Département EEA - Faculté Sciences et Ingénierie

Master EEA Parcours AURO

Optimisation et Estimation

Année 2022-2023

Compte rendu de travaux pratiques : Calcul numérique d'un MGI d'un RRR par PNL

Rédigé par RONK Antoine et GABRIEL CALIXTE Damien

29 septembre 2022

1 Méthode de Newton

1.1 Implémentation

Initialisation :

Pour l'implémentation de la méthode de Newton, nous avons commencé par initialiser `qinit` et fixer le *pas* et l'*erreur tolérée* ϵ. Par défaut, on a fixé le *pas* à 0.1 et *epsilon* à 0.001. La partie d'initialisation se termine avec la saisie par l'utilisateur du nombre maximum d'itération (utile pour stopper l'algorithme une fois ce nombre atteint).

Itération :

La boucle d'itération est structuré comme suit :

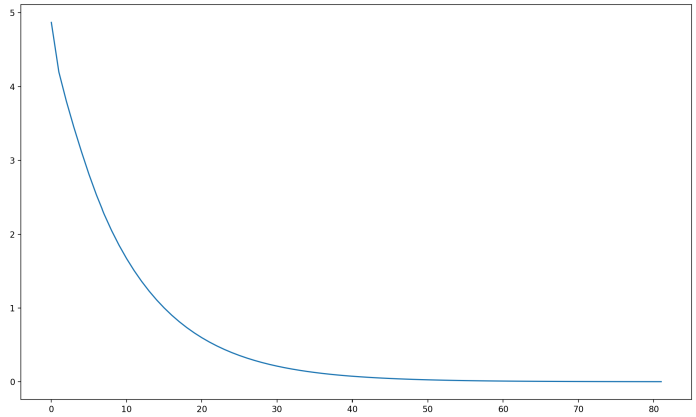
```
while(Hnorm>eps and i<NbIter) :
    # Jacobienne
    J = jacobienne(q)
    # Calcul de l'inverse
    Jinv=np.linalg.inv(J)
    # Calcul de l ecart entre le Xbut et le Xc actuel
    H=(Xbut-mgd(q))
    # Iteration
    q=q+pas*np.dot(Jinv,H)
    # Calcul de l erreur avec norme euclidienne
    Hnorm=np.linalg.norm(H)
    # Remplissage du vecteur avec la norme
    Hq.append(Hnorm)
    i=i+1
```

Listing 1 – Code de l'implémentation de la méthode des gradients

La boucle se lance si la métrique choisie (ici on a considéré la norme euclidienne) est supérieure à l'erreur tolérée ϵ et si l'indice de la boucle est plus petit que le nombre maximum d'itération. On y a calcul la jacobienne puis son inverse ainsi que la fonction $H(q)$. Ceci pour calculer l'équation de récurrence suivante :

$$q_{k+1} = q_k + pas \cdot J^{-1}(q)\left(X_d - f(q)\right)$$

Il reste alors à calculer l'erreur en utilisant la norme euclidienne ainsi que sauvegarder le résultat de l'erreur dans un tableau pour afficher son évolution au cours du temps.
Par exemple, on obtient le profil de convergence suivant pour un pas de $0,1$ et une erreur $\epsilon = 1 \times 10^{-3}$:



Évolution de l'erreur pour $\epsilon = 0.001$ et pas $= 0.1$

Notons que la boucle n'a pas été arrêtée par le nombre maximum d'itération mais par le critère ϵ. En effet, on a réalisé 82 itérations.

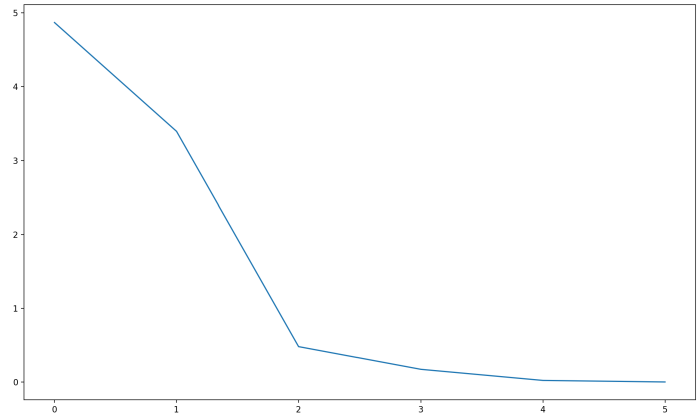
```
## METHODE DE NEWTON
La configuration de depart est [2.0943951  0.43633231 0.78539816]
Saisir le nombres d iterations : 100
La boucle a ete executee  82 fois
Evolution de l erreur
Position optimale trouvee avec Newton
```

Capture du shell après exécution de 82 itérations

Nous pouvons aussi noter l'influence des paramètres de la recherche du minimum.

Cas 1 :

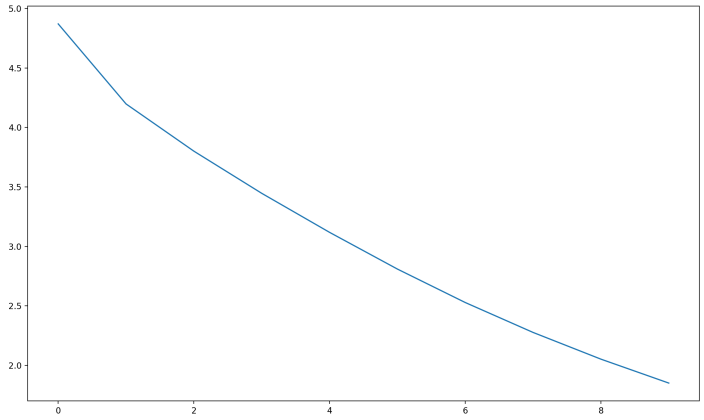
Tout d'abord, nous constatons que plus le pas est fin, plus la direction sera précise. Par conséquent, la trajectoire sera plus courte et donc le mouvement prendra moins de temps.



Évolution de l'erreur pour un pas = 0.1, ϵ = 0.001 et 100 itérations

Cas 2 :

Ensuite, un nombre d'itérations trop faible ne garantit pas de trouver la solution. On peut constater que si nous imposons NbIter=10 alors la solution n'a pas le temps de converger comme l'explicite la figure ci-dessous.



Évolution de l'erreur pour un pas=0.1, ϵ=0.001 et 10 itérations

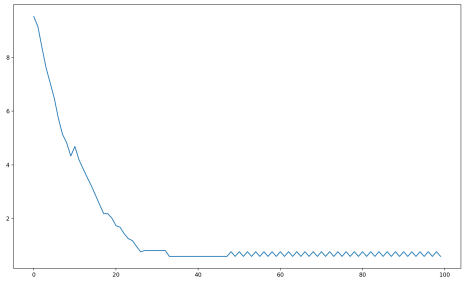
Cas 3 :

Enfin, le choix du point de départ influe sur la rapidité de la convergence. On prend, par exemple, le bras tendu comme point d'initialisation (i.e. une configuration proche d'une singularité qui fait chuter le nombre de degrés de liberté local). Cela peut se traduire par le code suivant qui place toutes les rotoïdes à un angle de 180° :

```
# Intialisation
q=np.radians([+180,+180,+180])
```

Listing 2 – Code permettant de placer la situation initiale dans une singularité

Dans ce cas, on aboutit à un erreur qui oscille autour du point minimisant $H(q)$.



Évolution de l'erreur pour une initialisation bras tendu avec un pas=0.1, ϵ=0.001 et 100 itérations

2 Méthode des gradients

L'implémentation de la méthode des gradients est réalisée grâce au code suivant :

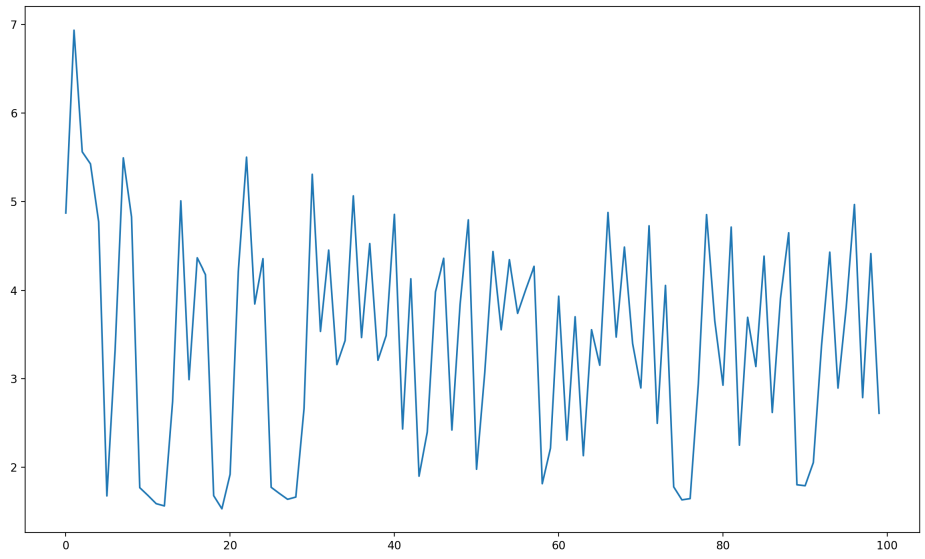
```
## Methode des gradients

print("## METHODE DES GRADIENTS")

# Intialisation
q=qinit
print('La configuration de depart est',qinit)
i=0
j=0
Hq=[]
pas=0.5 # pas fixe
eps=0.001 # espilon
Hnorm = 1000

# Saisie NnIter
NbIter=int(input('Saisir le nombres d iterations : '))

# Boucle principale
while(Hnorm>eps and i<NbIter) :
    # Jacobienne
    J = jacobienne(q)
    # Calcul de la transposee
    Jt=J.transpose()
    # Calcul de l ecart entre le Xbut et le Xc actuel
    H=(Xbut-mgd(q))
    # Iteration
    q=q+pas*np.dot(Jt,H)
    # Calcul de l erreur avec norme euclidienne
    Hnorm=np.linalg.norm(H)
    # Remplissage du vecteur avec la norme
    Hq.append(Hnorm)
```

```
32      i=i+1
33
34  # Info Boucle
35  print("La boucle a ete executee ",i,"fois")
36  # print("Xbut=", Xbut[0], "Ybut = ", Xbut[1], "Theta but (deg)= ", np.
        degrees(Xbut[2]))
37  # print("qbut=", qbut[0], "qbut = ", qbut[1], "Theta qbut (deg)= ", np.
        degrees(qbut[2]))
38
39  # Evolution Erreur
40  print("Evolution de l erreur")
41  plt.plot(Hq)
42  plt.show()
```

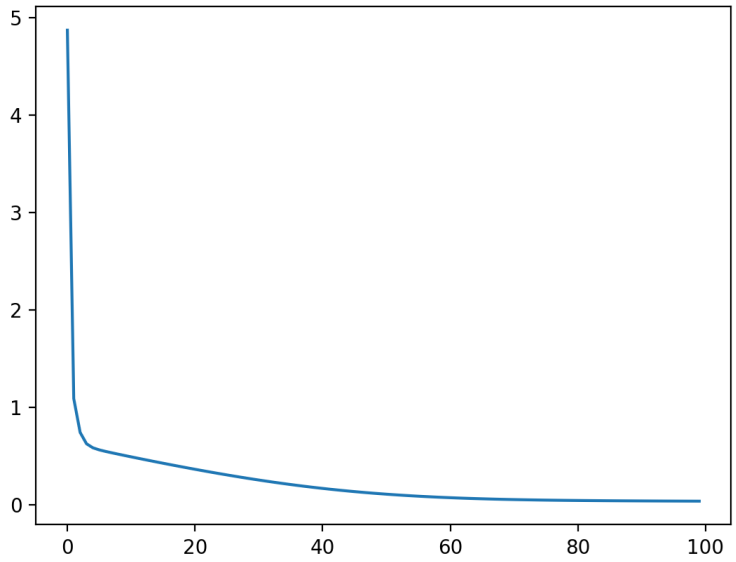
Listing 3 – Code de l'implémentation de la méthode des gradients

Dans ce code, il est demandé à l'utilisateur de saisir le nombre d'itérations souhaitées `NbIter`. Ensuite, on exécute une boucle `while()` dont la condition d'arrêt est atteinte si l'on excède le nombre d'itérations maximal ou si l'écart entre le vecteur souhaité `Xbu` et le Modèle Géométrique Direct (MGD) au point courant `q` est en dessous d'un certain seuil `eps`.
Au sein de cette boucle, on calcul la transposée de la matrice jacobienne pour la configuration courante. La configuration suivante est mise à jour par `q=q+pas*np.dot(Jt,H)`. On remplit un vecteur contenant l'écart entre les MGD courants et le but. Enfin, on compte le nombre d'itérations `i` qui peut être plus petit que `NbIter`.
Enfin, le module `numpy` permet de réaliser le tracés suivants :

Avec un `pas=0.5`, la solution ne converge pas et on obtient pour `NbIter=100` l'évolution de l'erreur suivante.



Évolution de l'erreur pour un pas de 0.5, 100 itérations $\epsilon = 0.001$
Il nous faut alors choisir un pas plus fin afin d'obtenir une meilleur précision. On s'aperçoit qu'un pas de `pas=0.2` permet une convergence de l'erreur mais cela créait toujours des oscillations.
C'est à partir de `pas=0.1` que l'on trouve une convergence de l'erreur en 100 itérations.



Évolution de l'erreur pour un pas de 0.1, 100 itérations $\epsilon = 0.001$

À noter qu'avec la méthode des gradients, l'algorithme a toujours atteint le nombre maximal d'itération ; ce qui n'était pas toujours le cas avec la méthode de Newton.

3 Utilisation de scipy.optimize

L'implémentation avec `scipy.minimize` est rendu possible par le code source suivant :

```
1
2  ## Methode des scipy.optimize
3
4  # Reinitialisation de la configuration initiale
5  q=qinit
6
7
8  # Definition de la fonction a minimiser (retourne une variable de dimension
       1)
9
10 FuncH=lambda q: np.linalg.norm((Xbut-mgd(q)))
11
12 # Construction du point de depart de l algorithme
13
14 initial_guess=(Xbut-mgd(qinit))
15
16
17 #Optimisation avec minimize
18
19 X=optimize.minimize(FuncH, qinit)
20 print("L optimum donnee par scipy est :", mgd(X.x), "\n Le but etait de",
       Xbut)
21
22
23 # Visualisation de la position du bras
24 print("Position optimale trouvee avec scipy.minimize")
25 dessinRRR(np.radians(X.x))
```

Listing 4 – Code de l'implémentation de scipy.mimize

Cette méthode nécessite de déclarer la fonction à optimiser. Ensuite, on passe en paramètre de 'optimize.minimise()' la référence à cette fonction ainsi que le point d'initialisation.

On obtient le résultat après exécution :

```
L optimum donnee par scipy est : [1.5731321883643097, 2.2071067807235756, 0.5235987697611594]
 Le but etait de [1.57313218 2.20710678 0.52359878]
Position optimale trouvee avec scipy.minimize
```

Capture du shell après exécution de scipy.optimize

4 Code Source

Vous trouverez ci-dessous le code final. Nous n'avons malheureusement pas eu le temps de terminer l'optimisation avec contraintes. L'optimisation par une approximation du gradient, quant à elle, se trouve en commentaire parce qu'elle contient encore des erreurs que nous n'avons pas pu déboguer.

```
##################
# OE TP1 MGI
# Antoine RONK
# Damien GABRIEL CALIXTE
# Toutes les d finitions de fonctions sont effectu  en local
# Un seul fichier TPMGI_GABRIEL_RONK.py a executer sur Python 3.10.6
##################

## Libraries
import matplotlib.pyplot as plt
import numpy as np
import time
from scipy import optimize

## MGD
# Calcul du MGD du robot RRR
# INPUT:  q = vecteur de configuration (radian, radian, radian)
# OUTPUT: Xc = vecteur de situation = (x,y, theta)
#              x,y en m tre et theta en radian
def mgd(qrad):
    # Param tres du robot
    l=[1,1,1]
    c1= np.cos(qrad[0])
    s1=np.sin(qrad[0])
    c12= np.cos(qrad[0]+qrad[1])
    s12=np.sin(qrad[0]+qrad[1])
    theta= qrad[0]+qrad[1]+qrad[2]
    c123=np.cos(theta)
    s123=np.sin(theta)
    x=l[0]*c1 + l[1]*c12 +l[2]*c123
    y=l[0]*s1 + l[1]*s12 +l[2]*s123
    Xd=[x,y,theta]
    return Xd

## Test de validation du MGD
# INPUT de q en degr
qdeg = [90, -90, 0]
qr = np.radians(qdeg)
Xd= mgd(qr)
print("X=", Xd[0], "Y = ", Xd[1], "Theta (deg)= ", np.degrees(Xd[2]))

## JACOBIENNE
# Calcul de J(q) du robot RRR
# INPUT:  q = vecteur de configuration (radian, radian, radian)
# OUTPUT: jacobienne(q) analytique
def jacobienne(qrad):
    # Param tres du robot
```

```
    l=[1,1,1]
    c1= np.cos(qrad[0])
    s1= np.sin(qrad[0])
    c12= np.cos(qrad[0]+qrad[1])
    s12=np.sin(qrad[0]+qrad[1])
    theta= qrad[0]+qrad[1]+qrad[2]
    theta= np.fmod(theta,2*np.pi)
    c123=np.cos(theta)
    s123=np.sin(theta)
    # Remplissage Jacobienne
    Ja=np.array([[-(l[0]*s1 + l[1]*s12 +l[2]*s123), -(l[1]*s12 +l[2]*s123),
     -(l[2]*s123)],
                 [(l[0]*c1 + l[1]*c12 +l[2]*c123), (l[1]*c12 +l[2]*c123),  (
    l[2]*c123)],
                  [1, 1, 1]])
    return Ja

##########################################################
# Afin de donner une situation atteignable pour le robot,
# vous pouvez utiliser le mgd pour d finir Xbut    partir d'une
    configuration en q
##########################################################

## qbut est donn  en degr
qbutdeg= np.asarray([-20, -90,  120])

## Calcul Xbut    partir de qbut
Xbut= np.asarray(mgd(np.radians(qbutdeg)))
# Print des resultats
print("Xbut=", Xbut[0], "Ybut = ", Xbut[1], "Theta but (deg)= ", np.degrees
    (Xbut[2]))

## Fct d'affichage 2D du robot dans le plan
def dessinRRR(q) :
    xA, yA = (0, 0)
    xB, yB = (np.cos(q[0]), np.sin(q[0]))
    xC, yC = (np.cos(q[0]) + np.cos(q[0]+q[1])), (np.sin(q[0]) + np.sin(q
    [0]+q[1]))
    xD, yD = (np.cos(q[0]) + np.cos(q[0]+q[1]) + np.cos(q[0]+q[1]+q[2])),(
    np.sin(q[0]) + np.sin(q[0]+q[1]) + np.sin(q[0]+q[1]+q[2]))
    X=[xA, xB,xC,xD]
    Y=[yA,yB,yC,yD]
    plt.plot([xA, xB], [yA, yB], color="orange", lw=10, alpha=0.5,
             marker="o", markersize=20, mfc="red")
    plt.plot([xB, xC], [yB, yC], color="orange", lw=10, alpha=0.5,
             marker="o", markersize=20, mfc="red")
    plt.plot(X,Y, color="orange", lw=10, alpha=0.5, marker="o", markersize
    =20, mfc="red")
    plt.axis('equal')
    plt.axis('off')
    plt.show()

## Affichage de qbutdeg
## Affichage de qbut
print("Affichage de la position du bras a la configuration but")
dessinRRR(np.radians(qbutdeg))



##########################################################
# Boucle principale de calcul du MGI
##########################################################

```

```
104
105
106 ## D finition de Xbut    partir de qbutdeg en degr
107 qbutdeg= np.asarray([45.,45.,-60])
108 qbut = np.radians(qbutdeg)
109 Xbut= np.asarray(mgd(qbut))
110
111
112
113 ## D finition de qinit
114 qinitdeg=np.asarray([120., 25,  45.])
115 qinit= np.radians(qinitdeg)
116 Xinit=np.asarray(mgd(qinit))
117 print("Xinit = ",Xinit)
118 print("qinit = ",qinit)
119
120 ## Methode de Newton
121
122 print("## METHODE DE NEWTON")
123
124 # Intialisation
125 q=qinit
126
127 print('La configuration de depart est',qinit)
128 i=0
129 j=0
130 Hq=[]
131 pas=0.1 # pas fixe
132 eps=0.001 # espilon
133 Hnorm = 1000
134
135 # Saisie NnIter
136 NbIter=int(input('Saisir le nombres d iterations : '))
137
138 # Boucle principale
139 while(Hnorm>eps and i<NbIter) :
140     # Jacobienne
141     J = jacobienne(q)
142     # Calcul de l'inverse
143     Jinv=np.linalg.inv(J)
144     # Calcul de l ecart entre le Xbut et le Xc actuel
145     H=(Xbut-mgd(q))
146     # Iteration
147     q=q+pas*np.dot(Jinv,H)
148     # Calcul de l erreur avec norme euclidienne
149     Hnorm=np.linalg.norm(H)
150     # Remplissage du vecteur avec la norme
151     Hq.append(Hnorm)
152     i=i+1
153
154 # Info Boucle
155 print("La boucle a ete executee ",i,"fois")
156 # print("Xbut=", Xbut[0], "Ybut = ", Xbut[1], "Theta but (deg)= ", np.
    degrees(Xbut[2]))
157 # print("qbut=", qbut[0], "qbut = ", qbut[1], "Theta qbut (deg)= ", np.
    degrees(qbut[2]))
158
159 # Evolution Erreur
160 print("Evolution de l erreur")
161 plt.plot(Hq)
162 plt.show()
163
164 # Visualisation de la position du bras
```

```
print("Position optimale trouvee avec Newton")
dessinRRR(np.radians(q))

## Methode des gradients

print("## METHODE DES GRADIENTS")

# Intialisation
q=qinit
print('La configuration de depart est',qinit)
i=0
j=0
Hq=[]
pas=0.001 # pas fixe
eps=0.001 # espilon
Hnorm = 1000

# Saisie NnIter
NbIter=int(input('Saisir le nombres d iterations : '))

# Boucle principale
while(Hnorm>eps and i<NbIter) :
    # Jacobienne
    J = jacobienne(q)
    # Calcul de la transposee
    Jt=J.transpose()
    # Calcul de l ecart entre le Xbut et le Xc actuel
    H=(Xbut-mgd(q))
    # Iteration
    q=q+pas*np.dot(Jt,H)
    # Calcul de l erreur avec norme euclidienne
    Hnorm=np.linalg.norm(H)
    # Remplissage du vecteur avec la norme
    Hq.append(Hnorm)
    i=i+1

# Info Boucle
print("La boucle a ete executee ",i,"fois")
# print("Xbut=", Xbut[0], "Ybut = ", Xbut[1], "Theta but (deg)= ", np.
    degrees(Xbut[2]))
# print("qbut=", qbut[0], "qbut = ", qbut[1], "Theta qbut (deg)= ", np.
    degrees(qbut[2]))

# Evolution Erreur
print("Evolution de l erreur")
plt.plot(Hq)
plt.show()


# Visualisation de la position du bras
print("Position optimale trouvee avec la methode des gradients")
dessinRRR(np.radians(q))

## Methode des scipy.optimize

# Reinitialisation de la configuration initiale
q=qinit


# Definition de la fonction a minimiser (retourne une variable de dimension
    1)

FuncH=lambda q: np.linalg.norm((Xbut-mgd(q)))
```

```
225
226 # Construction du point de depart de l algorithme
227
228 initial_guess=(Xbut-mgd(qinit))
229
230
231 #Optimisation avec minimize
232
233 X=optimize.minimize(FuncH, qinit)
234 print("L optimum donnee par scipy est :", mgd(X.x), "\n Le but etait de",
        Xbut)
235
236
237 # Visualisation de la position du bras
238 print("Position optimale trouvee avec scipy.minimize")
239 dessinRRR(np.radians(X.x))
240
241 ## Methode optimale comprenant une approxiamtion de la jacobienne
242
243 # Initialisation de la configuration
244 q=qinit
245
246
247 # Definition de la fonction objectif (qui donne un resulat vectoriel a
        present)
248 #FuncH=lambda q: (Xbut-mgd(q))
249
250 # Ititalisation et calcul du resulat de sortie : vecteur ligne de taille 3
251 #Jh=optimize.approx_fprime(np.ones(3),FuncH)
252
253 #print(Jh)
254
255 # Optimisation avec la jacobienne approximee
256 #X_japp=optimize.(FuncH,qinit,jac=Jh)
```

Listing 5 – Totalité du code Python