École Normale Supérieure Paris Saclay

SAPH211 : Identification paramétrique et optimisation

---

# Tir parabolique
# Où est le canon Caesar ?

---

Paul BOULLIER
Jules FARNAULT

2023

# Sommaire

# Introduction

L'artillerie est un ensemble d'équipements militaires qui ont pour objectif d'attaquer à distances des cibles choisies. Ainsi, des équipements tels que le canon français CAESAR (camion équipé d'un système d'artillerie) ont la capacité de tirer 6 à 8 obus par minute à une distance de 42 km des obus avec une précision de 50m.

**Figure 1** – Photo d'un canon CAESAR lors d'un tir d'obus

Ainsi, ces équipements sont redoutables sur le terrain. On cherche donc à les détruire dès que possible. Une méthode pour les endommager, voire les détruire est le contre-feu. Cela consiste à déterminer la position du canon d'artillerie à l'aide de la trajectoire de l'obus. En connaissant sa position, on peut procéder à un tir d'artillerie avec pour objectif de le toucher. Le problème principal est le temps. En effet, quand le canon d'artillerie procède à son tir, il peut y avoir plusieurs minutes avant que l'obus touche le sol. Le canon a donc la possibilité, pendant ce temps, de se replier et de partir, rendant le contre-feu inutile.

Dans le cadre du projet de 211, nous avons choisi d'étudier un tir "parabolique", du type tir de canon, plus précisément, nous nous sommes intéressés à retrouver les paramètres de mise à feu à partir de points décrivant la trajectoire du projectile.

Nous avons d'abord cherché un modèle physique en 2D de ces tirs, créer un programme python donnant une trajectoire à partir de ce modèle et chercher à retrouver dans un premier tant l'angle de tir, la vitesse initiale. Ensuite, nous passerons en 3D et nous chercherons à connaître la position initiale du tir et différents autres paramètres en ne connaissant que la fin de la courbe puis en voyant apparaître les points au cours du temps.

# 1 Présentation du modèle

Le premier modèle choisi est un modèle de connaissance basé sur les principes physiques de la trajectoire balistique en 2D $(0, \vec{x}, \vec{z})$.

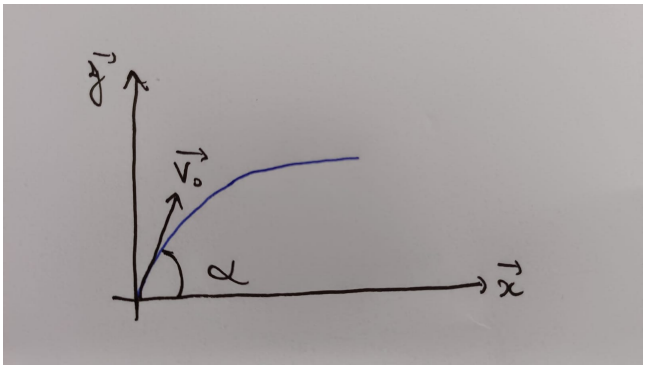



**Figure 2** – Représentation des paramètres en 2D

Référentiel : terrestre supposé galiléen.
Système : Projectile de masse m, de forme sphérique (coefficient de traînée Cx, surface frontale S), projeté avec une vitesse initiale $v_0$ avec un angle $\alpha$.
Forces : Poids ($\vec{P} = m\vec{g}$) et la force de traînée ($F_T = -\frac{1}{2}\rho CxS\vec{v^2}$)

On applique le PFD projeté sur les 2 axes ce qui donne :

$$m\frac{d^2x}{dt^2} + \frac{1}{2}\rho CxS(\frac{dx}{dt})^2 = 0$$

$$m\frac{d^2z}{dt^2} + \frac{1}{2}\rho CxS(\frac{dz}{dt})^2 + mg = 0$$

À partir de ces équations, ont écrit une fonction "canon" qui prend deux valeurs de $v_0$ et $\alpha$ et donne en sortie deux listes correspond aux coordonnées (x,y) d'un certain nombre de points définissant la trajectoire. On écrit aussi une fonction "Création Données" qui créer un fichier texte regroupant les données précédemment définies et une fonction "lecture données" qui extrait les données à partir d'un fichier texte. Les différents paramètres et les deux listes de coordonnées représentent les données du problème, l'objectif est maintenant de trouver les deux inconnues $V_0$ et $\alpha$.

# 2 Résolution du problème en 2D

## 2.1 Définition d'un critère et d'une méthode de convergence

Pour résoudre ce problème, on a tout d'abord utilisé les méthodes simples que sont la descente de gradients avec un calcul par différences finies et les moindres carrés.

### 2.1.1 Descente de gradient

Pour le calcul de gradient par différences finies, on utilise la formule :

$$g(\theta) = \frac{(j(\theta + \Delta) - j(\theta - \Delta))}{2\Delta}$$

Avec $\Delta$ fixé et j le critère à optimiser.

Par sécurité, on ajoutera certaines limites sur des paramètres (par exemple, la vitesse initiale ne pourra pas descendre en dessous de $1\,\mathrm{ms}^{-1}$ de même la masse minimale sera de $1\,\mathrm{kg}$) Cela évitera d'avoir des valeurs aberrantes dans les calculs qui risque de poser des problèmes (du type division par zéro) dans le calcul de la trajectoire.

Ensuite, on ajoutera une seconde sécurité qui consiste à normer le gradient. Cela réduira la vitesse de convergence, mais cela aura pour effet d'éviter que l'on ait un gradient trop grand et que donc on sorte des limites du système. La norme du gradient aura pour effet de rendre la précision du résultat limité tant que le pas adaptatif ne sera pas mis en place.

### 2.1.2 Critère des moindres carrés

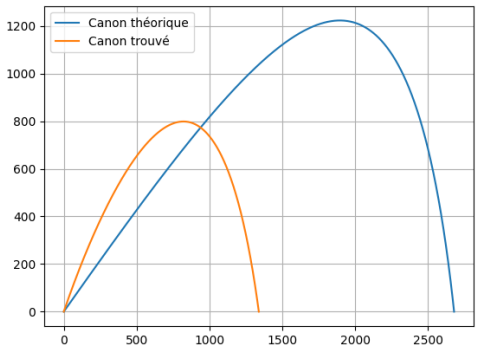
Pour le critère, on choisit le critère des moindres carrées que l'on obtient en utilisant la formule :

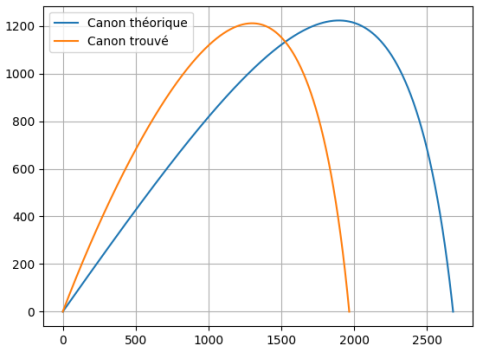
$$j_{MC}(p) = \sum_{i=0}^{N} (z(i) - z_m(p,i))^2 + (x(i) - x_m(p,i))^2$$

Avec p les paramètres, N le nombre de points, x et y les positions connues, $x_m$ et $y_m$ les positions du modèle.

Pour une trajectoire en 2D, on code la méthode des moindres carrés et la descente de gradient (voir annexe : Extrait de code n°1)

On obtient alors les courbes suivantes toutes les 100 itérations :



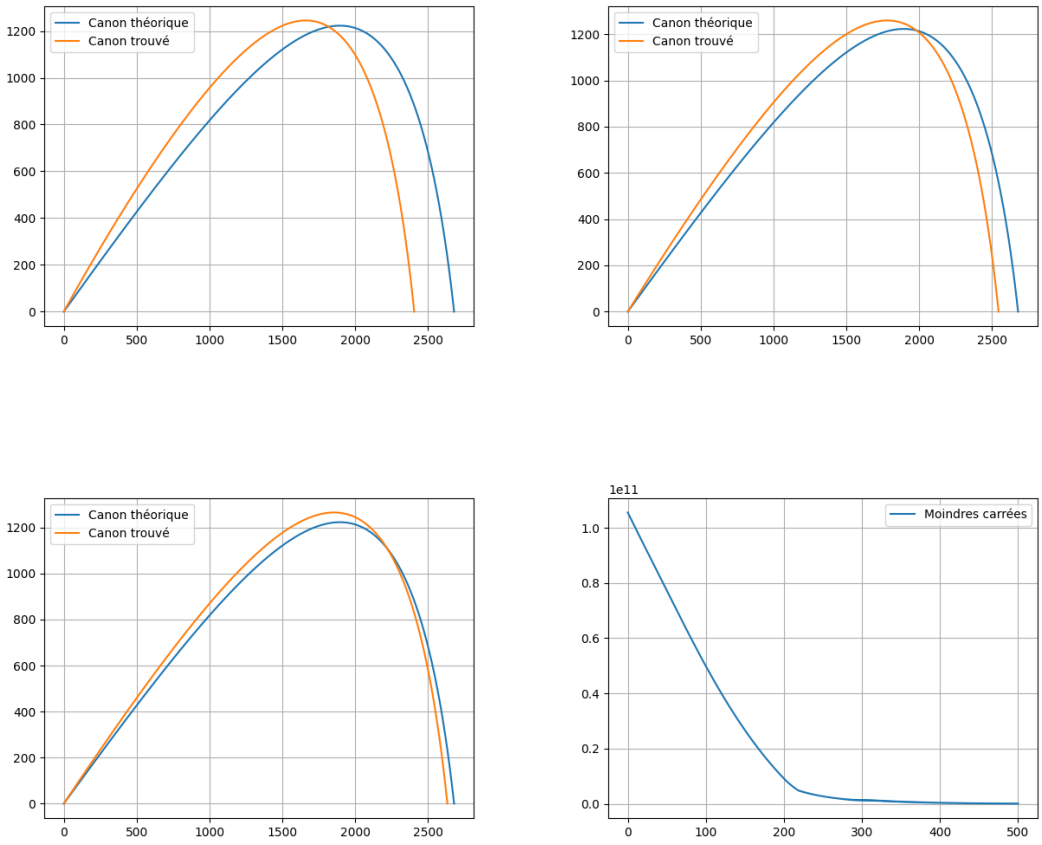



**Figure 3** – Trajectoires pour 100, 200, 300, 400, 500 itérations et évolution des moindres carrés

On observe la convergence progressive du tir vers le modèle, le critère des moindres carrées diminue beaucoup moins vite après la 200ième itération. Si on laisse le code tourner pendant plus d'itérations, on observe une convergence lente vers la courbe théorique.

## 2.2 Mise en place d'un pas adaptatif

On a ensuite implanté un pas variable à la descente de gradient, c'est-à-dire que l'on va normer le gradient puis on va l'utiliser pour trouver les nouvelles valeurs des paramètres, mais si le critère à minimiser est plus élevé avec ses nouveaux paramètres, on va étudier le point qui est entre le nouveau et l'ancien jusqu'à avoir une fonction à minimiser qui diminue.

Par sécurité, il ne sera pas possible d'avoir un gradient trop petit pour éviter d'être bloqué à un point. On préfèrera donc augmenter $j$ que de rester statique.

On garde ainsi le même code que précédemment, mais en ajoutant la partie pas adaptative (voir annexe : extrait de code n°2).

Le pas adaptatif consiste à diviser la valeur du gradient pas jusqu'à obtenir un point qui a un critère inférieur au point de précédent.

# 3 Résolution d'un problème 3D avec tous les points de la trajectoire

On passe maintenant en 3D. On utilise toujours le même critère et la même descente de gradient avec le pas adaptatif. Le passage de la 2D à la 3D nécessite de faire apparaître de nouveaux paramètres

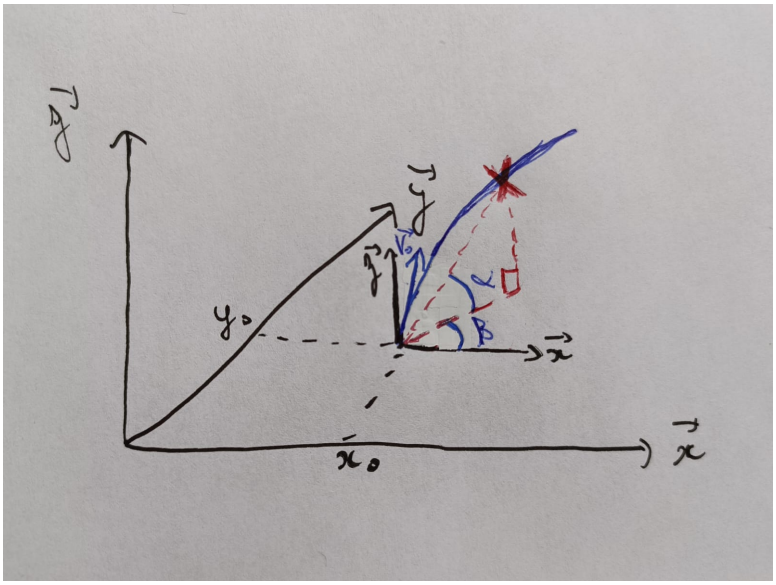
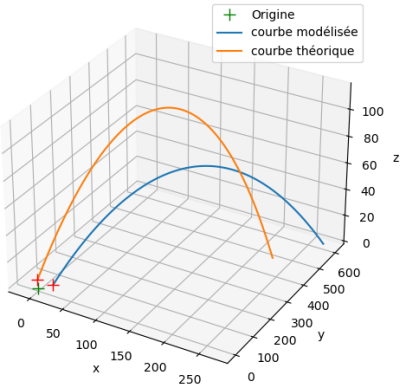


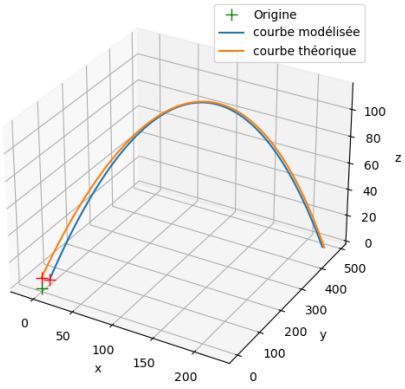
**Figure 4** – Représentation des paramètres en 3D

On décide aussi de complexifier le problème en ajoutant les éléments suivants :

- passage à la 3D donc ajout de coordonnées suivant y
- ajout de la masse du projectile comme inconnue
- inconnus supplémentaires : position initiale (x0 ;y0).
- ajout d'un angle $\beta$ pour orienter le tir

On adapte le code à ces nouveaux paramètres et on obtient les courbes suivantes :



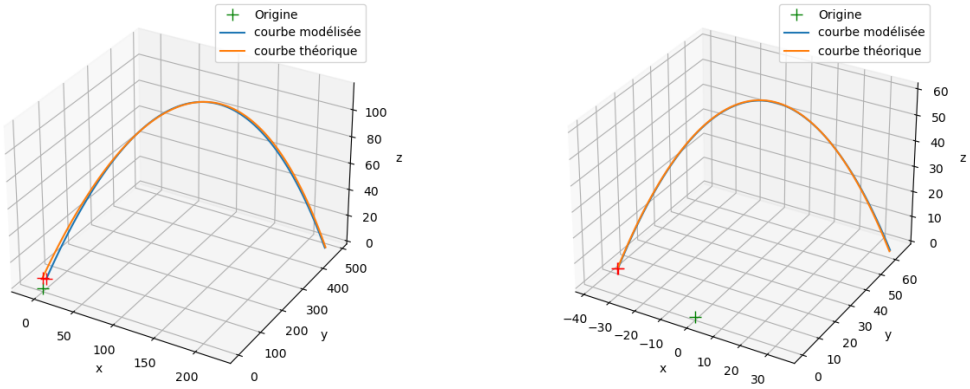



**Figure 5** – Trajectoires pour 10, 100, 300 et 800 itérations

On obtient une précision sur les moindres carrés de $10^{-3}$ à la 821ième itération, ce qui donne un temps de calcul de 182 secondes. Le programme converge, mais trop lentement, il faut opter pour une nouvelle solution de calcul numérique afin de réduire le temps de calcul.

# 4 Fonctions de sensibilités

Pour calculer le gradient, on va maintenant opter pour une méthode plus optimisée : les fonctions de sensibilité.
En notant $e(\theta, i)$ l'erreur sur le vecteur paramètre thêta à la ième itération, la fonction de sensibilité s'écrit :

$$s_e(\theta, i) = \frac{\partial e(\theta, i)}{\partial \theta}$$

Ici avec le critère des moindres carrés, le gradient vaudra

$$g(\theta) = 2\sum_{i=1}^{n} e(\theta, i) s_e(\theta, i)$$

.
Pour obtenir le gradient, il faudra donc calculer les fonctions de sensibilité. On se limite pour l'instant aux paramètres $\alpha$ et $V_0$. En reprenant les équations du modèle :

$$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + \frac{1}{2}\rho C_x S(\frac{\mathrm{d}x}{\mathrm{d}t})^2 = 0$$

$$m\frac{\mathrm{d}^2y}{\mathrm{d}t^2} + \frac{1}{2}\rho C_y S(\frac{\mathrm{d}y}{\mathrm{d}t})^2 = 0$$

$$m\frac{\mathrm{d}^2z}{\mathrm{d}t^2} + \frac{1}{2}\rho C_z S(\frac{\mathrm{d}z}{\mathrm{d}t})^2 + mg = 0$$

On obtient sur $\vec{x}$ en posant $\varepsilon = -0.5\rho S C_x$ :

$$\frac{\partial V_x}{\partial t} = \frac{\varepsilon}{m}V_x^2 \Rightarrow \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial V_x}{\partial \alpha} = 2\frac{\varepsilon}{m}\frac{\partial V_x}{\partial \alpha}V_x \Rightarrow \frac{\mathrm{d}^2 S_\alpha}{\mathrm{d}t^2} = 2\frac{\varepsilon}{m}\frac{\mathrm{d}S_\alpha}{\mathrm{d}t}V_x$$

$$\frac{\partial V_x}{\partial t} = \frac{V_0}{m} V_x^2 \Rightarrow \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial V_x}{\partial V_0} = 2\frac{\varepsilon}{m}\frac{\partial V_x}{\partial V_0} V_x \Rightarrow \frac{\mathrm{d}^2 S_{V_0}}{\mathrm{d}t^2} = 2\frac{\varepsilon}{m}\frac{\mathrm{d}S_{V_0}}{\mathrm{d}t} V_x$$

On procède de la même manière sur les autres axes.
Ensuite, en utilisant Euler, on a les formules suivantes :

$$S_{i+1} = \frac{\mathrm{d}S_{i+1}}{\mathrm{d}t}\mathrm{d}t + S_i$$
$$\frac{\mathrm{d}S_{i+1}}{\mathrm{d}t} = \frac{\mathrm{d}^2 S_i}{\mathrm{d}t^2}\mathrm{d}t + \frac{\mathrm{d}S_i}{\mathrm{d}t}$$
$$\frac{\mathrm{d}^2 S_{i+1}}{\mathrm{d}t^2} = 2\frac{\varepsilon}{m}\frac{\mathrm{d}S_i}{\mathrm{d}t} V$$

On devra les coder pour chaque ses équations, pour chaque direction (x, y et z) et pour chaque paramètre.

Maintenant que l'on a vérifié que les fonctions de sensibilités fonctionnent pour deux paramètres, on les calcule pour le reste des paramètres :
Pour la masse m on a :

$$\frac{\partial V_x}{\partial t} = \frac{\varepsilon}{m} V_x^2 \Rightarrow \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial V_x}{\partial m} = -\frac{\varepsilon}{m^2} V_x^2 + 2\frac{\varepsilon}{m}\frac{\partial V_x}{\partial m} V_x \Rightarrow \frac{\mathrm{d}^2 S}{\mathrm{d}t^2} = -\frac{\varepsilon}{m^2} V_x^2 + 2\frac{\varepsilon}{m}\frac{\mathrm{d}S}{\mathrm{d}t} V_x$$

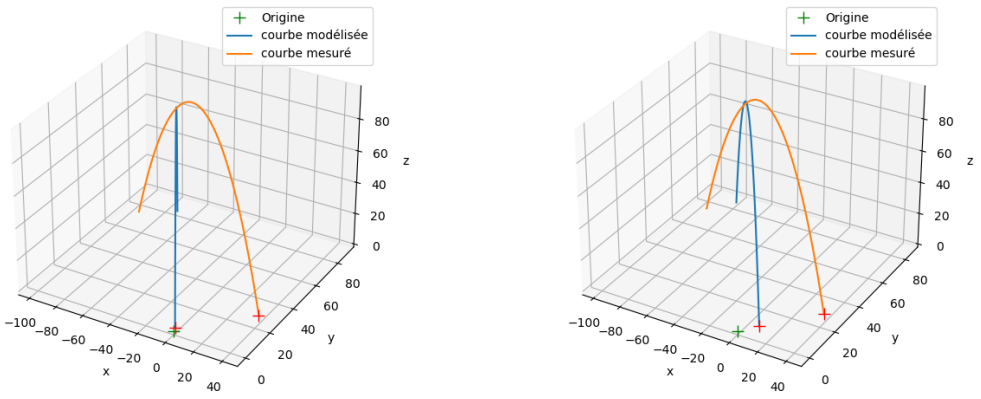
Pour les autres inconnues $\beta$, $x_0$, $y_0$, on a le même résultat que pour $V_0$ et $\alpha$.
Par contre, les conditions initiales varient pour chaque fonction de sensibilité :

**Table 1** – Conditions initiales des fonctions de sensibilités

| | $x_0$ | $y_0$ | $\alpha$ | $\beta$ | $V_0$ | $m$ |
|---|---|---|---|---|---|---|
| $S_x$ | 1 | 0 | 0 | 0 | 0 | 0 |
| $S_y$ | 0 | 1 | 0 | 0 | 0 | 0 |
| $S_z$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $\frac{\mathrm{d}S_x}{\mathrm{d}t}$ | 0 | 0 | $-V_0 \sin(\alpha)\cos(\beta)$ | $-V_0 \cos(\alpha)\sin(\beta)$ | $\cos(\alpha)\cos(\beta)$ | 0 |
| $\frac{\mathrm{d}S_y}{\mathrm{d}t}$ | 0 | 0 | $-V_0 \sin(\alpha)\sin(\beta)$ | $V_0 \cos(\alpha)\cos(\beta)$ | $\cos(\alpha)\sin(\beta)$ | 0 |
| $\frac{\mathrm{d}S_z}{\mathrm{d}t}$ | 0 | 0 | $V_0 \cos(\alpha)$ | 0 | $\sin(\alpha)$ | 0 |

On code ces nouvelles valeurs (voir annexe : extrait de code n°3).

On obtient les résultats suivants :



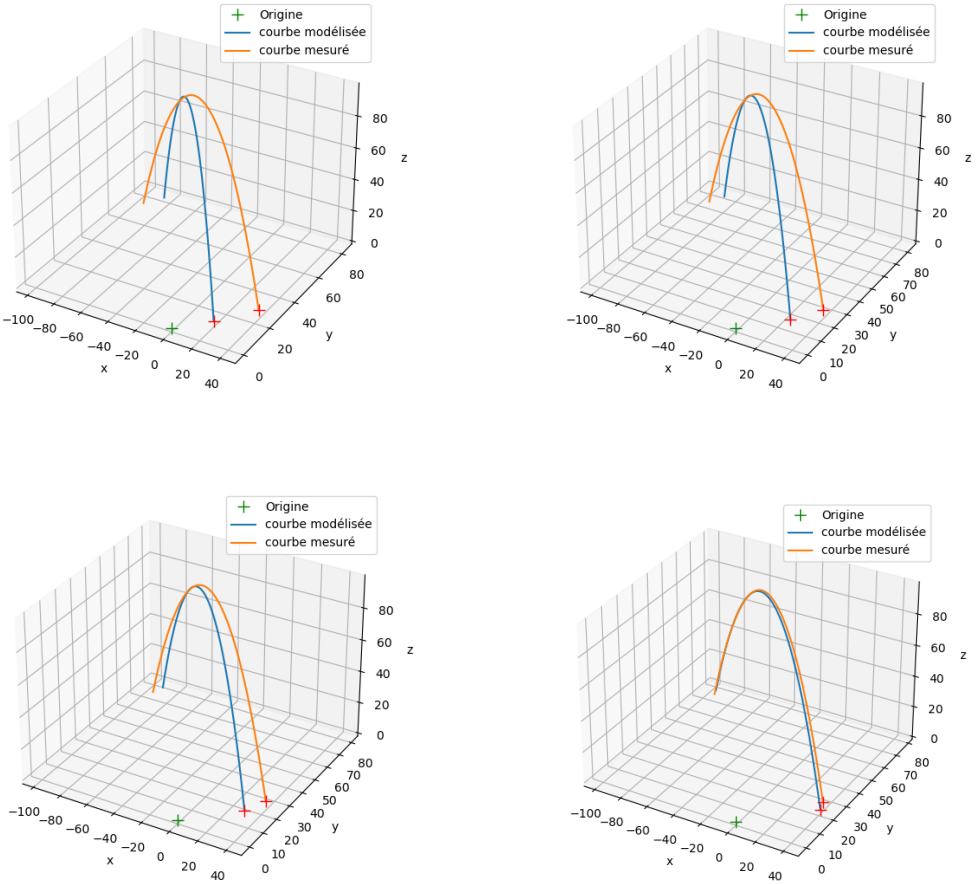



**Figure 6** – Trajectoires pour 50, 500, 1000, 1500, 2000 et 3050 itérations

# 5 Résolution d'un problème 3D sans les points du début de la trajectoire

Le but est maintenant de s'approcher au mieux du cas réel. En effet, à quoi cela sert de chercher à modéliser la courbe pour trouver le point initial alors que l'on mesure le point initial ? Ce qui se passe en réalité est que l'on ne connaît pas les premiers points. En effet, le radar ne détectera l'obus que s'il est assez haut. On a ainsi plus accès aux points du début de la trajectoire. Nous allons donc supprimer aléatoirement un certain nombre de points (entre 30% et 50%) des points du début de la trajectoire.
On observe des problèmes de convergence avec cette méthode, car le système à du mal à calibrer la quantité de points manquants au début de la trajectoire et à gérer ce manque d'informations.

Pour pallier à ce problème, on ajoute le paramètre $t_0$. Ce paramètre correspond à l'instant où l'on a le premier point de mesure. Ainsi, le calcul de la trajectoire commencera à l'instant $-t_0$. On calculera les positions $n_0$ fois avec $n_0 = \frac{t_0}{\mathrm{d}t}$ et $\mathrm{d}t$ le pas de temps qui est de 100 ms. On va faire évoluer notre code avec les différences finies et avec les sensibilités pour comparer le résultat.
On pose donc $x_{t_0}(t) = x(t - t_0)$ On calcule les fonctions de sensibilités :

**Table 2** – Conditions initiales des fonctions de sensibilités pour $t_0$

| | $S_x$ | $S_y$ | $S_z$ | $\frac{\mathrm{d}S_x}{\mathrm{d}t}$ | $\frac{\mathrm{d}S_y}{\mathrm{d}t}$ | $\frac{\mathrm{d}S_z}{\mathrm{d}t}$ |
|---|---|---|---|---|---|---|
| $t_0$ | $-x(t_0)$ | $-y(t_0)$ | $-z(t_0)$ | 0 | 0 | 0 |

Avec ce nouveau code (voir annexe : extrait de code n°3), on obtient finalement les résultats suivant :

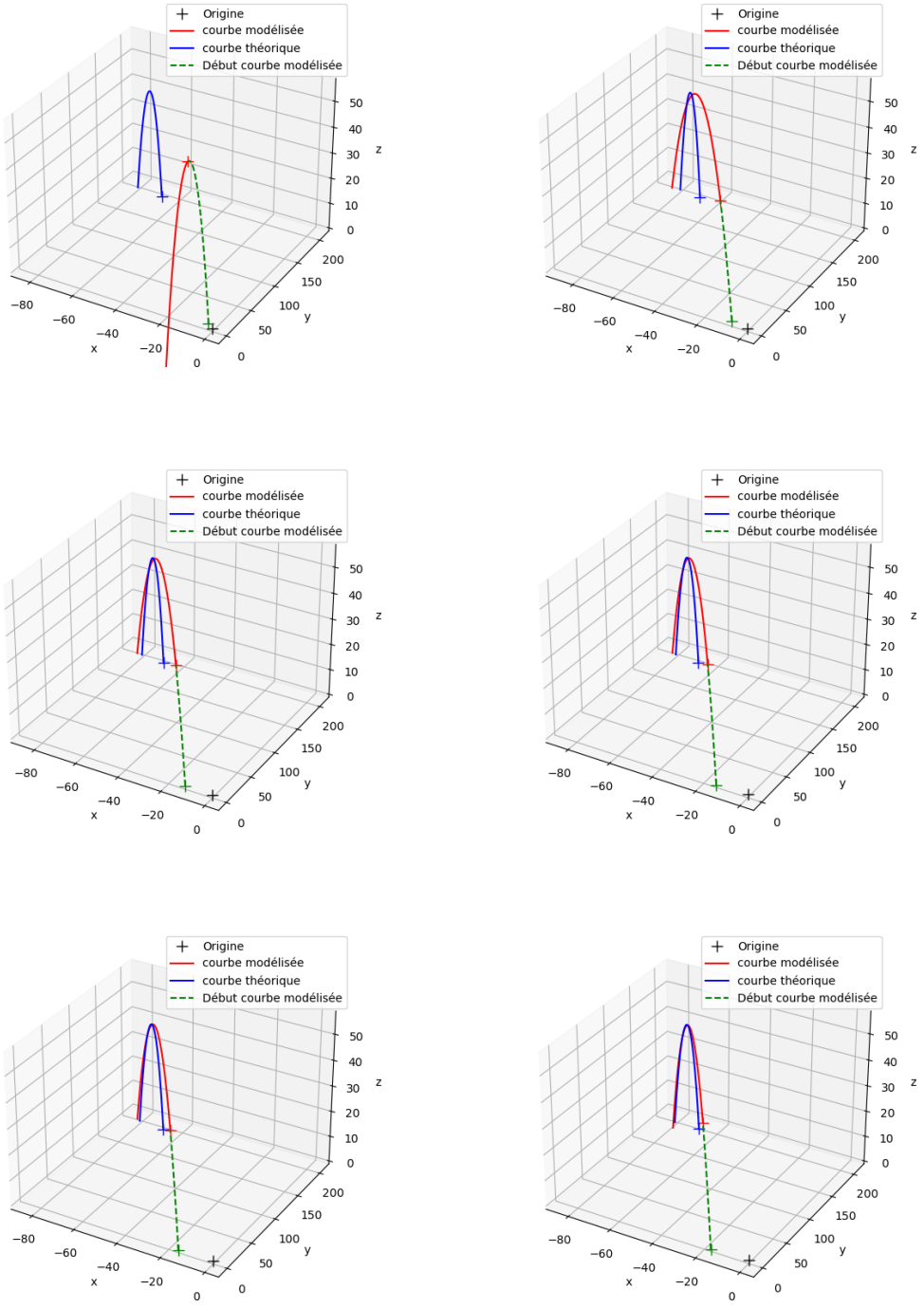



**Figure 7** – Trajectoires pour 100, 4000, 10000, 15000, 20000 et 27000 itérations

Après avoir fait 25 essais, on remarque que l'on gagne 25% de temps de calcul pour trouver une précision de 1 avec les fonctions de sensibilités, mais que l'on obtient plus rapidement une meilleure précision lors de l'utilisation des différences finies.

# 6 Résolution d'un problème 3D avec l'apparition des points au cours du temps

Le but de cette partie est encore de s'approcher du cas réel. En effet, après qu'un canon CAESAR a tiré, il lui faut ensuite très peu de temps pour se mettre en mouvement. On cherche à optimiser les moindres millisecondes possibles pour faire les calculs et trouver sa position. Ainsi, dans cette partie, nous allons commencer les calculs pour trouver les paramètres alors que nous n'avons pas encore tous les points de la trajectoire.

Nous allons donc ajouter les positions au cours du temps. On va ainsi attendre d'avoir 5 points de calcul puis toutes les 100 ms, on va ajouter une nouvelle position à nos mesures. Cela aura pour effet de commencer le calcul des paramètres alors que l'on n'a pas encore tous les points (voir annexe : extrait de code n°3).
On remarque lors de l'utilisation du programme que le temps de calcul reste important et que lorsque l'obus touche le sol, on est loin de savoir d'où il a été tiré. Il faut donc encore plusieurs minutes de calculs pour avoir une précision qui commence à être acceptable.

# Conclusion

Pour conclure, l'étude d'une trajectoire d'obus nous a permis de confronter diverses méthodes d'optimisation numérique. En partant d'un programme simple de descente de gradient non optimisé qui fonctionnait dans un cas 2D mais qui mettait beaucoup plus de temps dans un cas 3D lorsque la complexité du modèle augmente. On a ensuite choisi d'utiliser les fonctions de sensibilité pour le calcul du gradient, cela demande quelques calculs à la main mais est beaucoup plus rapide que le calcul par différence. Pour augmenter la complexité après être passé en 3D, nous avons ajouté comme inconnues une partie de la trajectoire (entre 30% et 50%) ainsi que le point d'origine du tir. Cela nous rapproche des conditions réelles lors de contre tir, car les radars prennent du temps à percevoir le tir, la trajectoire est donc incomplète.

# A Annexe

## A.1 Extrait de code n°1 : Moindres carres en 2D

**Code Python :**

```python
def Moindres_carres(Lparametre):
    """
    Fonction de calcul des moindres carrées entre les positions réelles et les positions calculés
    Variable d'entrée: liste des paramètres\n
    Retourne la valeur des moindres carrées pour ce tuple de paramètres.\n
    """
    Lx,Ly,Lz=Canon(Lparametre)
    #Initialisation Ldi: listes des différences aux carrées de chaque point selon l'axe i
    Ldx=[]
    Ldy=[]
    Ldz=[]

    #Calcul des moindres carrés
    for i in range(len(Lx0)):
        Ldx.append((Lx[i]-Lx0[i])**2)
        Ldy.append((Ly[i]-Ly0[i])**2)
        Ldz.append((Lz[i]-Lz0[i])**2)
    MC=sum(Ldx+Ldy+Ldz)/len(Lx0)
    return MC
```

## A.2 Extrait de code n°2 : Descente de gradient

**Code Python :**

```python
def Descente_Gradient(p0=[45,10],FctAMinimiser=Moindres_carres,eps=1e-3):
    """
    Fonction de descente de gradient avec les sensibilités
    Variable d'entrée: liste des paramètres initiales; la fonction utilisé pour minimiser, la valeur final maximale que l'on souhaite\n
    Retourne les paramètres finaux.\n
    """

    # Initialisation
    delta = 1e-6
    params = p0
    MC=FctAMinimiser(params)
    print("Étape {:5d} : theta={:.3f} V0={:.3f} MC= {:.3e}".format(0,params[0],params[1],MC))
    LMC=[MC]

    # Boucle de descente de gradient
    n=0
    while MC>eps:
        n+=1
        # Calcul du gradient estimé de la fonction de perte au point actuel
        gradient = [0]*len(params)

        for j in range(len(params)):
            params_plus = np.copy(params)
            params_plus[j] += delta
            params_moins = np.copy(params)
            params_moins[j] -= delta
            gradient[j] = (FctAMinimiser(params_plus) - FctAMinimiser(params_moins)) / (2*delta)

        #Pas adptatif
        gradient=normer(gradient)
        params_suivant=[]
        for i in range(len(params)):
            params_suivant.append(params[i]-gradient[i])

        #Sécurité
```

```
36        params_suivant[-1]=max(1,params_suivant[-1])
37
38        MC=FctAMinimiser(params_suivant)
39
40        #limite en norme du gradient en fonction de la valeur de la fonction a minimiser
41        normeMin=1e-4
42        #Pas adaptatif dans la limite de la norme du gradient
43        while MC>LMC[-1] and np.linalg.norm(gradient)>=normeMin:
44           for i in range(len(params)):
45              gradient[i]*=0.5
46              params_suivant[i]=params[i]-gradient[i]
47           #Sécurité
48           params_suivant[-1]=max(1,params_suivant[-1])
49           MC=FctAMinimiser(params_suivant)
50
51        #Mise à jour des paramètres et affichage des résultats
52        params=params_suivant[:]
53
54        LMC.append(MC)
55        print("Étape {:5d} : theta={:.3f} V0={:.3f} MC= {:.3e}".format(n,params[0],params[1],MC
              ))
56
57        #Affichage 3D qui se met à jour toutes les 100 étapes
58        if n%10==0 :
59           # Affichage de l'étape finale
60           Lx,Ly,Lz=Canon(params)
61
62           plt.close()
63           affichage2D([[Lx0,Lz0,'Canon théorique'],[Lx,Lz,'Canon trouvé']])
64           plt.draw()
65           plt.savefig("2D "+str(n)+"iterations")
66           plt.pause(0.5)
67
68     # Affichage de l'étape finale
69     print(">>> Fin: Étape {:5d} : theta={:.3f} V0={:.3f} MC= {:.3e}".format(n,params[0],params
           [1],MC))
70     Lx,Ly,Lz=Canon(params)
71
72     plt.close()
73     affichage2D([[Lx0,Lz0,'Canon théorique'],[Lx,Lz,'Canon trouvé']])
74     plt.draw()
75     plt.pause(2)
76     plt.savefig("2D "+str(n)+"iterations")
77     plt.close()
78     affichage2D([[[i for i in range(len(LMC))], LMC,'Moindres carrées']])
79     plt.draw()
80     plt.pause(2)
81     plt.savefig("2D MC au cours des itérations")
82
83     #Enregistrement du résultat dans un fichier
84     fichier=open('Resultat.txt','a')
85     fichier.write(FctAMinimiser.__name__+":\n Étape {:5d} : theta={:.5f} V0={:.5f} MC= {:.3e}\
           n".format(n,params[0],params[1],MC))
86     fichier.close()
87
88     return(params)
```

## A.3 Extrait de code n°3 : Calcul des fonctions de sensibilités et ajout de $t_0$

**Code Python :**

```
1  def Canon(Lparametre):
2     """
3     Fonction qui calcule les positions en X, Y et Z du boulet de canon au cours du temps.\n
4     Variable d'entrée: liste des paramètres\n
5     Retourne des listes contenant les positions en X, Y et Z et la liste contenant les
          sensibilités pour chaque paramètres.\n
```

```
    On utilisera un algorithme d'Euler pour calculer les positions.\n
    On calculera aussi la sensibilité en fonction de chaque paramètre.
    """
    global rho,S,Cx,Cy,Cz,g,masse

    x0=Lparametre[0]
    y0=Lparametre[1]
    theta=Lparametre[2]
    gamma=Lparametre[3]
    vitesseInitiale=Lparametre[4]
    masse=Lparametre[5]
    t0=Lparametre[6]

    #Passage des angles de deg en rad
    alpha=theta*3.14/180
    beta=gamma*3.14/180

    #Autres paramètres divers
    vitesseVent=[0,0,0]
    rho=1.225
    S=(0.121/2)**2*np.pi
    Cx=0.3
    Cy=0.3
    Cz=0.3
    g=9.81
    dt=1e-1

    #Définitions des conditions initiales
    Lt=[-t0]
    Ldx=[vitesseInitiale*np.cos(alpha)*np.cos(beta)]
    Ldy=[vitesseInitiale*np.cos(alpha)*np.sin(beta)]
    Ldz=[vitesseInitiale*np.sin(alpha)]
    Lx=[x0]
    Ly=[y0]
    Lz=[0]

    n0=int(10*t0)

    K=-0.5*rho*S*Cx #Constante pour le calcul de la sensibilité

    LS=[[[1],[0],[0],[0],[0],[0],[-Lx[-1]]], #S de X
       [[0],[1],[0],[0],[0],[0],[-Ly[-1]]], #S de Y
       [[0],[0],[0],[0],[0],[0],[-Lz[-1]]]] #S de Z

    LdS=[[[0],[0],[-vitesseInitiale*np.sin(alpha)*np.cos(beta)],[-vitesseInitiale*np.cos(alpha
        )*np.sin(beta)],[np.cos(alpha)*np.cos(beta)],[0]],
        [[0],[0],[-vitesseInitiale*np.sin(alpha)*np.sin(beta)],[vitesseInitiale*np.cos(alpha)*
            np.cos(beta)],[np.cos(alpha)*np.sin(beta)],[0]],
        [[0],[0],[vitesseInitiale*np.cos(alpha)],[0],[np.sin(alpha)],[0]]]

    #Calcul des positions du canon au cours du temps avec Euler
    for i in range(n0+len(Lx0)-1):
       #Calcul des positions et vitesses à partir de la 1ère position mesuré
       Lt.append(Lt[-1]+dt)
       Ldx.append(Ldx[-1]+dt*fx(Ldx[-1]+vitesseVent[0]))
       Lx.append(Lx[-1]+dt*Ldx[-1])

       Ldy.append(Ldy[-1]+dt*fy(Ldy[-1]+vitesseVent[1]))
       Ly.append(Ly[-1]+dt*Ldy[-1])

       Ldz.append(Ldz[-1]+dt*(fz(Ldz[-1]+vitesseVent[2])-g))
       Lz.append(Lz[-1]+dt*Ldz[-1])

       #Calcul de la sensibilité
       LdS[0][0].append(2*K/masse*Ldx[-1]*LdS[0][0][-1]*dt+LdS[0][0][-1])#X x0
       LdS[0][1].append(2*K/masse*Ldx[-1]*LdS[0][1][-1]*dt+LdS[0][1][-1])#X y0
       LdS[0][2].append(2*K/masse*Ldx[-1]*LdS[0][2][-1]*dt+LdS[0][2][-1])#X alpha
       LdS[0][3].append(2*K/masse*Ldx[-1]*LdS[0][3][-1]*dt+LdS[0][3][-1])#X beta
       LdS[0][4].append(2*K/masse*Ldx[-1]*LdS[0][4][-1]*dt+LdS[0][4][-1])#X V0
       LdS[0][5].append((-K*(Ldx[-1]**2)/(masse**2)+2*K/masse*LdS[0][5][-1]*Ldx[-1])*dt+LdS
           [0][5][-1])#X
          m
```

```
        LS[0][6].append(Lx[-1])#X t0

        LdS[1][0].append(2*K/masse*Ldy[-1]*LdS[1][0][-1]*dt+LdS[1][0][-1])#Y x0
        LdS[1][1].append(2*K/masse*Ldy[-1]*LdS[1][1][-1]*dt+LdS[1][1][-1])#Y y0
        LdS[1][2].append(2*K/masse*Ldy[-1]*LdS[1][2][-1]*dt+LdS[1][2][-1])#Y alpha
        LdS[1][3].append(2*K/masse*Ldy[-1]*LdS[1][3][-1]*dt+LdS[1][3][-1])#Y beta
        LdS[1][4].append(2*K/masse*Ldy[-1]*LdS[1][4][-1]*dt+LdS[1][4][-1])#Y V0
        LdS[1][5].append((-K*(Ldy[-1]**2)/(masse**2)+2*K/masse*LdS[1][5][-1]*Ldy[-1])*dt+LdS
            [1][5][-1])#Y
            m
        LS[1][6].append(Ly[-1])#Y t0

        LdS[2][0].append(2*K/masse*Ldz[-1]*LdS[2][0][-1]*dt+LdS[2][0][-1])#Z x0
        LdS[2][1].append(2*K/masse*Ldz[-1]*LdS[2][1][-1]*dt+LdS[2][1][-1])#Z y0
        LdS[2][2].append(2*K/masse*Ldz[-1]*LdS[2][2][-1]*dt+LdS[2][2][-1])#Z alpha
        LdS[2][3].append(2*K/masse*Ldz[-1]*LdS[2][3][-1]*dt+LdS[2][3][-1])#Z beta
        LdS[2][4].append(2*K/masse*Ldz[-1]*LdS[2][4][-1]*dt+LdS[2][4][-1])#Z V0
        LdS[2][5].append((-K*(Ldz[-1]**2)/(masse**2)+2*K/masse*LdS[2][5][-1]*Ldz[-1])*dt+LdS
            [2][5][-1])#Z
            m
        LS[2][6].append(Lz[-1])#Z t0

        for xyz in range(len(LdS)):
            for param in range(len(LdS[0])):
                LS[xyz][param].append(dt*LdS[xyz][param][-1]+LS[xyz][param][-1])

    return(Lx,Ly,Lz,LS)
```

## A.4 Extrait de code n°4 : Mise en place de la récursivité

**Code Python :**

```
def lecture_donnes(fichier='Donnees_test.txt'):
    """
    Fonction de lecture des données de la courbe àretrouver
    Variable d'entrée: nom du fichier ou sont enregistré les données.\n
    Retourne les listes de positions selon X, Y et Z \n
    """
    #lecture du fichier
    fichier=open(fichier,'r')
    doc=fichier.readlines()
    fichier.close()

    #Séparation des données et de l'entête (contient les paramètres exactes)
    lignes=doc[7:]
    entete=doc[:7]

    #Notation de l'entête dans un fichier ou l'on écrira les paramètres trouvés
    fichier2=open('Resultat.txt','a')
    fichier2.write('\n')
    fichier2.write('Canon 3D Sensibilités\n')
    for i in entete:
        fichier2.write(i)
    fichier2.write('\n')
    fichier2.close()

    #Création des listes de positions en global et d'une valeur contenant le nombre de
        positions
    global Lx0,Ly0,Lz0,taille
    taille=len(lignes)
    Lx0=[]
    Ly0=[]
    Lz0=[]
    for ligne in lignes:
        position=(ligne.split('\n')[0]).split(';')
        Lx0.append(float(position[0]))
        Ly0.append(float(position[1]))
        Lz0.append(float(position[2]))

    return(Lx0,Ly0,Lz0)
```