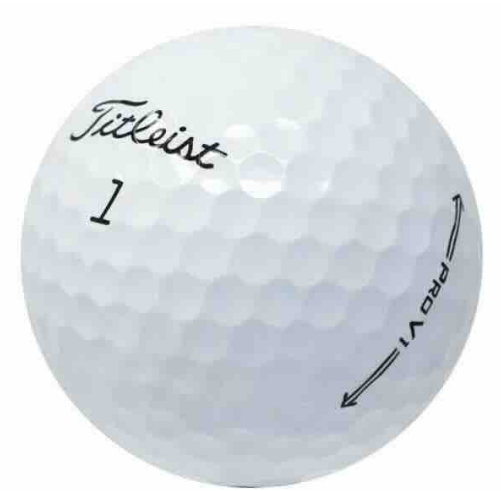
Aérodynamisme d'une balle de golf

Nils Xhoffray - 14665



Observations

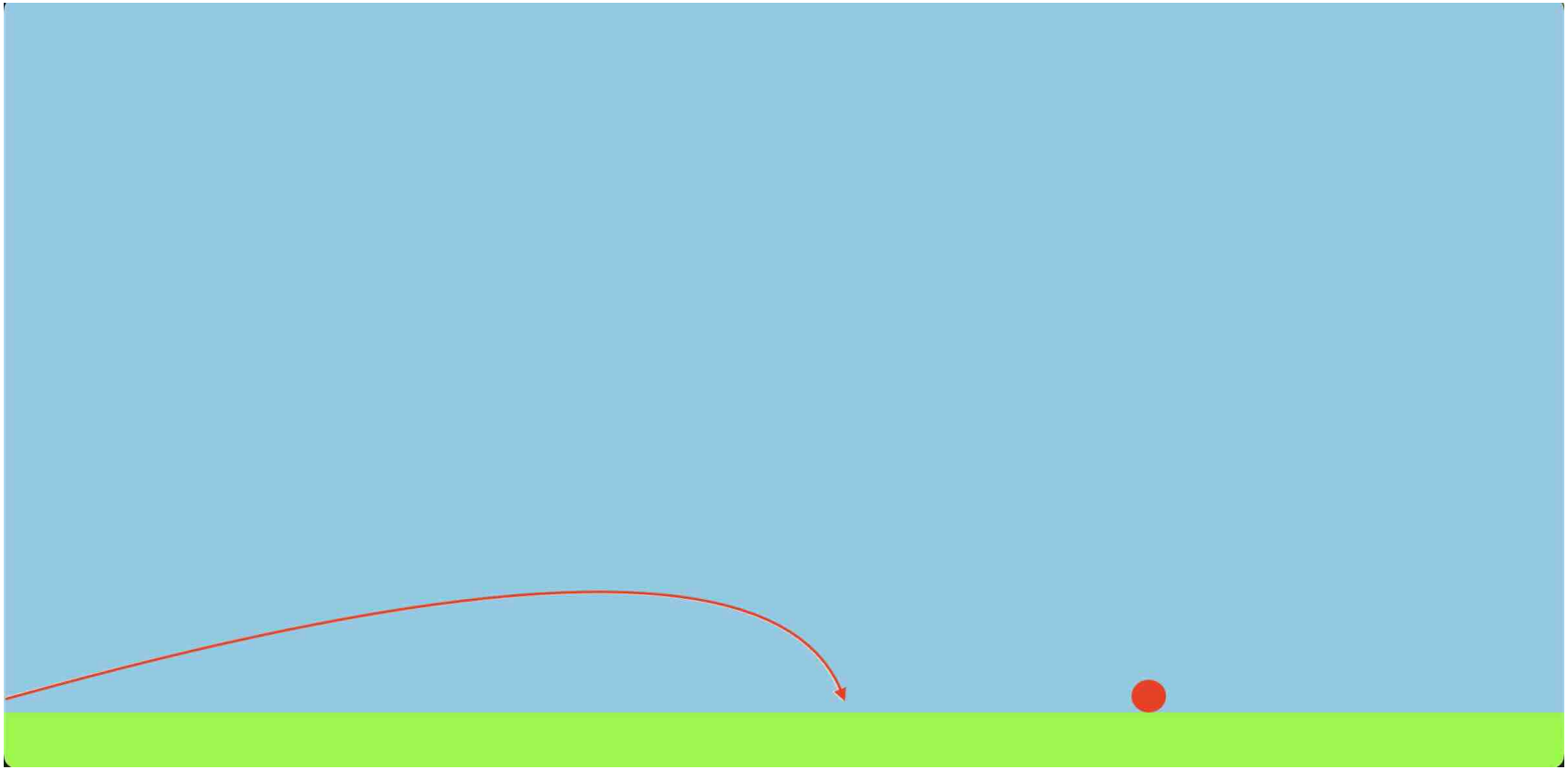


Figure – Tir simple

Observations

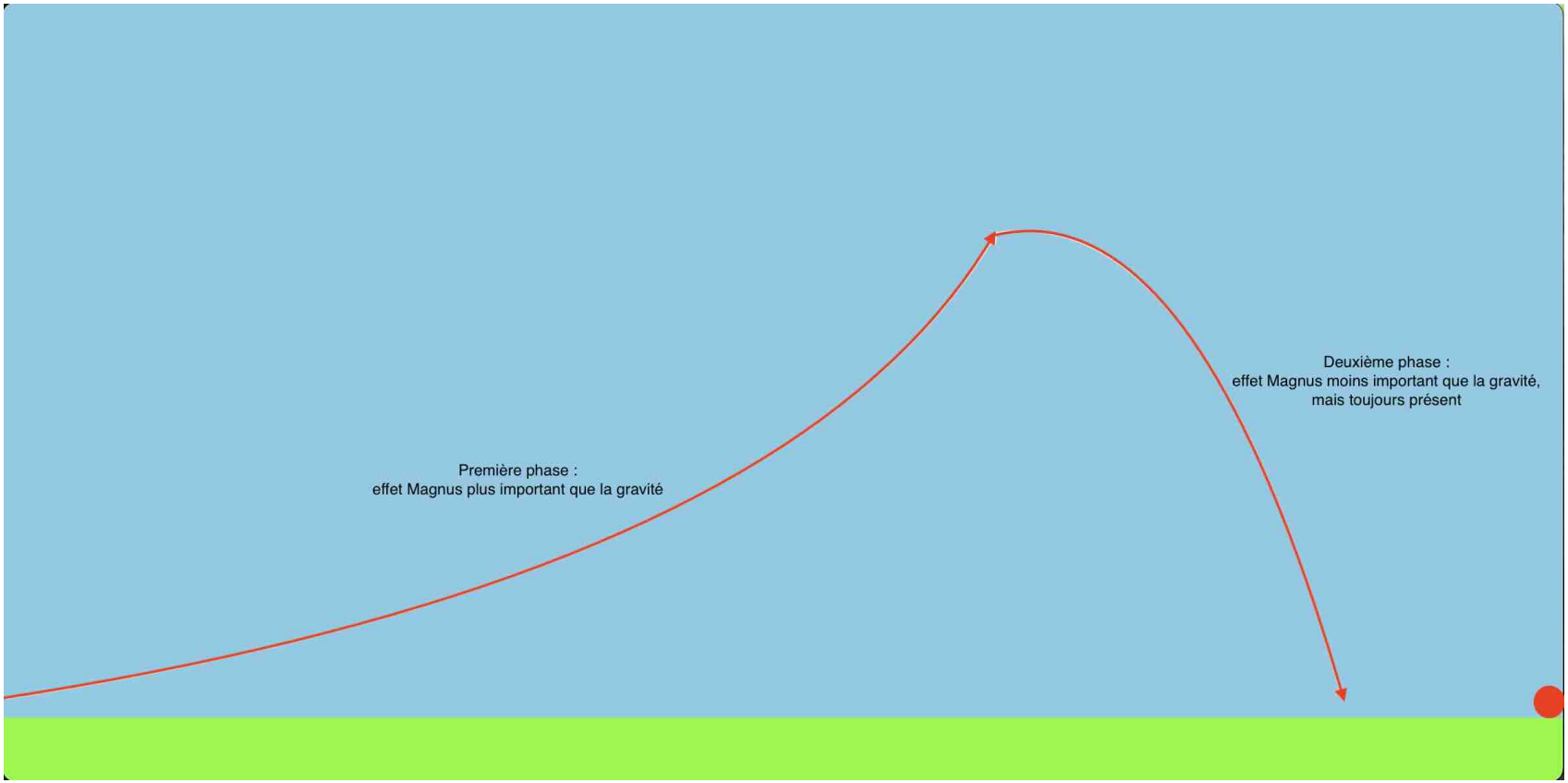


Figure – Tir d'une balle de golf

Problématique

Quels sont les impacts des alvéoles sur l'aérodynamisme d'une balle de golf ?

Nombre de Reynolds et régime de flux

Nombres de Reynolds unidimensionnel R_e

$$\boldsymbol{R_e} := \frac{\rho_f L_{car} v_{rel}}{\eta}$$

- $\boldsymbol{\rho_f}$: Masse volumique du fluide **ici** $\boldsymbol{\rho_{air}} = 1.225 \boldsymbol{kg/m^3}$
- $\boldsymbol{L_{car}}$: Longueur caractéristique (diamètre, etc...) **ici diamètre** $\boldsymbol{d} = 0.04\boldsymbol{m}$
- $\boldsymbol{v_{rel}}$: Vitesse du fluide par rapport à l'objet **ici** $\boldsymbol{v} = 30\boldsymbol{m/s}$
- η : Viscosité dynamique du fluide **ici** $\eta = 1.81 \cdot 10^5 \boldsymbol{Pa \cdot s}$.

Ici on obtient $\boldsymbol{R_e} = 6.8 \cdot 10^5$

Différents régimes

- **Flux laminaire** : $R_e < 2000$
- **Flux de transition** : $2000 < R_e < 4000$
- **Flux turbulent** : $R_e > 4000$

Ecoulement et Trainée aérodynamique

Coefficient de trainée aérodynamique C_x

$$\text{Pour } \boldsymbol{R_e} > 10^3 : \vec{\boldsymbol{F_t}} := -\frac{1}{2} C_x S \rho_f v_{rel}^2$$

- $\boldsymbol{C_x}$: Coefficient de traînée aérodynamique
- **S** : Surface de référence **ici demi-sphère de surface** πr^2
- ρ_f : Masse volumique du fluide
- $\boldsymbol{v_{rel}}$: Vitesse relative entre l'objet et le fluide

Coefficient de portance C_L

$$\vec{\boldsymbol{L}} := \frac{1}{2} S C_L \rho_f v_{rel}^2$$

- **S** : Surface de référence
- $\boldsymbol{C_L}$: Coefficient de portance

Théorème de Bernoulli

Théorème de Bernoulli

Pour un fluide incompressible, non visqueux et en écoulement, l'énergie se conserve sur les lignes de courant

Expression mathématique

$$P + \frac{1}{2}\rho_f v^2 + \rho_f g h = \textbf{\textit{cste}}$$

- $\boldsymbol{P}$: Pression statique
- $\boldsymbol{\rho_f}$: Masse volumique du fluide
- $\boldsymbol{h}$: Hauteur par rapport à une référence

Effet Magnus

Effet Magnus

Selon le principe de Bernoulli, pour un objet en rotation, l'augmentation de la vitesse du fluide autour de celui-ci entraîne une diminution de la pression du côté où la vitesse du fluide est plus élevée.

Expression mathématique

$$\vec{F_m} := \frac{1}{2} C_L S \rho_f (\vec{v} \cdot \vec{\omega})$$

- $\vec{\omega}$: vecteur rotation

Effet Magnus

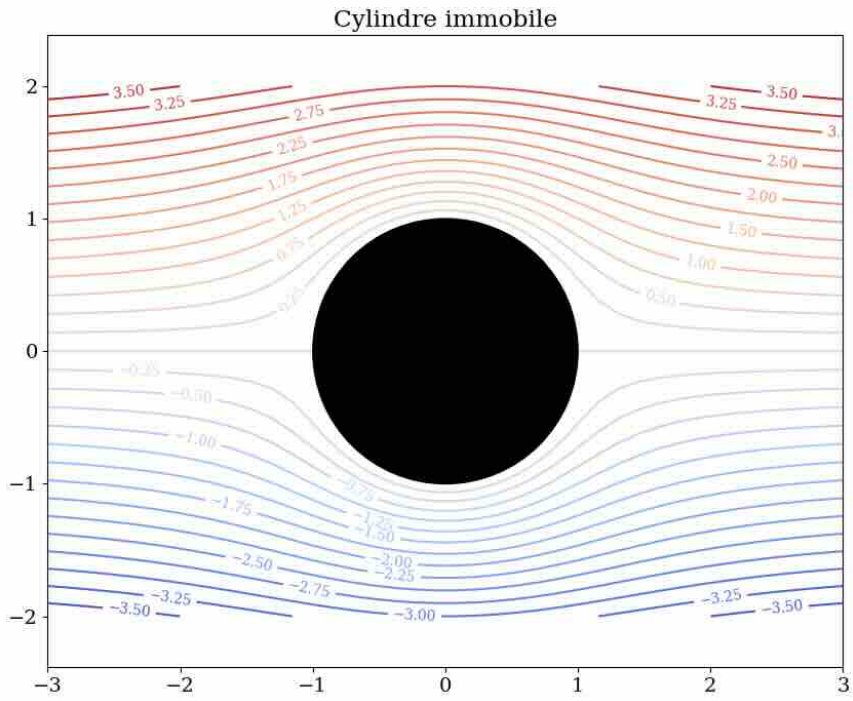


Figure – balle statique

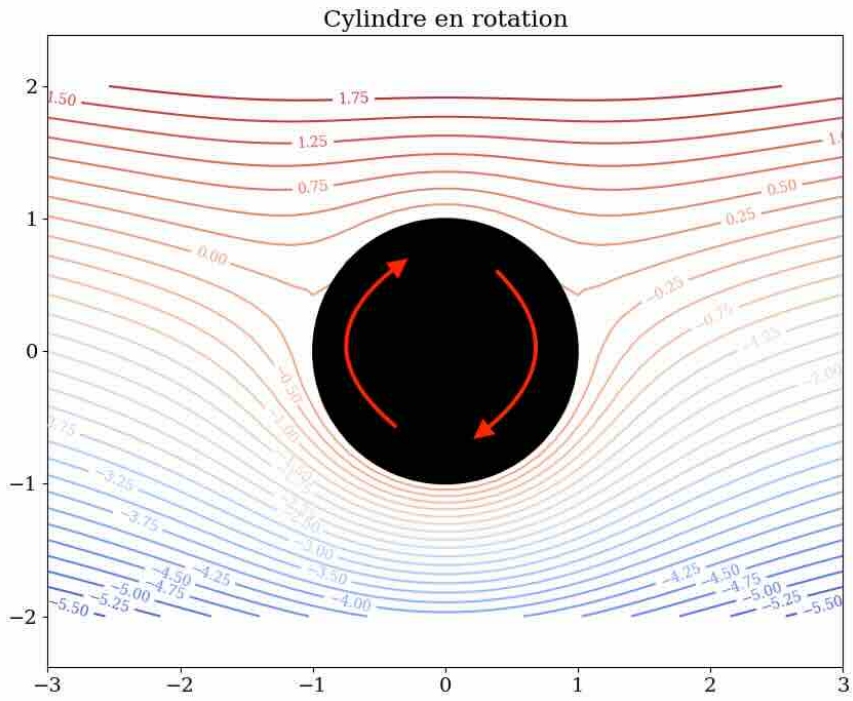


Figure – balle en rotation

Effet Magnus

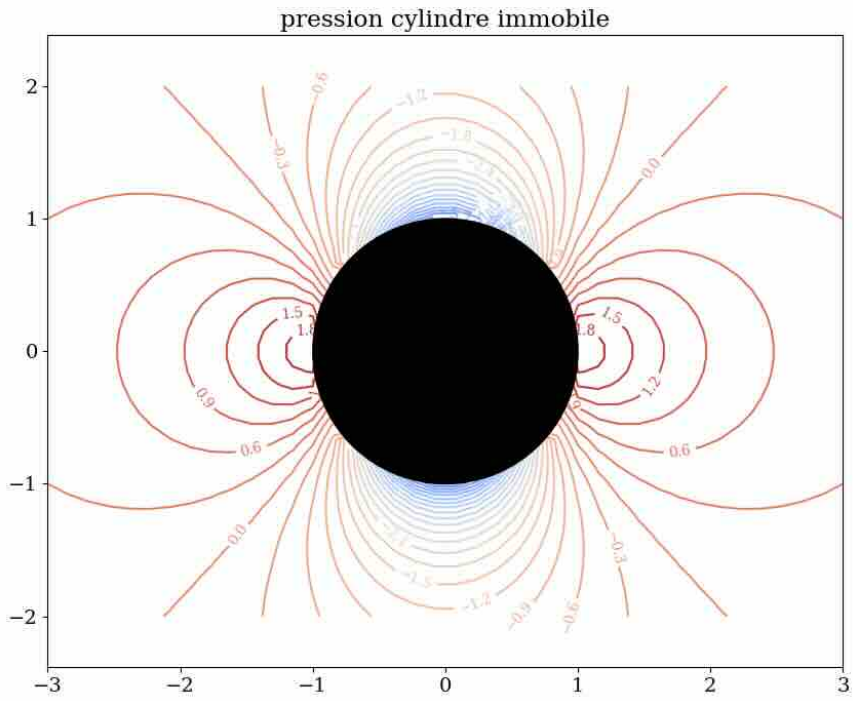


Figure – balle statique

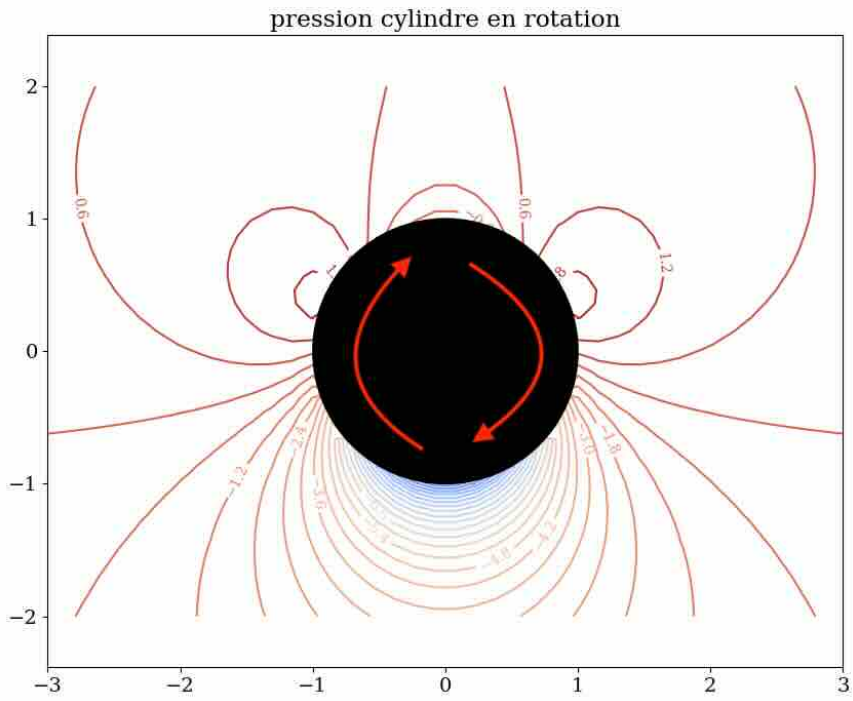


Figure – balle rotation

Curl

Définition du Curl

Le **curl** d'un champ vectoriel $\vec{\boldsymbol{F}}$, noté $\boldsymbol{\nabla} \times \vec{\boldsymbol{F}}$, est un vecteur qui mesure la rotation ou la circulation du champ en chaque point.

Expression mathématique en coordonnées cartésiennes

$$\boldsymbol{\nabla} \times \vec{\boldsymbol{F}} = \begin{vmatrix} \hat{\boldsymbol{i}} & \hat{\boldsymbol{j}} & \hat{\boldsymbol{k}} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ F_x & F_y & F_z \end{vmatrix} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \hat{\boldsymbol{i}} - \left(\frac{\partial F_z}{\partial x} - \frac{\partial F_x}{\partial z} \right) \hat{\boldsymbol{j}} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \hat{\boldsymbol{k}}$$

- $\vec{\boldsymbol{F}} = F_x \hat{\boldsymbol{i}} + F_y \hat{\boldsymbol{j}} + F_z \hat{\boldsymbol{k}}$: Champ vectoriel
- $\hat{\boldsymbol{i}}$, $\hat{\boldsymbol{j}}$, $\hat{\boldsymbol{k}}$: Vecteurs unitaires dans les directions x, y et z
- $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, $\frac{\partial}{\partial z}$: Dérivées partielles par rapport à x, y et z

Un peu d'histoire

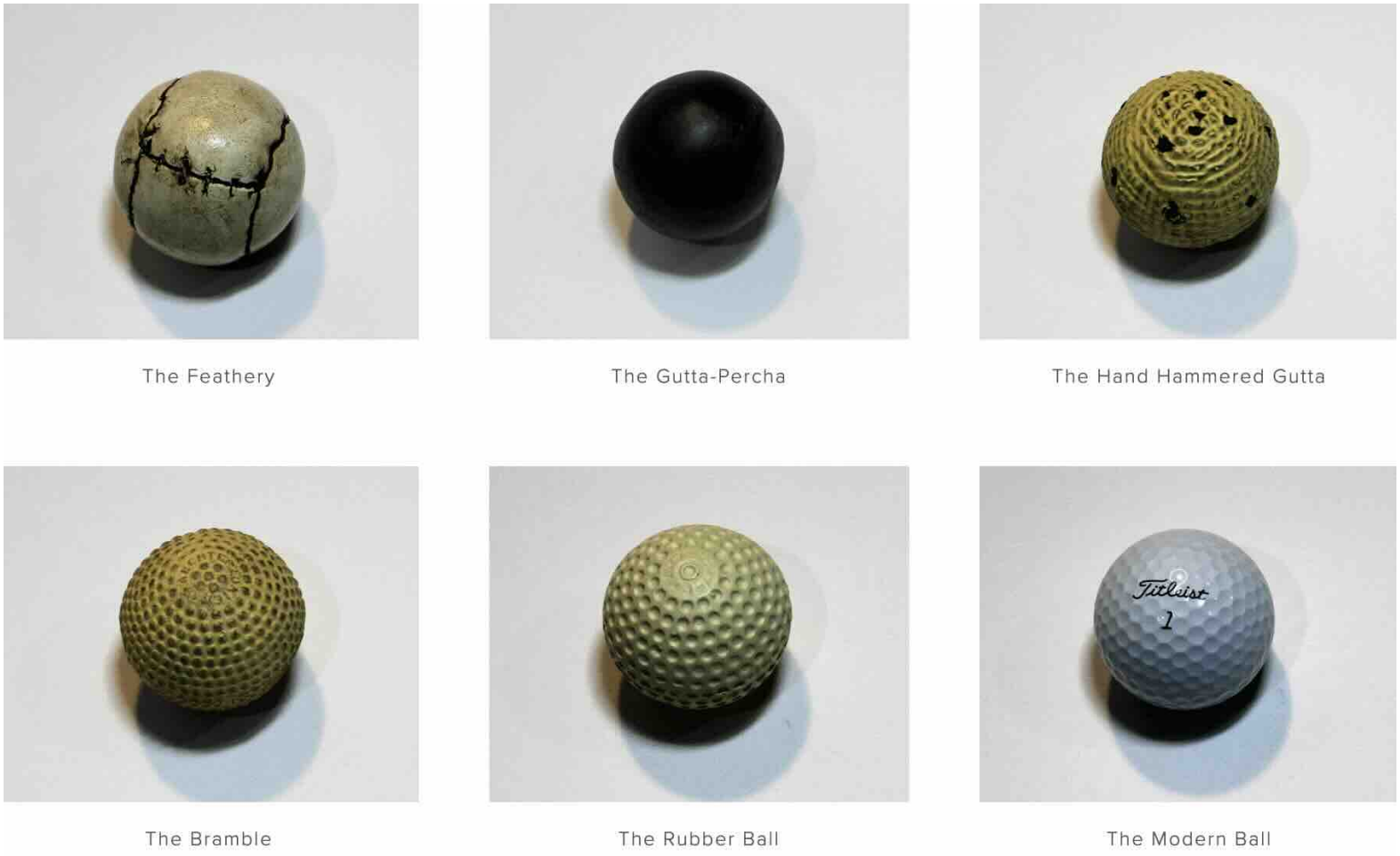


Figure – Évolution de la balle de golf

Hypothèses des effets possibles des alvéoles

- **Perturbation de l'écoulement**
- **Augmentation de la portance**
- **Réduction de la trainée**

Visualisation du curl au sein des alvéoles

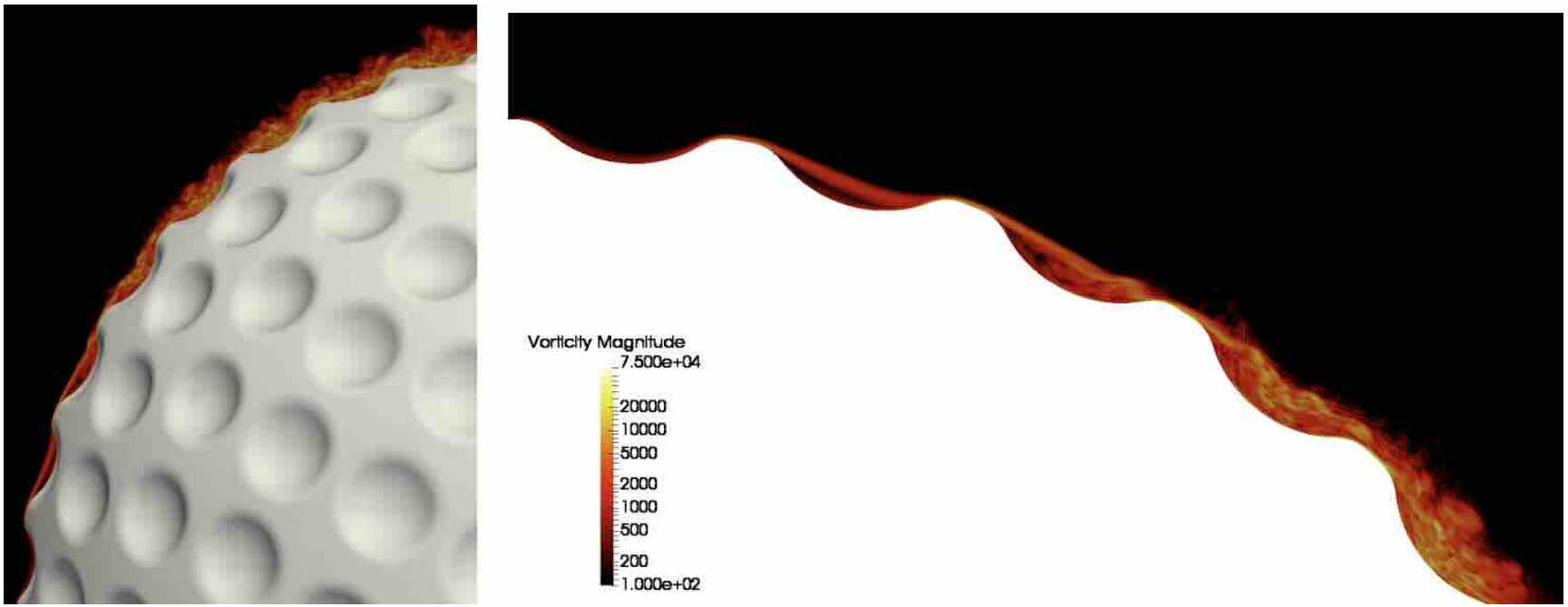


Figure – intensité de la vorticité

Visualisation du curl autour de la balle

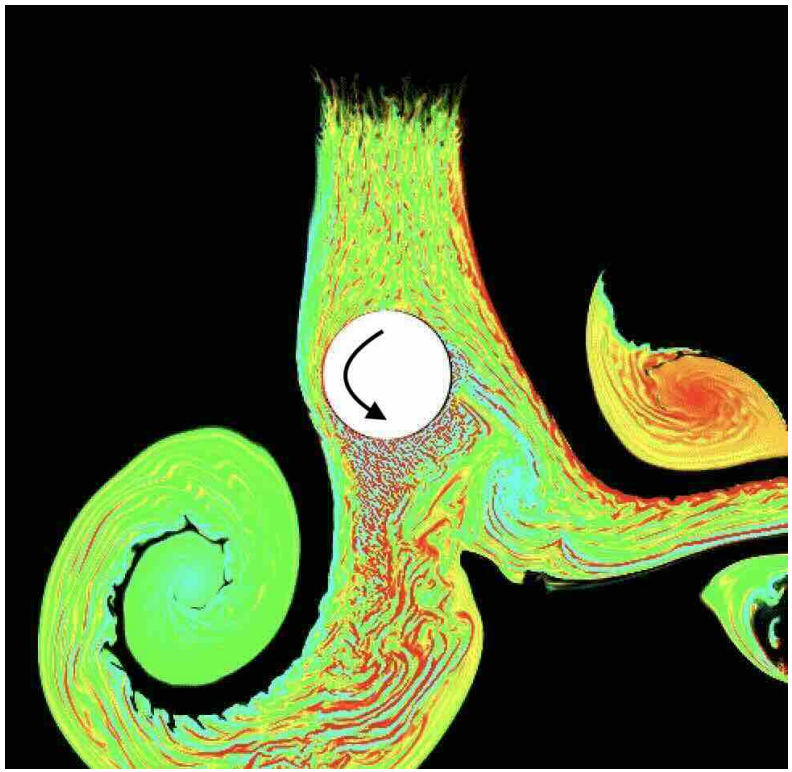


Figure – balle lisse

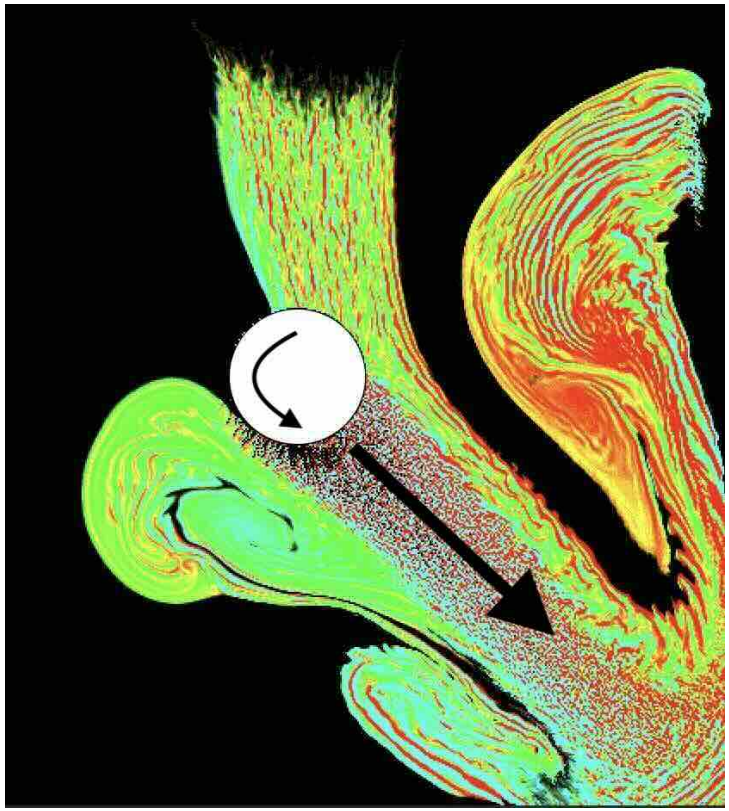


Figure – balle alvéolée

Pression autour de la balle

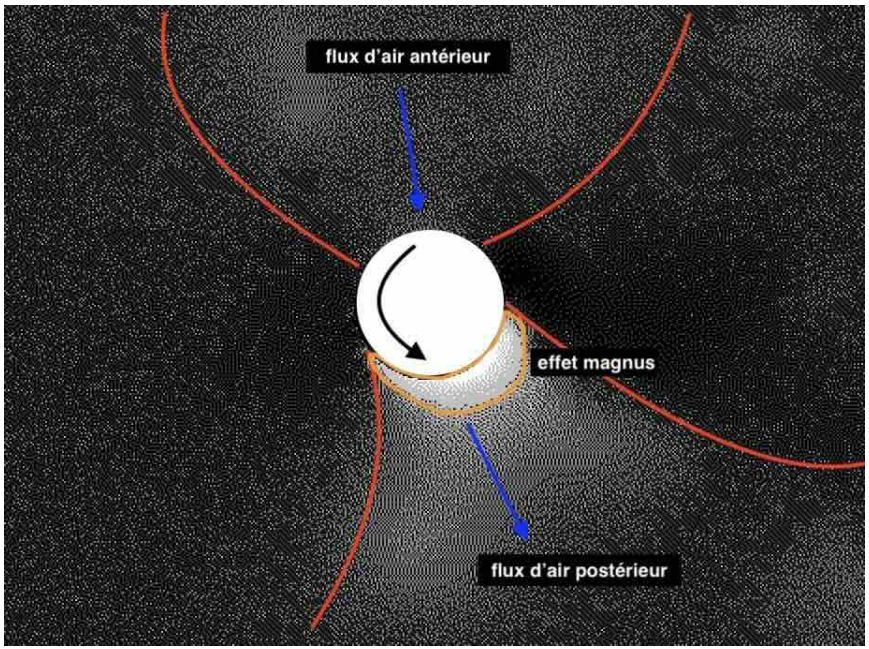


Figure – balle lisse

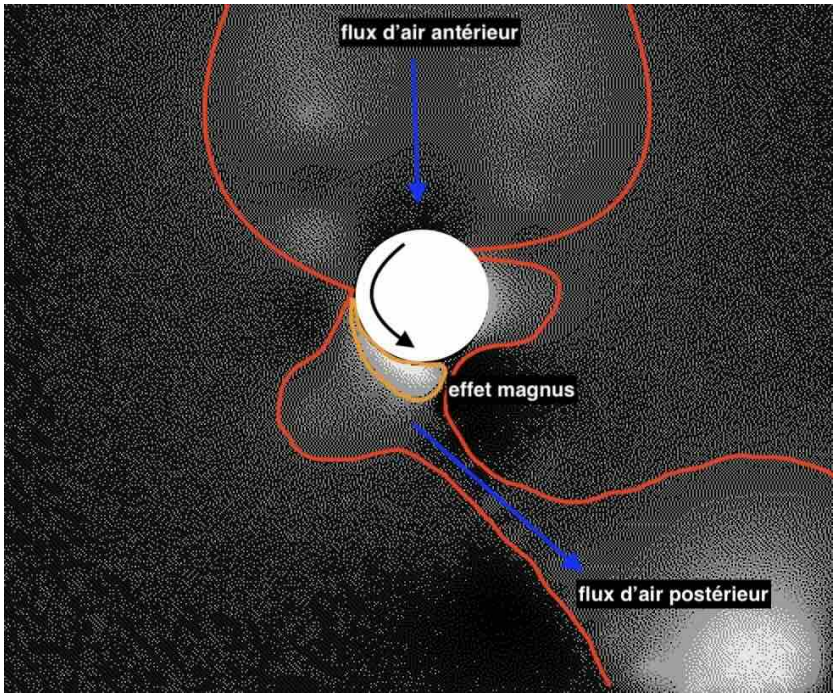


Figure – balle alvéolée

Augmentation de la portance : expérience

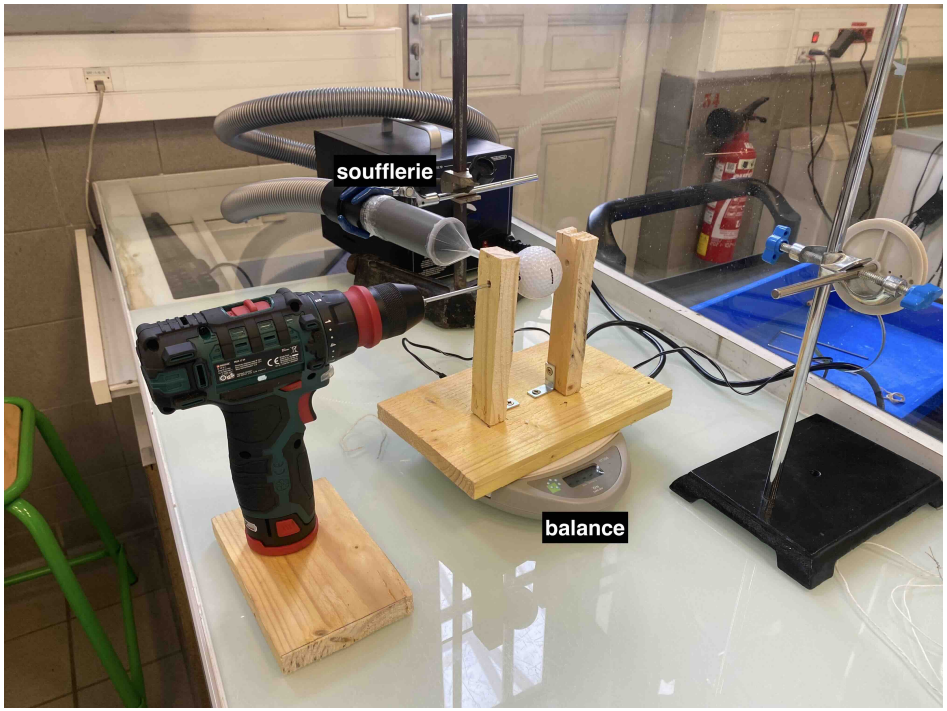


Figure – Montage expérimental

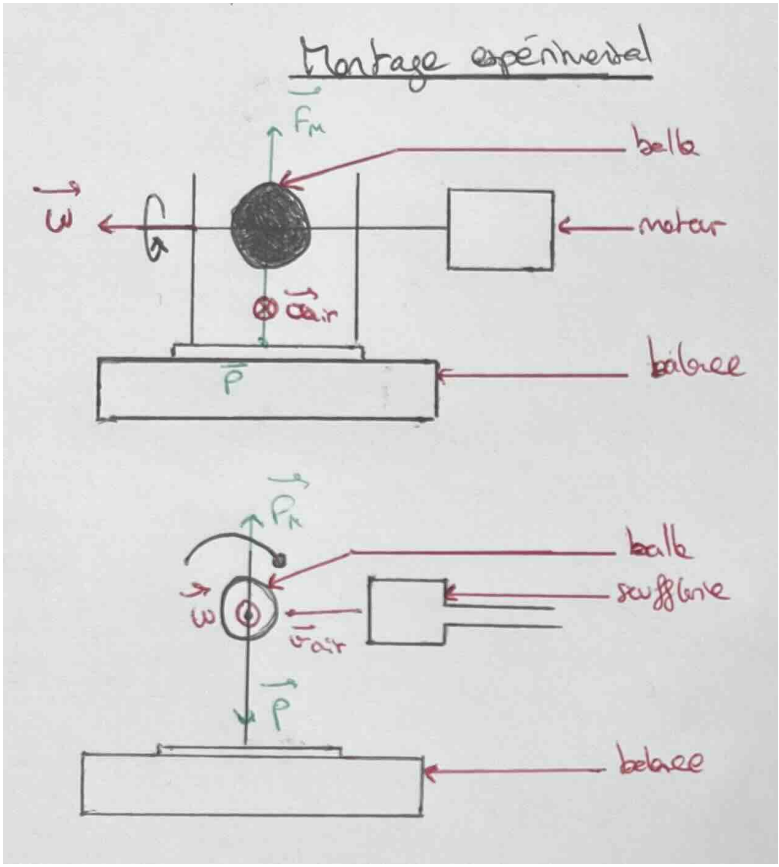


Figure – Schéma du montage

Augmentation de la portance : exploitation des données et limites

- $\|\vec{\omega}\| \approx 100\,\mathbf{rad/s}$
- **vitesse relative de l'air** $\boldsymbol{v} \approx 30\ \mathbf{m/s}$

Effet magnus théorique : $\|\vec{F_m}\| = \mathbf{20g}$

Réduction de la trainée : expérience

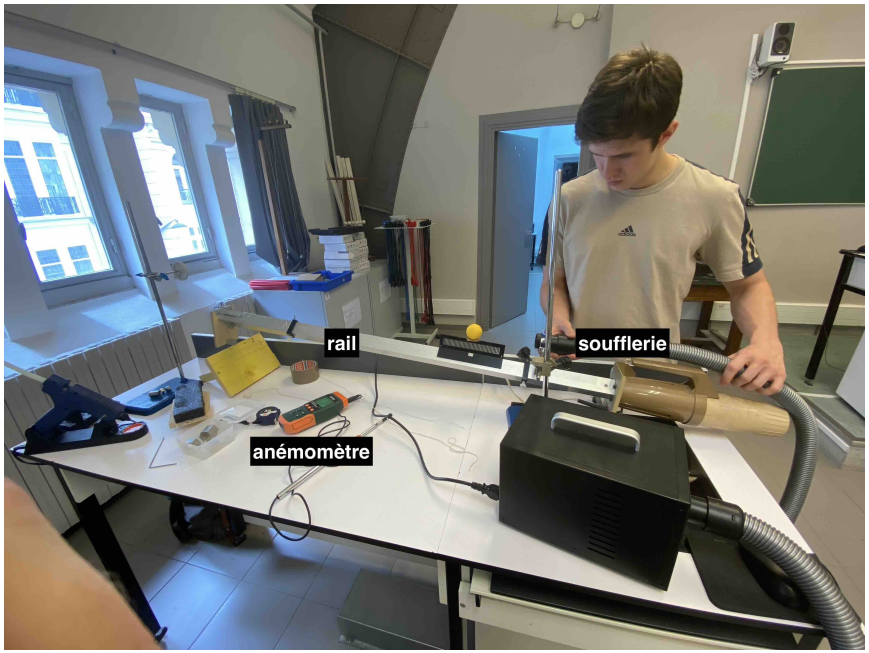


Figure – Mesure du C_x : Montage expérimental

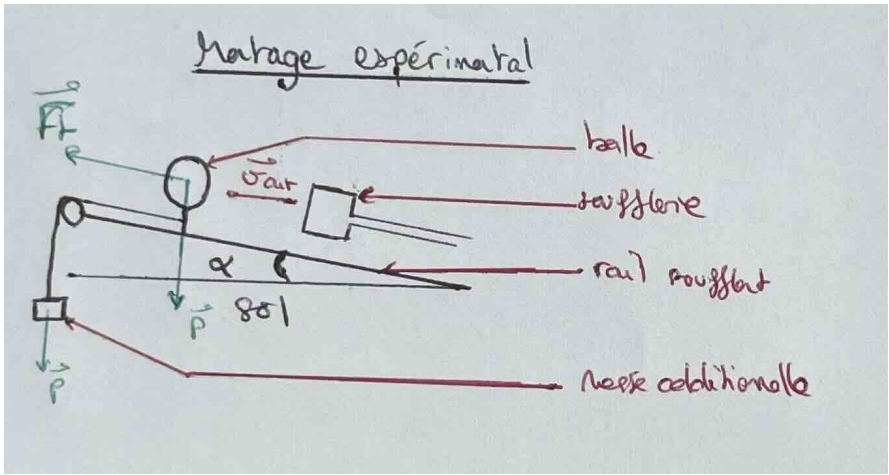


Figure – Schéma du montage

Réduction de la trainée : exploitation des données et limites

Balle alvéolée :

Angles (degré)	C_x
8	0,12
6	0,14
4	0,17
2	0,28

Moyenne du C_x : 0,18
Attendu : 0.25

Balle lisse :

Angles (degré)	C_x
8	0,38
6	0,15
4	0,21
2	0,28

Moyenne du C_x : 0,26
Attendu : 0.4

Dispositif du Trackman

Impact sur la portée et la stabilité : Trackman exploitation des données

Pro V1

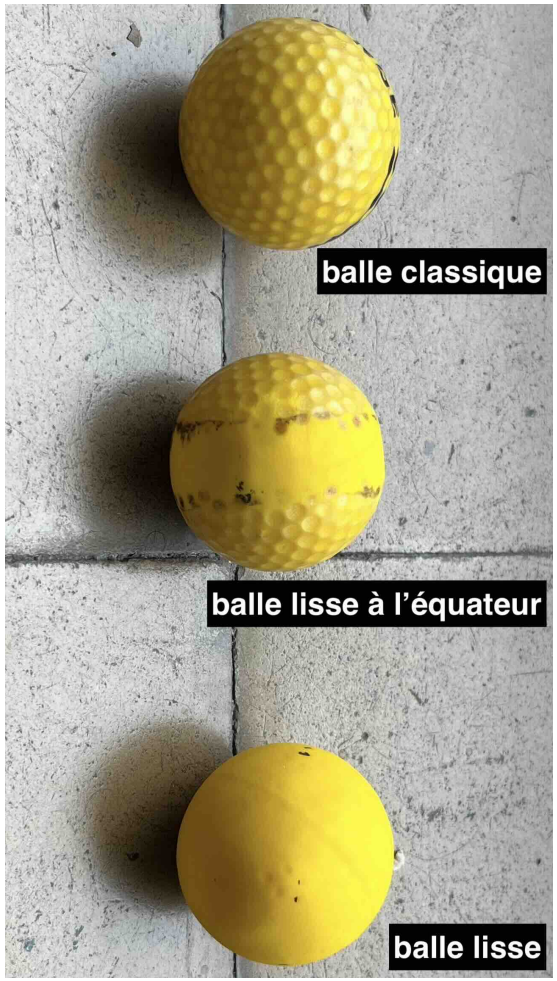
Metric	Vitesse balle (m, Mph)	Vitesse club (m, Mph)	Distance premier rebond (m)	Distance totale (m)	Vitesse rotation balle (Rpm)
Average	118.4	95.4	151.5	162.8	5558
Consistency (Variance)	4.6	0.8	7.3	9.3	707

Lisse à l'équateur

Metric	Ball Speed (m, Mph)	Club Speed (m, Mph)	Carry (m)	Total (m)	Spin Rate (Rpm)
Average	116.0	96.3	143.2	156.5	5928
Consistency (Variance)	5.5	1.8	9.0	11.6	1171

Lisse

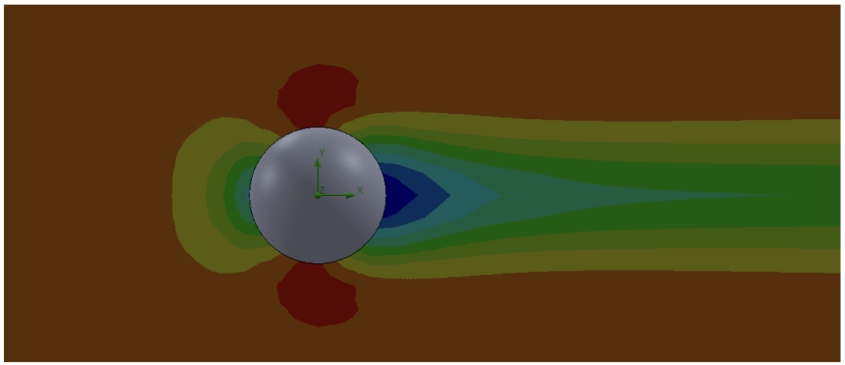
Metric	Ball Speed (m, Mph)	Club Speed (m, Mph)	Carry (m)	Total (m)	Spin Rate (Rpm)
Average	117.1	95.4	148.7	172.9	3601
Consistency (Variance)	2.4	1.1	4.8	3.5	469



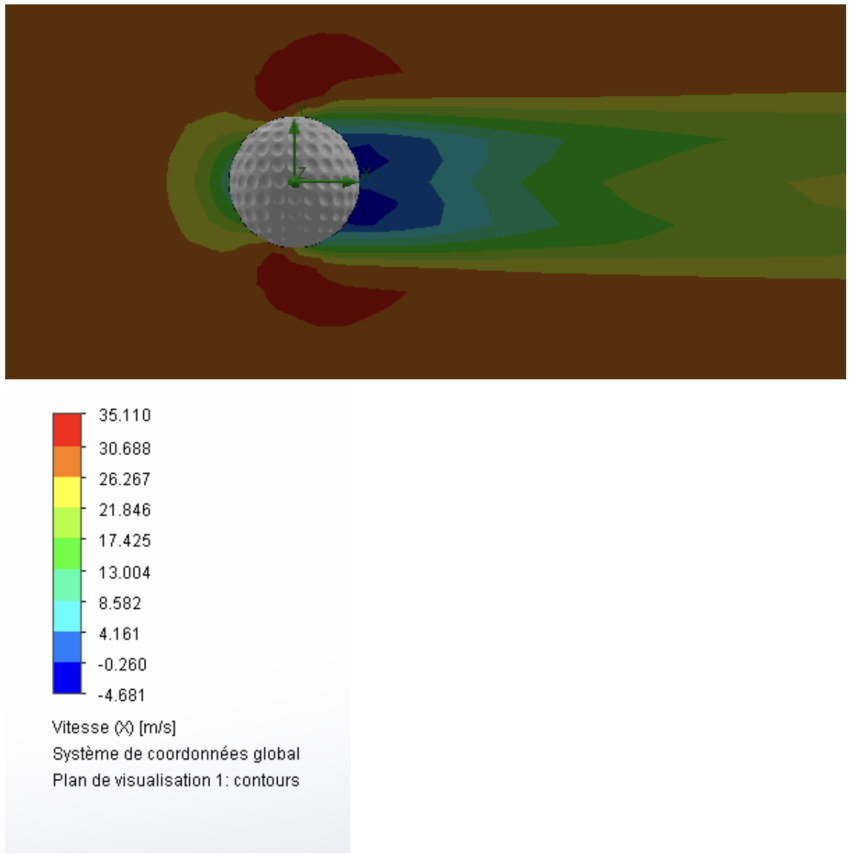
Conclusion

Annexe solidworks

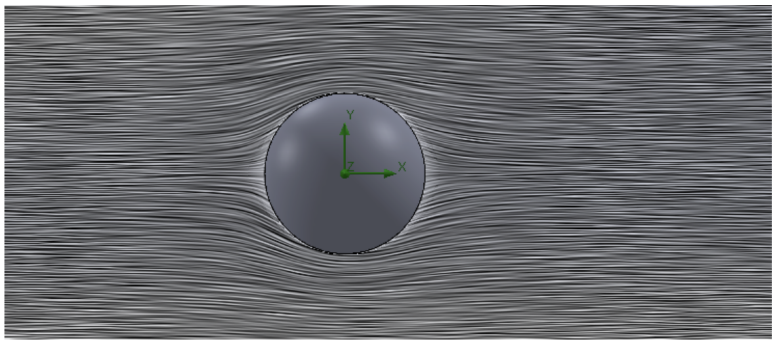
Balle lisse en vitesse

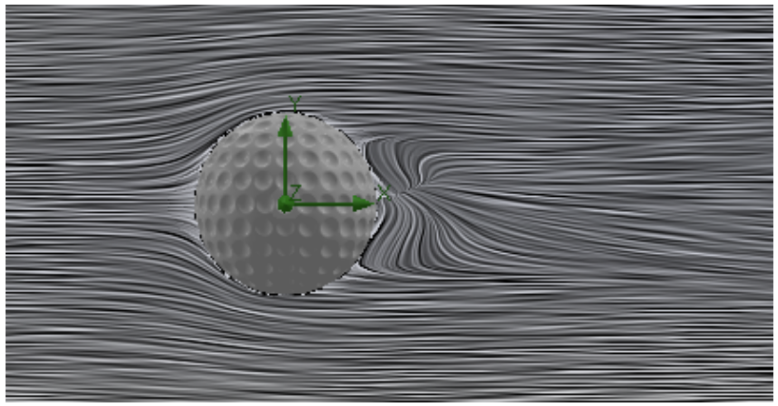


Balle alvéolée vitesse



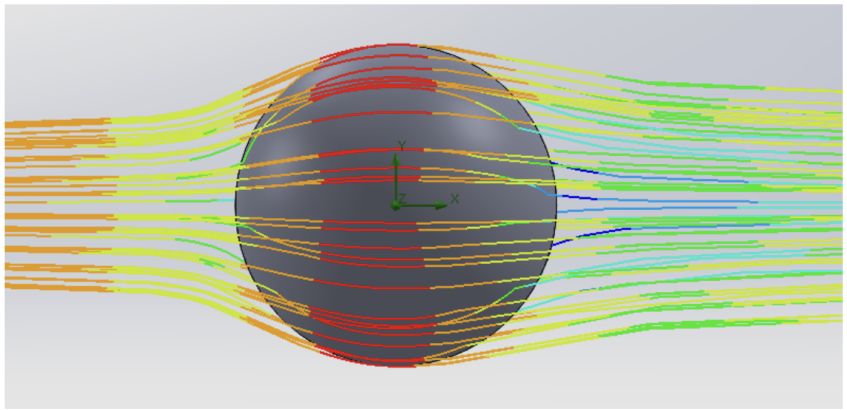
Annexe solidworks

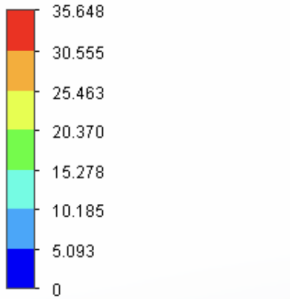


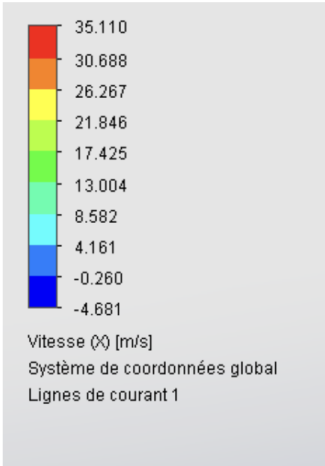


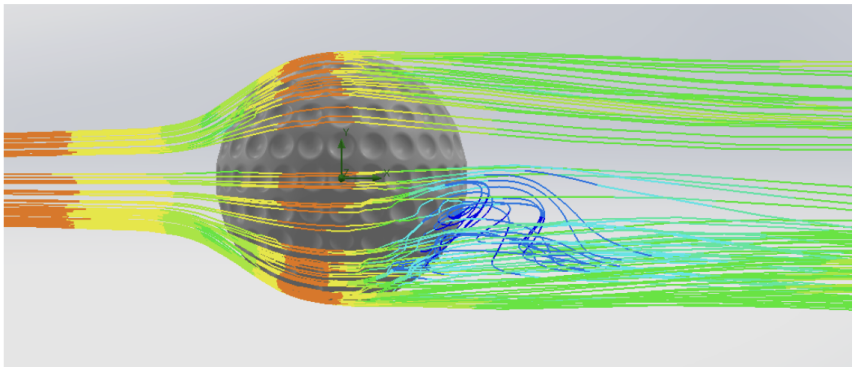
Annexe solidworks

Balle lisse vitesse









simulation masse

```
1   import pygame
2   import numpy as np
3   from colorama import Fore, Style
4   import random as rd
5   import time
6
7   aaa = True
8   temps_rota = 1
9   premier_contact = True
10  carry = 0
11
12  def print_red(text):
13      print("\033[91m" + text + "\033[0m")
14  def print_blue(text):
15      print("\033[94m" + text + "\033[0m")
16  def print_green(text):
17      print("\033[32m" + text + "\033[0m")
18
19  def max_dico(D):
20      m = 0
21      c = 0
22      for e in D:
23          if D[e] > m:
24              m = D[e]
25              c = e
26      return (m, c)
27
```

simulation masse

```
1     def min_dico(D):
2         m = float('inf')
3         c = None
4         for cle, valeur in D.items():
5             if valeur < m:
6                 m = valeur
7                 c = cle
8         return (m, c)
9
10    def print_dico(D):
11        m, c = max_dico(D)
12        for e in D:
13            if e == c:
14                texte = f"{np.degrees(e)}: {D[e]}"
15                texte_colore = f"{Fore.RED}{texte}{Style.RESET_ALL}"
16                print(texte_colore)
17            else:
18                texte = f"{np.degrees(e)}: {D[e]}"
19                print(texte)
20
```

simulation masse

```
1     class Masse:
2         def __init__(self, masse, rayon, pos, vit, acc, ang, couleur):
3             self.x, self.y = pos
4             vx, vy = vit
5             self.vit_x, self.vit_y = vx * np.cos(ang), vy * np.sin(ang)
6             self.acc_x, self.acc_y = acc, acc
7             self.masse = masse
8             self.rayon = rayon
9             self.couleur = couleur
10            self.ang = ang
11            self.spin = True
12
13        def rota(self):
14            global aaa
15            global temps_rota
16
17            if self.spin:
18                if aaa:
19                    self.temps_debut = time.time()
20                    aaa = False
21
22                temps_ecoule = time.time() - self.temps_debut
23                if temps_ecoule >= temps_rota:
24                    self.spin = False
25
```

simulation masse

```
1      class Sol:
2          def __init__(self, altitude, hauteur, largeur, absor):
3              self.altitude = altitude
4              self.hauteur = hauteur
5              self.largeur = largeur
6              self.absor = absor
7
```

simulation masse

```
    class main:
        def __init__(self, width, height, Sol, grav, frot, frot_s):
            self.width, self.height = width, height
            self.batis_x, self.batis_y = self.width // 2, 100
            self.Masse = []
            self.Sol = Sol
            self.grav = grav
            self.frot = frot
            self.frot_s = frot_s

        def init_pygame(self):
            pygame.init()
            self.screen = pygame.display.set_mode((self.width, self.height))
            self.clock = pygame.time.Clock()

        def handle_events(self):
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    quit()

```

simulation masse

```
        def ajouter_masse(self, masse):
            self.Masse.append(masse)

        def update(self, masse):
            global premier_contact
            global carry
            temps_actuel = pygame.time.get_ticks()
            temps_ecoule_secondes = temps_actuel / 1000.0

            masse.rota()

```

simulation masse

```
1            if masse.y == self.Sol.altitude - self.Sol.hauteur//2 - masse.rayon
     + 1 and masse.x >= 300:
2                masse.acc_x = (-self.frot * masse.vit_x - self.frot_s * masse.
     vit_x) / masse.masse
3                masse.vit_x += masse.acc_x
4                masse.x += masse.vit_x
5                masse.acc_y = (-self.frot * masse.vit_y) / masse.masse  + self.
     grav
6                masse.vit_y += masse.acc_y
7                masse.y += masse.vit_y
8            else:
9                masse.acc_x = -self.frot * masse.vit_x / masse.masse
10               masse.vit_x += masse.acc_x
11               masse.x += masse.vit_x
12               if masse.spin:
13                   masse.acc_y = -self.frot * masse.vit_y / masse.masse + self.
     grav - 1.1 ** temps_ecoule_secondes
14               else:
15                   masse.acc_y = -self.frot * masse.vit_y / masse.masse + self.
     grav - 0.6
16               masse.vit_y += masse.acc_y
17               masse.y += masse.vit_y
18
19           if (masse.y + masse.rayon) >= self.Sol.altitude - self.Sol.hauteur
     //2 and not masse.x >= 300:
20               masse.vit_y *= -1
21               masse.y = self.Sol.altitude - self.Sol.hauteur//2 - masse.rayon
     + 1
22
```

simulation masse

```
            if (masse.y + masse.rayon) >= self.Sol.altitude - self.Sol.hauteur
        //2 and masse.x >= 300:
                masse.vit_y *= -self.Sol.absor
                masse.y = self.Sol.altitude - self.Sol.hauteur//2 - masse.rayon
        + 1
                if premier_contact:
                    carry = int(masse.x)
                    premier_contact = False

            return int(masse.x), int(masse.y), int(masse.vit_x), int(masse.vit_y
        )

        def run(self):
            global angle
            global carry
            self.init_pygame()
            termine = []
            en_cours = [self.Masse[0]]
            i = 0
            bool = True
            D = {}
            D_y = []

            try:
                while True:
                    self.handle_events()
                    self.screen.fill((135, 206, 235))
                    pygame.draw.rect(self.screen, (124, 252, 0), (0, self.Sol.
        altitude, self.width, self.height - self.Sol.altitude))

```

simulation masse

```
                    for masse in en_cours:
                        x, y, vx, vy = self.update(masse)
                        D_y.append(y)
                        pygame.draw.circle(self.screen, masse.couleur, (x, y),
    masse.rayon)
                        if vx == 0 and vy == 0:
                            termine.append(self.Masse[i])
                            en_cours = []
                            i += 1
                            if i < len(self.Masse):
                                en_cours.append(self.Masse[i])
```

simulation masse

```
                    for masse in termine:
                        x, y, vx, vy = self.update(masse)
                        pygame.draw.circle(self.screen, masse.couleur, (x, y),
    masse.rayon)
                        if masse.ang not in D:
                            D[angle] = x

                    pygame.display.flip()
                    self.clock.tick(60)

                    if len(termine) == len(self.Masse) and bool == True:
                        m, c = max_dico(D)
                        p, q = min_dico(D)
                        print("\n")
                        print_red("angle maximal :" + str(np.degrees(c)))
                        print("\n")
                        print_blue("distance maximale :" + str(m))
                        print_blue("carry :" + str(carry))
                        print("\n")
                        print_green("altitude maximale :"+ str(max(D_y) - min(
    D_y)))
                        print("\n")
                        bool = False

```

simulation masse

```
            except Exception as e:
                print("Une erreur s'est produite:", str(e))
            finally:
                pygame.quit()



    width, height = 1200, 600

    alti = height - 50
    hauteur = 10
    largeur = width
    absor = 0.75

    masse = 10
    rayon = 10
    pos = (width // 2 - largeur // 2, alti - hauteur//2 - rayon + 1)
    vit = (15, 15)
    acc = 0

    grav = 1
    frot = 0.1
    frot_s = 0.25

    ran = 50

    angle = np.pi/18
    couleur = (255, 255, 255)

```

simulation masse

```
1      sol = Sol(alti, hauteur, largeur, 1 - absor)
2
3      simu = main(width, height, sol, grav, frot, frot_s)
4      simu.ajouter_masse(Masse(masse, rayon, pos, vit, acc, angle, couleur))
5
6      simu.run()
7
```

effet magnus

```
1   import numpy as np
2   import sympy as sp
3   import sympy.vector as sv
4   sp.init_printing()
5   import matplotlib.pyplot as plt
6   from matplotlib import rcParams
7   rcParams['font.family'] = 'serif'
8   rcParams['font.size'] = 14
9
10  R0 = sv.CoordSys3D('R_0')
11  x = R0.x
12  y = R0.y
13  U0, R, K, Gamma = sp.symbols('U_0 R K Gamma')
14
15  # 1/ psi1 correspond    un champ uniforme
16  psi1 = U0*y
17  # verification equation de laplace
18  sv.divergence(sv.gradient(psi1))
19
20  # 2/ psi2 correspond    un doublet
21  psi2 = -K*y/(x**2+y**2)
22  sv.gradient(psi2)
23
24  # verification equation de laplace
25  sv.divergence(sv.gradient(psi2)).simplify()
26
27  # 3/ psi3 correspond    un tourbillon
28  psi3 = -Gamma/(2*sp.pi)*sp.log(x**2+y**2)
29
```

effet magnus

```
    # verification equation de laplace
    sv.divergence(sv.gradient(psi3)).simplify()
    psi = psi1 + psi2.subs(K,U0*R**2) + psi3

    def trace_solution(psi,R,titre):
        # conversion fonction numpy
        F = sp.lambdify([R0.x,R0.y],psi,'numpy')
        # grille de calcul
        N = 40
        X1 = np.linspace(-3*R,3*R,N)
        Y1 = np.linspace(-2*R,2*R,N)
        X, Y = np.meshgrid(X1,Y1)
        FXY = F(X,Y)
        # filtrage
        for i in range(N):
            for j in range(N):
                if (X[i,j]**2 + Y[i,j]**2) < 0.8*R**2 : FXY[i,j] = 0.
        # tracer
        plt.figure(figsize=(10,8))
        ax = plt.gca()
        CS = ax.contour(X, Y, FXY, levels=31, cmap='coolwarm')
        ax.clabel(CS, inline=1, fontsize=10)
        plt.axis('equal')
        if titre != None : ax.set_title(titre)
        cercle = plt.Circle((0., 0.), R, color='k',zorder=10)
        ax.add_artist(cercle)
        return
    rayon=1

```

effet magnus

```
1     valnum = { U0:2, R:0, Gamma:0}
2     psi0 = psi.subs(valnum)
3
4     trace_solution(psi0,rayon,"Ecoulement uniforme")
5
6     valnum = { U0:2, R:rayon, Gamma:0}
7     psi0 = psi.subs(valnum)
8
9     trace_solution(psi0,rayon,"Cylindre immobile")
10
11    valnum = { U0:2, R:rayon, Gamma:5}
12    psi0 = psi.subs(valnum)
13
14    trace_solution(psi0,rayon,"Cylindre en rotation")
15
16    Nabla = sv.Del()
17    rho0 = sp.symbols('rho_0')
18    p = sp.Function('p')(x,y)
19    u = sp.Function('u')(x,y)
20    v = sp.Function('v')(x,y)
21    U = u*R0.i + v*R0.j
22
23    # calcul de la pression
24    pr = rho0*U0**2/2 -rho0/2*U.dot(U)
25    pr = rho0*U0**2/2-rho0/2*sv.gradient(psi).dot(sv.gradient(psi))
26
```

effet magnus

```
    valnum = { rho0:1, U0:2, R:rayon, Gamma:0}
    pr0 = pr.subs(valnum)
    trace_solution(pr0,rayon,"pression cylindre immobile")

    valnum = { rho0:1, U0:2, R:rayon, Gamma:5}
    pr0 = pr.subs(valnum)
    trace_solution(pr0,rayon,"pression cylindre en rotation")

```

numerical.py

```
1   from functools import reduce
2   from itertools import cycle
3   from math import factorial
4
5   import numpy as np
6   import scipy.sparse as sp
7
8
9   def difference(derivative, accuracy=1):
10      # Central differences implemented based on the article here:
11      # http://web.media.mit.edu/~crtaylor/calculator.html
12      derivative += 1
13      radius = accuracy + derivative // 2 - 1
14      points = range(-radius, radius + 1)
15      coefficients = np.linalg.inv(np.vander(points))
16      return coefficients[-derivative] * factorial(derivative - 1), points
17
18
19  def operator(shape, *differences):
20      # Credit to Philip Zucker for figuring out
21      # that kronsum's argument order is reversed.
22      # Without that bit of wisdom I'd have lost it.
23      differences = zip(shape, cycle(differences))
24      factors = (sp.diags(*diff, shape=(dim,) * 2) for dim, diff in
     differences)
25      return reduce(lambda a, f: sp.kronsum(f, a, format='csc'), factors)
26
```

fluid.py

```
1     import numpy as np
2     from scipy.ndimage import map_coordinates, spline_filter
3     from scipy.sparse.linalg import factorized
4
5     from numerical import difference, operator
6
7     class Fluid:
8         def __init__(self, shape, *quantities, pressure_order=1, advect_order=3)
       :
9             self.shape = shape
10            self.dimensions = len(shape)
11
12            # Prototyping is simplified by dynamically creating advected
       quantities as needed.
13            self.quantities = quantities
14            for q in quantities:
15                setattr(self, q, np.zeros(shape))
16
17            self.indices = np.indices(shape)
18            self.velocity = np.zeros((self.dimensions, *shape))
19
20            laplacian = operator(shape, difference(2, pressure_order))
21            self.pressure_solver = factorized(laplacian)
22
23            self.advect_order = advect_order
24
```

fluid.py

```
1            # Ajouter une balle solide au centre du domaine
2            self.ball_radius = 40
3            ball_center = np.floor_divide(shape, 2)
4            self.ball_mask = np.linalg.norm(self.indices - ball_center[:, None,
     None], axis=0) <= self.ball_radius
5            self.ball_rotation_speed = 0.5  # Vitesse angulaire de rotation de
     la balle
6            # Alv oles
7            self.alveoles = np.zeros(shape, dtype=bool)
8            # D finir les positions des alv oles (exemple ici, au centre)
9            center = np.floor_divide(shape, 2)
10           alveole_radius = 8
11           self.alveoles[center[0] - alveole_radius:center[0] + alveole_radius,
12                         center[1] - alveole_radius:center[1] + alveole_radius]
      = True
13
14         def step(self, dt=1):
15           # Advection is computed backwards in time as described in Stable
     Fluids.
16           advection_map = self.indices - self.velocity
17           # SciPy's spline filter introduces checkerboard divergence.
18           # A linear blend of the filtered and unfiltered fields based
19           # on some value epsilon eliminates this error.
20           def advect(field, filter_epsilon=10e-2, mode='constant'):
21               filtered = spline_filter(field, order=self.advect_order, mode=
     mode)
22               field = filtered * (1 - filter_epsilon) + field * filter_epsilon
23               return map_coordinates(field, advection_map, prefilter=False,
     order=self.advect_order, mode=mode)
24
```

fluid.py

```
1            # Apply advection to each axis of the velocity field and each user-
     defined quantity.
2            for d in range(self.dimensions):
3                self.velocity[d] = advect(self.velocity[d])
4
5            for q in self.quantities:
6                setattr(self, q, advect(getattr(self, q)))
7
8            # Compute the jacobian at each point in the velocity field to
     extract curl and divergence.
9            jacobian_shape = (self.dimensions,) * 2
10           partials = tuple(np.gradient(d) for d in self.velocity)
11           jacobian = np.stack(partials).reshape(*jacobian_shape, *self.shape)
12
13           divergence = jacobian.trace()
14
15           # If this curl calculation is extended to 3D, the y-axis value must
     be negated.
16           # This corresponds to the coefficients of the levi-civita symbol in
     that dimension.
17           # Higher dimensions do not have a vector -> scalar, or vector ->
     vector,
18           # correspondence between velocity and curl due to differing
     isomorphisms
19           # between exterior powers in dimensions != 2 or 3 respectively.
20           curl_mask = np.triu(np.ones(jacobian_shape, dtype=bool), k=1)
21           curl = (jacobian[curl_mask] - jacobian[curl_mask.T]).squeeze()
22
```

fluid.py

```
1           # Apply the pressure correction to the fluid's velocity field.
2           pressure = self.pressure_solver(divergence.flatten()).reshape(self.
    shape)
3           self.velocity -= np.gradient(pressure)
4
5           # Mettre    jour la position de la balle en fonction de la vitesse
    angulaire
6           ball_center = np.array([self.shape[0] / 2, self.shape[1] / 2])
7           ball_center += self.ball_rotation_speed * np.array([np.cos(dt * self
    .ball_rotation_speed),
8                                                               np.sin(dt * self
    .ball_rotation_speed)])
9
10          # Mettre    jour le masque de la balle avec la nouvelle position et
    les alv oles
11          self.ball_mask = np.linalg.norm(self.indices - ball_center[:, None,
    None], axis=0) <= self.ball_radius
12          self.ball_mask |= self.alveoles  # Combinaison avec le masque des
    alv oles
13
```

fluid.py

```
1         # Interagir avec le fluide en d finissant la vitesse autour de la
      balle et des alv oles
2         solid_velocity = np.zeros_like(self.velocity)
3         ball_indices = np.nonzero(self.ball_mask)
4         solid_velocity[:, ball_indices[0], ball_indices[1]] = self.
      ball_rotation_speed * np.array([-ball_indices[1] + ball_center[1],
      ball_indices[0] - ball_center[0]])
5         self.velocity += solid_velocity
6
7         return divergence, curl, pressure
8
```

curl.py

```
1   import numpy as np
2   from PIL import Image
3   from scipy.special import erf
4
5   from fluid import Fluid  # Utiliser la nouvelle classe avec les alv oles
6
7   RESOLUTION = 500, 500
8   DURATION = 100
9
10  INFLOW_PADDING = 50
11  INFLOW_DURATION = 100
12  INFLOW_RADIUS = 100
13  INFLOW_VELOCITY = 2
14  INFLOW_COUNT = 1
15
16  print('Generating fluid solver, this may take some time.')
17  fluid = Fluid(RESOLUTION, 'dye', pressure_order=2, advect_order=2)
18
19  center = np.floor_divide(RESOLUTION, 2)
20  r = np.min(center) - INFLOW_PADDING
21
22  points = np.linspace(-np.pi, np.pi, INFLOW_COUNT, endpoint=False)
23  points = tuple(np.array((np.cos(p), np.sin(p))) for p in points)
24  normals = tuple(-p for p in points)
25  points = tuple(r * p + center for p in points)
26
```

curl.py

```
    inflow_velocity = np.zeros_like(fluid.velocity)
    inflow_dye = np.zeros(fluid.shape)
    for p, n in zip(points, normals):
        mask = np.linalg.norm(fluid.indices - p[:, None, None], axis=0) <=
     INFLOW_RADIUS
        inflow_velocity[:, mask] += n[:, None] * INFLOW_VELOCITY
        inflow_dye[mask] = 1

    frames = []
    for f in range(DURATION):
        print(f'Computing frame {f + 1} of {DURATION}.')
        if f <= INFLOW_DURATION:
            fluid.velocity += inflow_velocity
            fluid.dye += inflow_dye

        # Mettre    jour la position de la balle pour la faire tourner dans le
     sens anti-horaire
        dt = 2 * np.pi / DURATION
        fluid.step(dt)

        curl = fluid.step()[1]
        # Using the error function to make the contrast a bit higher.
        # Any other sigmoid function e.g. smoothstep would work.
        curl = (erf(curl * 2) + 1) / 4

```

curl.py

```
        color = np.dstack((curl, np.ones(fluid.shape), fluid.dye))
        color = (np.clip(color, 0, 1) * 255).astype('uint8')

        # Remplacer la couleur de la balle par du rouge
        ball_color = np.array([255, 0, 0], dtype='uint8')
        color[fluid.ball_mask] = ball_color

        frames.append(Image.fromarray(color, mode='HSV').convert('RGB'))

    print('Saving simulation result.')
    frames[0].save('example2.gif', save_all=True, append_images=frames[1:],
     duration=20, loop=0)

```

pression.py

```
1   import numpy as np
2   from PIL import Image
3   from scipy.special import erf
4
5   from fluid import Fluid
6
7   RESOLUTION = 500, 500
8   DURATION = 100
9
10  INFLOW_PADDING = 50
11  INFLOW_DURATION = 100
12  INFLOW_RADIUS = 100
13  INFLOW_VELOCITY = 2
14  INFLOW_COUNT = 1
15
16  print('Generating fluid solver, this may take some time.')
17  fluid = Fluid(RESOLUTION, 'dye', pressure_order=2, advect_order=2)
18
19  center = np.floor_divide(RESOLUTION, 2)
20  r = np.min(center) - INFLOW_PADDING
21
```

pression.py

```
    points = np.linspace(-np.pi, np.pi, INFLOW_COUNT, endpoint=False)
    points = tuple(np.array((np.cos(p), np.sin(p))) for p in points)
    normals = tuple(-p for p in points)
    points = tuple(r * p + center for p in points)

    inflow_velocity = np.zeros_like(fluid.velocity)
    inflow_dye = np.zeros(fluid.shape)
    for p, n in zip(points, normals):
        mask = np.linalg.norm(fluid.indices - p[:, None, None], axis=0) <=
     INFLOW_RADIUS
        inflow_velocity[:, mask] += n[:, None] * INFLOW_VELOCITY
        inflow_dye[mask] = 1

    frames = []
    for f in range(DURATION):
        print(f'Computing frame {f + 1} of {DURATION}.')
        if f <= INFLOW_DURATION:
            fluid.velocity += inflow_velocity
            fluid.dye += inflow_dye

        # Mettre    jour la position de la balle pour la faire tourner dans le
     sens anti-horaire
        dt = 2 * np.pi / DURATION
        fluid.step(dt)

        # Calculate pressure
        _, _, pressure = fluid.step()

```

pression.py

```
1         # Normalize pressure values to range [0, 1] for visualization
2         pressure = (pressure - np.min(pressure)) / (np.max(pressure) - np.min(
      pressure))
3         pressure = np.clip(pressure, 0, 1)
4
5         # Convert pressure values to colors (example: blue to red colormap)
6         pressure_color = np.zeros((*fluid.shape, 3), dtype='uint8')
7         pressure_color[:, :, 0] = (pressure * 255).astype('uint8')  # Red
      channel
8         pressure_color[:, :, 2] = ((1 - pressure) * 255).astype('uint8')  # Blue
       channel
9
10        # Combine pressure with other visualizations (curl and dye)
11        curl = fluid.step()[1]
12        curl = (erf(curl * 2) + 1) / 4
13        color = np.dstack((curl, np.ones(fluid.shape), fluid.dye))
14        color = (np.clip(color, 0, 1) * 255).astype('uint8')
15
```

pression.py

```
        # Apply pressure visualization on top of other visualizations
        color[:, :, :3] = pressure_color

        # Remplacer la couleur de la balle par du rouge
        ball_color = np.array([255, 0, 0], dtype='uint8')
        color[fluid.ball_mask] = ball_color

        frames.append(Image.fromarray(color, mode='HSV').convert('RGB'))

    print('Saving simulation result.')
    frames[0].save('example2.gif', save_all=True, append_images=frames[1:],
     duration=20, loop=0)

```