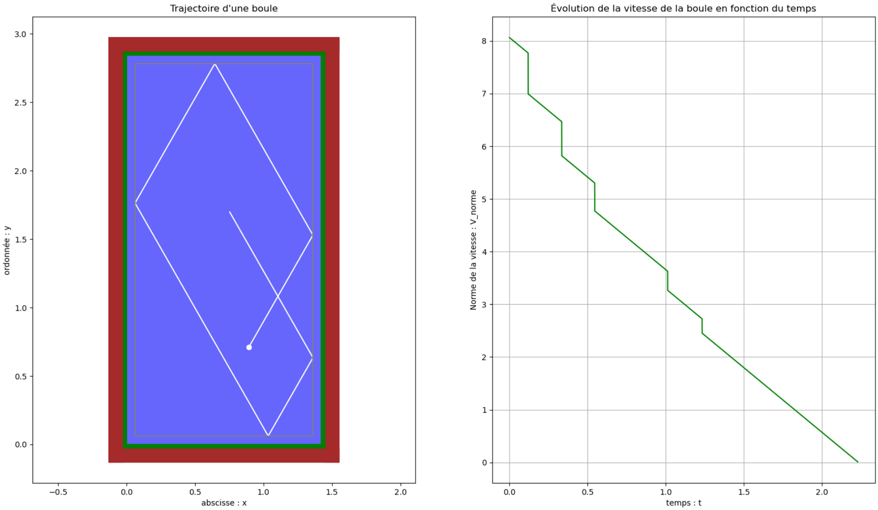


Figure 5.5 – Affichage de la trajectoire d'une seule boule

Fonction principale : "trajectoires"

Utilisation de dictionnaires

```python
def trajectoires(P0_blanche, V0_blanche, P0_rouge, P0_jaune):
    global Long_bi, Larg_bi, e, dt

    t = 0

    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}
    }

    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}
    }
```

```python
while any(np.linalg.norm(boules[key]['vitesse']) > 1e-2 for key in boules.keys()):

    for key in boules.keys():
        if np.linalg.norm(boules[key]['vitesse']) > 1e-2:
            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:

                        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]['positions'].append(P1)
                        trajectoires_boules[key]['vitesses'].append(boules[key]['vitesse'])
                        trajectoires_boules[other_key]['vitesses'].append(boules[other_key]['vitesse'])

                        boules[key]['temps'] = 0
                        boules[other_key]['temps'] = 0

                        # On relève cette collision entre boules
                        trajectoires_boules[key]['collisions'] += 1
                        trajectoires_boules[other_key]['collisions'] += 1
```

```python
            # Gestion des collisions avec les parois
            if np.size(P1) >= 4 :
                P1 = P1[0,:]

            if P1[0] <= Rb or P1[0] >= Larg_bi - Rb :
                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:
                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 : # si pas de collisions avec les parois
                # Mise à jour des positions et vitesses
                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
```

Figure 5.6 – Code de la fonction "trajectoires"

Fonction "collision"

```python
def collision(P1, P2, V1):
    dist = P2 - P1
    u = dist / np.linalg.norm(dist)
    v = np.array([1, 0])
    theta = np.arccos(np.dot(V1, v) / np.linalg.norm(V1))
    b = np.dot(dist, v) * v
    a = b / (2 * Rb)
    theta1 = np.arccos(a)
    theta2 = np.pi / 2 - theta1
    V2prime = np.linalg.norm(V1) * np.cos(theta2) * u
    alpha = np.arccos(V2prime[0] / np.linalg.norm(V2prime))
    V1prime = np.linalg.norm(V1) * a * np.array([np.cos(alpha), np.sin(alpha)])
    return V1prime, V2prime
```

Figure 5.7 – Première fonction collision

Conclusion

Il faut réaliser une disjonction de cas et étudier toutes les collisions envisageables.

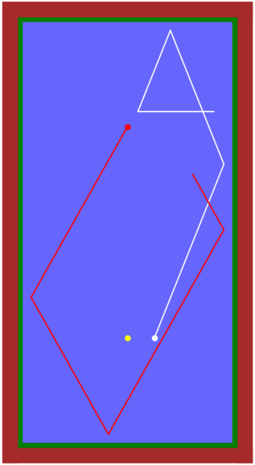


Figure 5.8 – Premier affichage des collisions

Fonction "collision"

```python
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'arr
        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'arr
        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'arr
        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'arr
        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'arr
        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'arr
        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'arr
        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]))
```

Figure 5.9 – fonction collision finale

Exemple d'une simulation

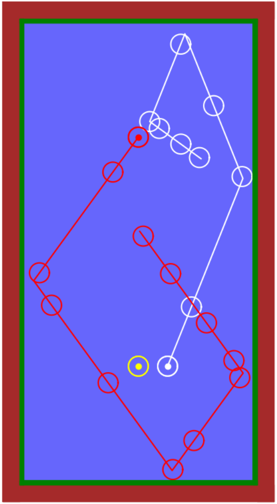


Figure 5.10 – une simulation

Plan

Utilisation de bases de données

SQLite Studio

Pourquoi une base de données ?

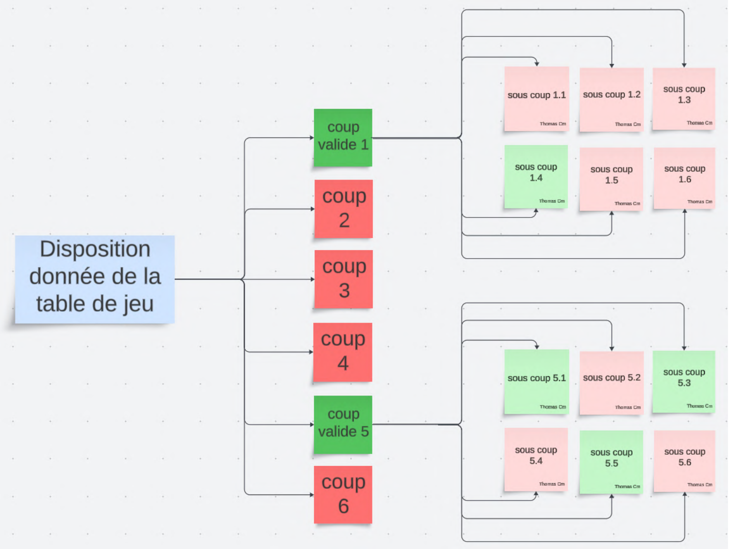
Une BDD nous permet de :

- stocker un grand nombre de coups pour une position donnée,
- récupérer les coups valides.

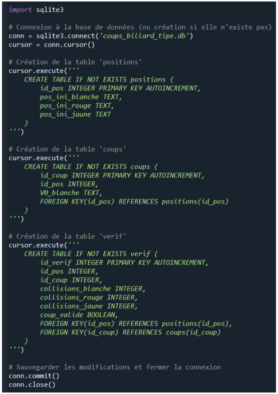
On doit donc :

- Créer notre base de données,
- Y ajouter des coups,
- Récupérer les coups valides.

Utilisation de bases de données



Utilisation de bases de données



```python
import sqlite3

# Connexion à la base de données (ou création si elle n'existe pas)
conn = sqlite3.connect('coups_billard_tipe.db')
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,
        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,
        id_coup INTEGER,
        collisions_blanche INTEGER,
        collisions_rouge INTEGER,
        collisions_jaune INTEGER,
        coup_valide BOOLEAN,
        FOREIGN KEY(id_pos) REFERENCES positions(id_pos),
        FOREIGN KEY(id_coup) REFERENCES coups(id_coup)
    )
''')

# Sauvegarder les modifications et fermer la connexion
conn.commit()
conn.close()
```

Figure 6.1 – Création de notre base de données avec python

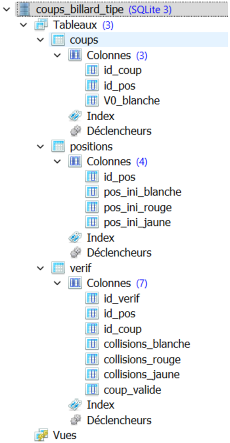


Figure 6.2 – Sommaire de notre base de données

Utilisation de bases de données

```python
def ajouter_position_et_coup(pos_ini_blanche, pos_ini_rouge, pos_ini_jaune, V0_blanche):
    # Connexion à la base de données
    conn = sqlite3.connect('coups_billard_tipe.db')
    cursor = conn.cursor()

    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']

    if collisions_blanche != collisions_rouge + collisions_jaune :
        coup_valide = 0
    elif collisions_blanche == 0 or collisions_jaune == 0 or collisions_rouge == 0 :
        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)))

    id_pos = cursor.lastrowid  # Récupérer l'ID de la position insérée

    # 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))

    # Sauvegarder les modifications et fermer la connexion
    conn.commit()
    conn.close()
```

Figure 6.3 – Fonction d'ajout des coups et positions

```python
def recuperer_coups():
    # Connexion à la base de données
    conn = sqlite3.connect('coups_billard_tipe.db')
    cursor = conn.cursor()

    # Requête pour récupérer les coups, leurs positions et la vérification
    cursor.execute('''
        SELECT 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 coup_valide = 1 ;
    ''')
    coups_valides = cursor.fetchall()

    # Fermer la connexion
    conn.close()

    return coups_valides
```

Figure 6.4 – Fonction de récupération des coups valides

Exemple pour une disposition donnée

Bibliothèque **random** :

- Position initiale boule blanche : $[0.96698804, 0.65669534]$
- Position initiale boule rouge : $[0.2285659, 0.40666086]$
- Position initiale boule jaune : $[0.57550489, 2.77659158]$

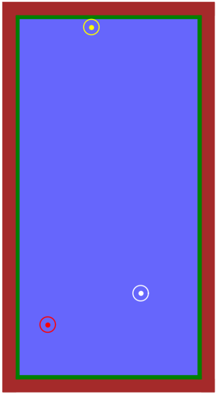


Figure 6.5 – Affichage des positions initiales aléatoires

On obtient 5 coups valides :

- coup d'indice 8 : $v_0 = [11.14717238, 10.0369591]$
- coup d'indice 22 : $v_0 = [-8.81677878, 12.13525492]$
- coup d'indice 35 : $v_0 = [-13.70318186, -6.10104965]$
- coup d'indice 49 : $v_0 = [4.63525492, -14.26584774]$
- coup d'indice 60 : $v_0 = [14.91782843, -1.56792695]$

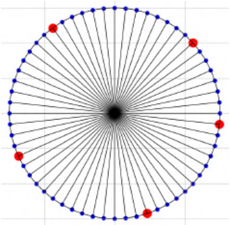


Figure 6.6 – Représentation en rouge des vitesses initiales permettant d'obtenir un coup valide

Exemple pour une disposition donnée

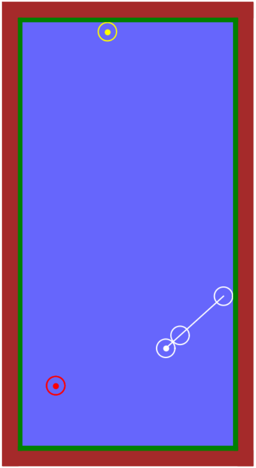


Figure 6.7 – Représentation du premier coup valide

Exemple pour une disposition donnée

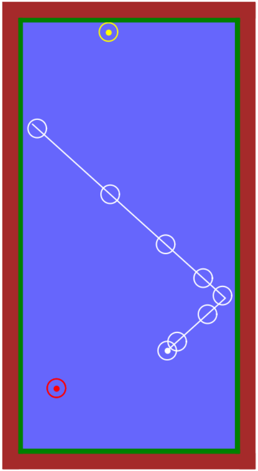


Figure 6.8 – Représentation du premier coup valide

Exemple pour une disposition donnée

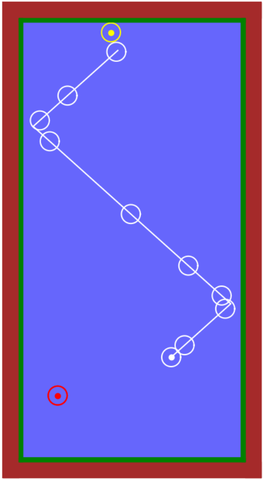


Figure 6.9 – Représentation du premier coup valide

Exemple pour une disposition donnée

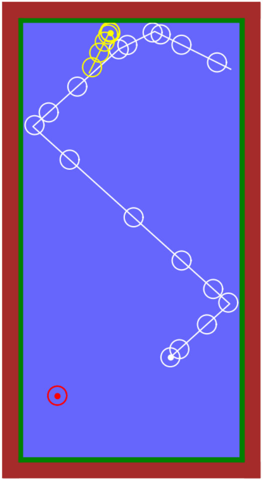


Figure 6.10 – Représentation du premier coup valide

Exemple pour une disposition donnée

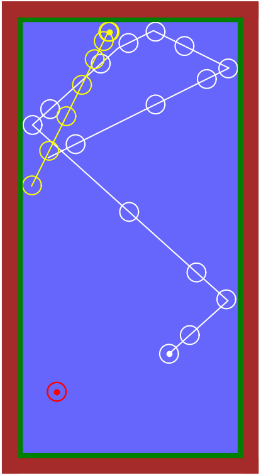


Figure 6.11 – Représentation du premier coup valide

Exemple pour une disposition donnée

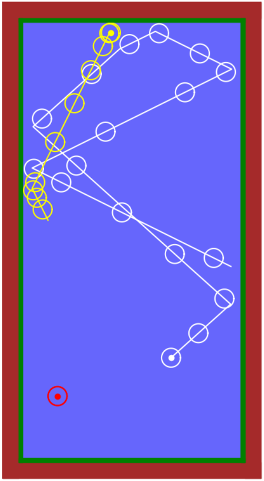


Figure 6.12 – Représentation du premier coup valide

Exemple pour une disposition donnée

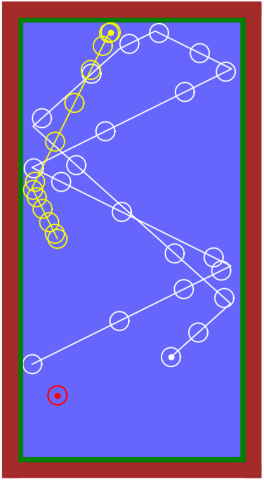


Figure 6.13 – Représentation du premier coup valide

Exemple pour une disposition donnée

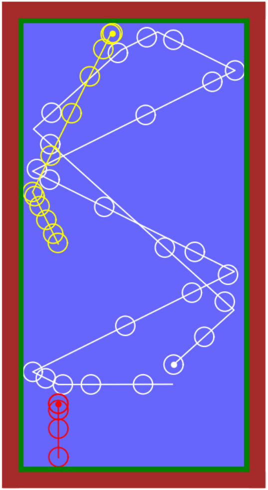


Figure 6.14 – Représentation du premier coup valide

Exemple pour une disposition donnée

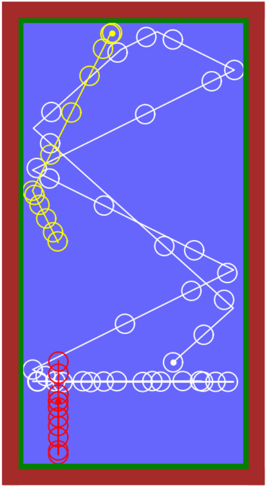


Figure 6.15 – Représentation du premier coup valide

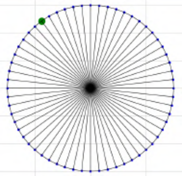
Exemple pour une disposition donnée

Les nombres de sous coups valides associés :

- indice 8 : **16** sous coups valides
- indice 22 : **48** sous coups valides
- indice 35 : **16** sous coups valides
- indice 49 : **16** sous coups valides
- indice 60 : **19** sous coups valides

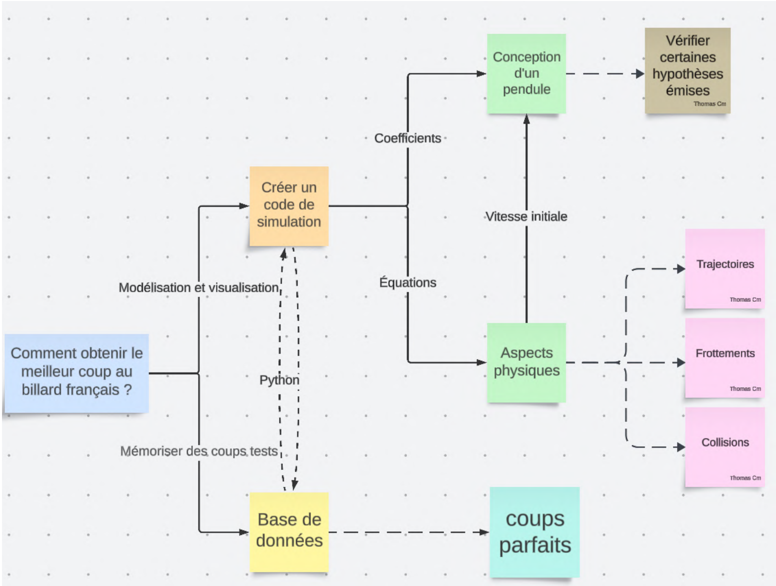
Le meilleur coup

Coup d'indice 22 : $v_0 = [-8.81677878, 12.13525492]$



Plan

Bilan



Vérifier certaines hypothèses émises
Thomas Cm
Conception d'un pendule
Coefficients
Créer un code de simulation
Vitesse initiale
Trajectoires
Thomas Cm
Modélisation et visualisation
Équations
Comment obtenir le meilleur coup au billard français ?
Aspects physiques
Frottements
Thomas Cm
Python
Collisions
Thomas Cm
Mémoriser des coups tests
Base de données
coups parfaits

Perspectives d'amélioration

1. Concernant l'aspect physique :
   - angle pas vraiment égal à 90 degrés après une collision,
   - loi de Descartes non valide pour les grands angles,
   - tenir compte du glissement de la boule.
2. Il faudrait également **réaliser un plus grand nombre de tests pour déterminer e et f**.
3. Concernant la partie avec les bases de données :
   - Optimiser le code, le réduire en complexité ;
   - un plus grand nombre de tests,
   - chercher à itérer le processus pour un grand nombre de coups.

Merci pour votre attention.

Annexe 1 - Caractéristiques

Les caractéristiques du billard :

- Longueur : 107.6 cm
- largeur : 53.7 cm
- largeur bandes : 4 cm
- largeur bords en bois : 1.3 cm
- Rayon grosse boule blanche : 179.7 g
- Rayon petite boule blanche : 139.9 g
- Rayon petite boule jaune : 139.1 g
- Rayon petite boule rouge : 131 g

Les caractéristiques du pendule :

- Longueur du baton : L = 28 cm
- Longueur de la gomme : l = 9.2 cm
- Rayon de la gomme : $R_m = 2.7$ cm
- Rayon du cylindre en bois : $R_b = 1.3$ cm

Annexe 2 - Le moment d'inertie du maillet

$$I(\{maillet\})_{O,(x,y,z)} = \begin{pmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{pmatrix}$$

$$B = \iiint_{V_{baton}} (x^2 + z^2)\, dm_{baton} + \iiint_{V_{gomme}} (x^2 + z^2)\, dm_{gomme}$$

$$B = \rho_{baton} \pi {R_b}^2 \times (\frac{L{R_b}^2}{4} + \frac{(L + R_m)^3 - {R_m}^3}{3}) + \frac{l \rho_{gomme} {R_m}^2}{4} \times ({R_m}^2 + \frac{l^2}{3})$$

Moment d'inertie du maillet

$$B = 4,8 \times 10^{-3} kg.m^2$$

Annexe 3 - Physique des collisions

Théorème du centre d'inertie : $\frac{dp(t)}{dt} = \sum F_{ext} = 0 \rightarrow p_i = p_f$

Premier principe de la thermodynamique : $E_{c,i} = E_{c,f}$

On obtient le système d'équation :

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

Annexe 3 - Physique des collisions

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

En élevant au carré la première équation, puis en soustrayant avec la deuxième, on obtient :

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

Résultat 1

Les vecteurs vitesse des deux boules à l'issue de la collision forment un angle droit : $\theta_1 - \theta_2 = \frac{\pi}{2}$

Annexe 3 - Physique des collisions

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

On multiplie la première équation par $\vec{v_{1,f}}$, et l'on obtient :

Résultat 2

$$\begin{cases} v_{1,f} = v_{1,i} \times cos(\theta_1) \\ v_{2,f} = -v_{1,i} \times sin(\theta_1) \end{cases}$$

Annexe 4 - Vérification du parallèlogramme

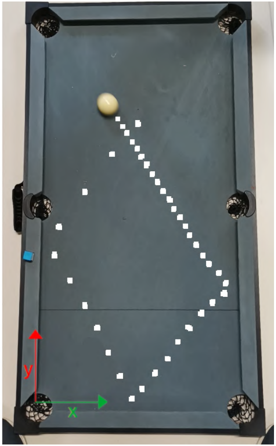


Figure 7.1 – Résultat n°1 parallèlogramme

Figure 7.2 – Résultat n°2 parallèlogramme

Références

J.PLOQUIN
Simulateur de billard réaliste
Faculté des Sciences, Université de Sherbrooke, Québec, Canada, 11 Octobre 2012, p.1-44

J-F.LANDRY
Planification optimale discrète et continue – un joueur de billard autonome optimisé
Mémoire PhD : Département informatique, Faculté des Sciences, Université de Sherbrooke, Québec, Canada, 19 Décembre 2012, p.16-19

M.LEMELIN, M.LEFRANCOIS
Optimisation du coup à jouer au jeu de billard
Université de Laval, Québec, 21 Avril 2015

Références

D.ALCIATORE
PhD : 90 degrees and 30 degrees Rule Follow-up – Part III : inelasticity and friction
http ://billiards.colostate.edu/bd articles/2005/april05.pdf

A.SINATRA, E.BRUNET
Cours de mécanique
Université Paris Sorbonne : 23 Janvier 2021, p.15-19

H.RESAL
Commentaire à la théorie mathématique du jeu de billard
Journal de mathématiques pures et appliquées, 3 ème série, tome 9 (1883), p. 65-98

O.EZRATTI
Les usages de l'intelligence artificielle
Edition Novembre 2018, p.31-123

Références

Physics Of Billiards
real-world-physics-problems.com