CALVEZ Tony CANEVET Antoine	AUTOMATIQUE Les Méthodes de Simulation	Séquence 2 ENSTA Bretagne

Table des matières

AUTOMATIQUE - Les Méthodes de Simulation 1

1- Introduction 3

2- La Méthode d'Euler 3

a- Méthode 3

b- Les Paramètres 4

c- Le Calcul 4

d- Les Erreurs 5

e- Conclusion de la méthode d'Euler 5

3- La Méthode de Runge-Kutta 6

a- Méthode 6

b- Les Paramètres 7

c- Le Calcul 7

d- Les Erreurs 8

e- Conclusion de la méthode de Runge-Kutta 8

4- Exercice 1 – LE PENDULE 9

Résolution : 9

Simulation : 9

Conclusion : 9

5- Exercice 2 – VAN DER POL 10

Résolution : 10

Simulation : 10

Conclusion : 10

6- Exercice 3 – Conduite automatique d'une voiture 11

Simulation : 11

Pour aller un peu plus loin : 11

Conclusion : 11

7- Conclusion 12

ANNEXE 1 : EXERCICE 1 – Simulation 13
ANNEXE 2 : EXERCICE 1 – Programme Python 14
ANNEXE 3 : EXERCICE 2 – Simulation 16
ANNEXE 4 : EXERCICE 2 – Programme Python 17
ANNEXE 5 : EXERCICE 3 – Simulation 19
ANNEXE 6 : EXERCICE 3 – Accumulation des erreurs avec la méthode d'Euler 21
ANNEXE 7 : EXERCICE 3 – Programme Python 22
ANNEXE - Sources : 24

1- Introduction

Dans le cadre de nos études sur les systèmes dynamiques, nous sommes amenés à résoudre des équations différentielles du premier ordre avec des conditions initiales.

Lors de la résolution, nous souhaitons mettre le résultat sous la forme suivante :

$$\begin{cases} \dfrac{\mathrm{d}y}{\mathrm{d}t} &= f(t, y(t)), \quad 0 \leq t \leq T \\ y(0) &= y_0 \end{cases}$$

Nous détaillons les points suivants :

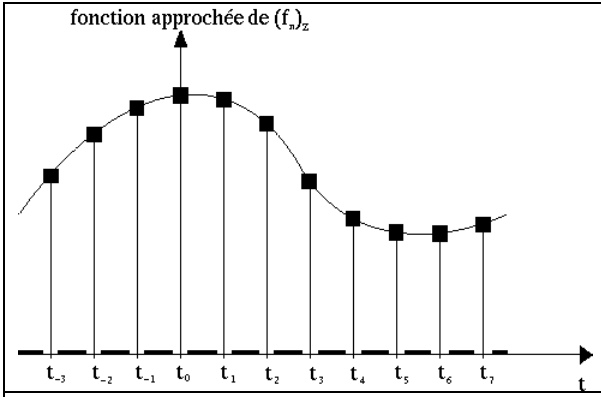
- $y(t)$: le vecteur ou scalaire recherché
- $y(0)$: la valeur initiale
- $f(t, y(t))$: une fonction aux dérivées partielles afin d'obtenir une solution

2- La Méthode d'Euler

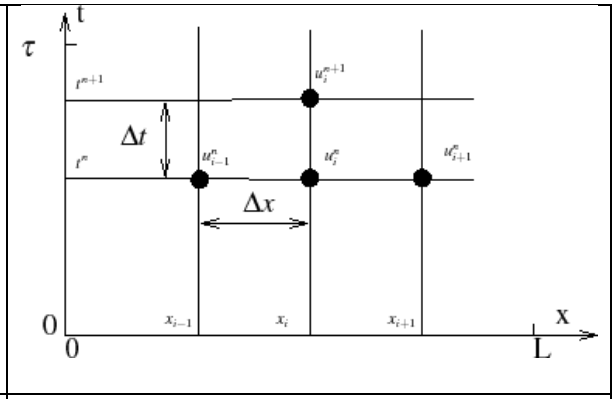
a- Méthode

Nous pouvons tenter de résoudre notre équation différentielle à partir d'une discrétisation du signal de notre système dynamique linéaire.

A partir de la discrétisation de la variable $y(t)$ avec un pas dt, nous sommes en capacité d'approximer la dérivée de notre équation grâce aux équations ci-dessous.



Discrétisation d'un Signal



Différences Finies

Nous obtenons la formule d'intégration suivante :

$$\int_{t_n}^{t_{n+1}} f(t, y(t))\,\mathrm{d}t \simeq h \times f(t_{n+1}, y(t_{n+1}))$$

Et le résultat sera sous la forme de :

$$\begin{cases} y_{n+1} &= y_n + hf(t_n, y_{n+1}) \\ y_0 &= y(0) \end{cases}$$

Nous pouvons calculer notre pas de discrétisation avec la formule suivante :

$$h = \frac{(xi + 1) - x0}{n}$$

b- Les Paramètres

Pour élaborer une simulation, nous aurons besoin des paramètres suivants :

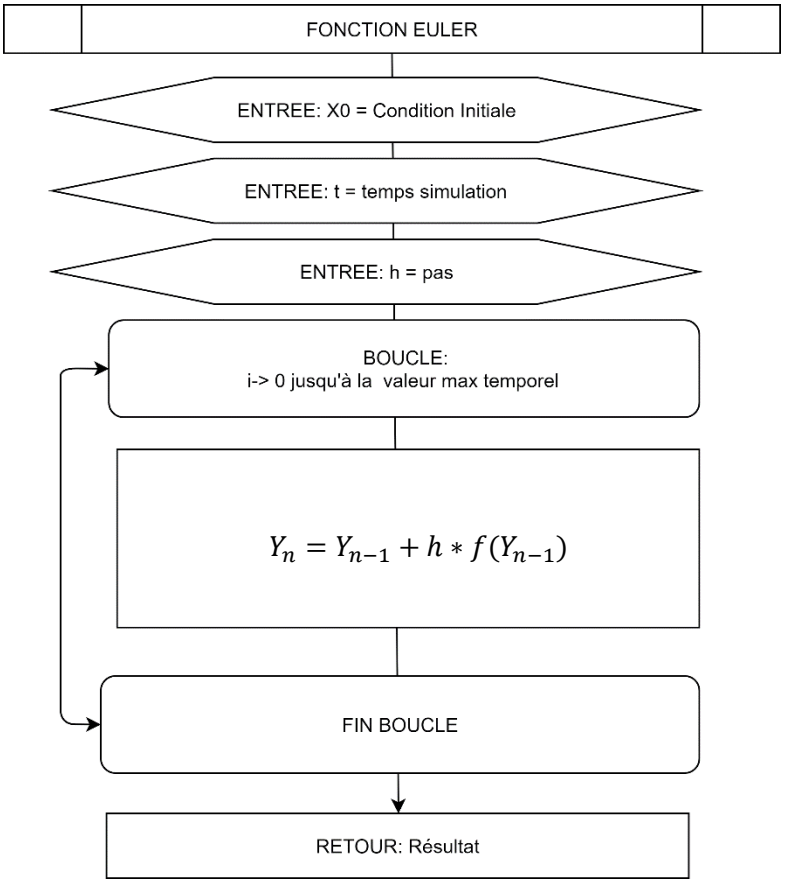
- Fonction Initiale de y'(t) : $f(t, y(t))$
- Conditions Initiales $y(o)$
- Temps (t) – Durée de la simulation
- Pas de discrétisation (h) ou nombre de point avec $h = (tmax - tmin)/n$

c- Le Calcul

Durant cette phase, nous allons créer une fonction ayant pour principe de retourner la dérivée de la fonction (y'(t)), puis discrétiser et stocker les valeurs avec la méthode d'Euler.

L'algorithme est vraiment très rapide à réaliser et très simple à mettre en œuvre.

Organigramme de programmation :



Programme Python :

```
#  METHODE D 'EULER
def methode_euler(t, X0, h):
   #CONDITION INITIALES
   u = 0
   X = np.zeros((len(X0), len(t)), 'float')
   X[: , 0] = X0
   for i in range(0, len(t) - 1):
       X[: , i + 1] = X[: , i] + h * f(X[: , i], u)
return X
```

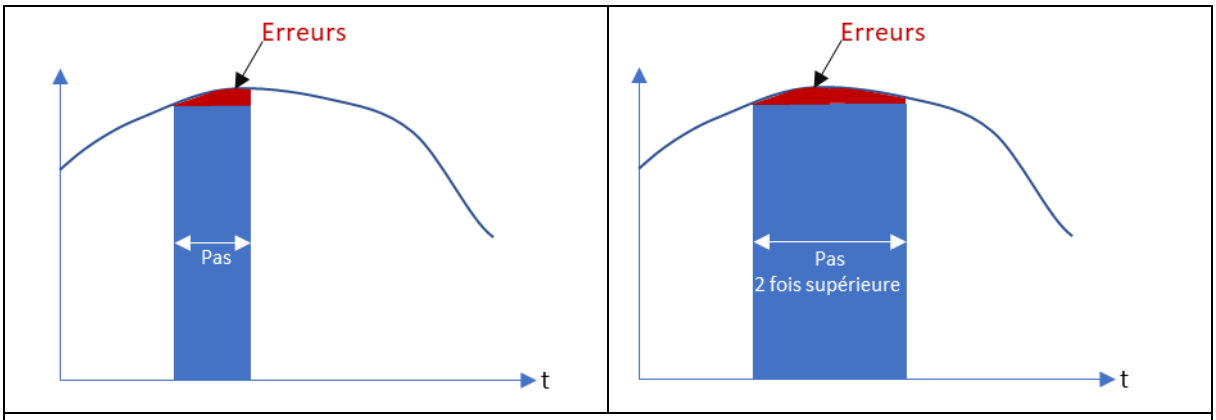
La point à l'instant t+1 est égal à la valeur du point précédent plus le pas multiplié par la dérivée de la fonction au point précédent.

d- Les Erreurs

Nous réalisons une somme d'équations dérivables pour obtenir les solutions. Dans notre cas, les pas de discrétisations (faibles) sont primordiaux pour réduire les erreurs.

Toutefois, quand il y a plus de point de discrétisation, plus il faut du temps pour calculer les coordonnées de chaque point par le programme et surtout beaucoup de mémoire pour stocker les valeurs.

Le système de boucle génère des petites erreurs qui s'accumulent tout le long de l'exécution et entraine une amplification de l'erreur (erreur globale) après plusieurs itérations et donc un résultat qui peut-être erroné.



Nous pouvons observer sur les figures ci-dessus, plus le pas est réduit, plus l'erreur sera faible.

e- Conclusion de la méthode d'Euler

Nous pouvons conclure que la méthode d'Euler répond parfaitement pour faire une approche numérique simple et permet d'avoir une réponse approximative rapidement.

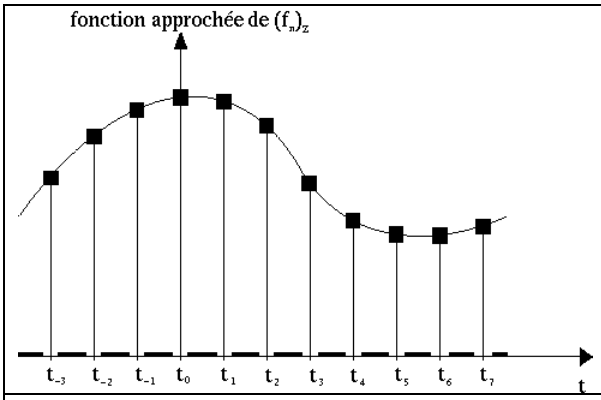
Toutefois, l'algorithme produit des erreurs et peut facilement les amplifier. En effet, l'algorithme part du dernier point connu et à partir de la pente de ce point, il détermine un nouveau point dans le prolongement de la pente. Après plusieurs itérations, l'erreur globale est non négligeable. Et pour les systèmes complexes, il existe de meilleures méthodes pour analyser un signal par exemple le Runge-Kutta.

3- La Méthode de Runge-Kutta

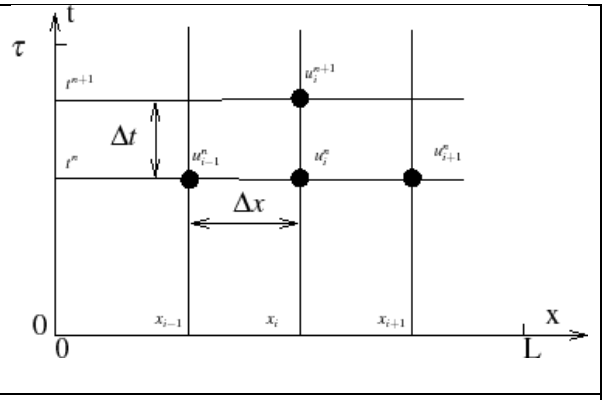
a- Méthode

Nous pouvons tenter de résoudre notre équation différentielle à partir de la méthode Runge-Kutta 2. Nous résolvons à partir d'une intégrale avec la technique des trapèzes au lieu d'avoir uniquement des rectangles (méthode d'Euler).

A partir de la discrétisation de la variable (t) en $y(t)$ et des différences finies, nous sommes en capacité d'approximer la dérivée de notre équation.



Discrétisation d'un Signal



Différences Finies

Nous obtenons la formule d'intégration suivante :

$$\int_{t_n}^{t_{n+1}} f(t, y(t))\,\mathrm{d}t \simeq \frac{h}{2}\left[(f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1}))\right]$$

Et le résultat sera sous la forme de :

$$y_{n+1} = y_n + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right) \quad \text{avec} \quad \begin{cases} k_1 = f(t_n, y_n) \\ k_2 = f(t_n + h, y_n + hk_1) \\ y_0 = y(0) \end{cases}$$

Nous déduisons que la méthode de Runge-Kutta est utilisée pour calculer implicitement : y_{n+1} tout en introduisant une meilleure valeur approchée. Le calcul est cependant plus complexe.

b- Les Paramètres

Pour élaborer une simulation, nous aurons besoin des paramètres suivants (identiques à Euler) :

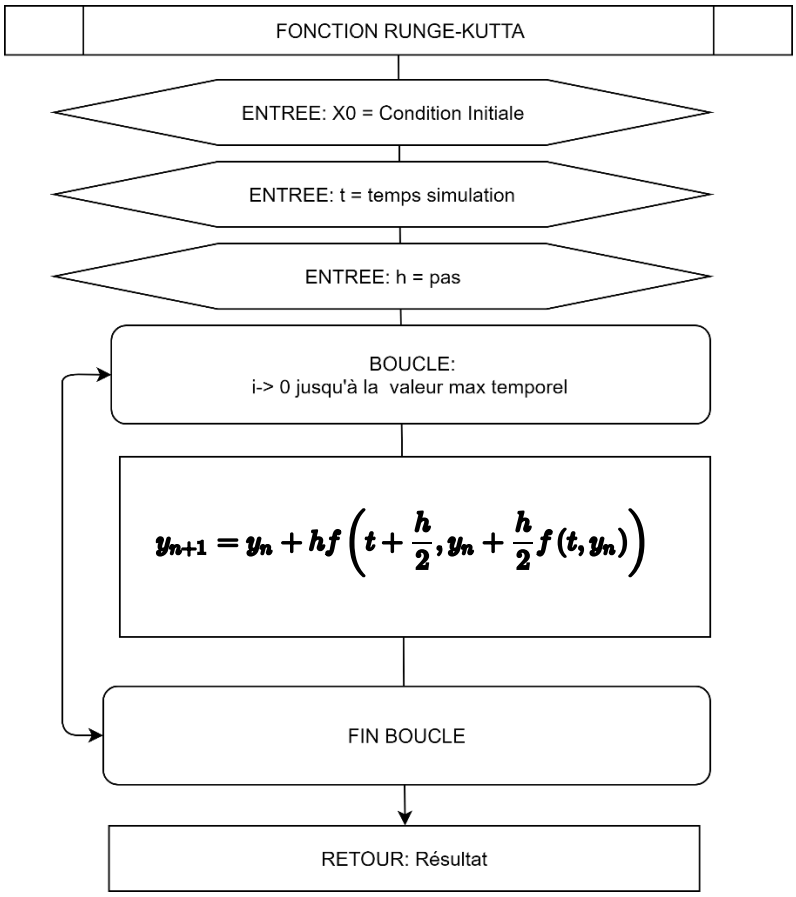
- Fonction Initiale $f(t, y(t))$
- Conditions Initiales $y(o)$
- Temps (t) – Durée de la simulation
- Pas de discrétisation (h)

c- Le Calcul

Durant cette phase, nous allons créer une fonction retournant la fonction dérivée, puis discrétiser et stocker les valeurs avec la méthode de Runge-Kutta.

L'algorithme reste simple à mettre en œuvre.

Organigramme de programmation :



Programme Python :

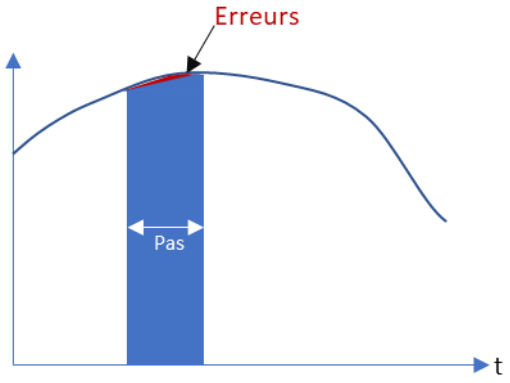
```
1.  #  METHODE RK2 - Runge Kutta
2.  def methode_RK(t, X0, h):
3.      #CONDITION INITIALES
4.      u = 0
5.      X = np.zeros((len(X0), len(t)), 'float')
6.      X[: , 0] = X0
7.
8.  for i in range(0, len(t) - 1):
9.      X[: , i + 1] = X[: , i] + h * f(X[: , i] + h / 2 * f(X[: , i], u), u)

10. return X
```

Le programme estime la valeur de $X[\,t\,+\,1]$ à partir du point $x[\,t]$ ainsi que de la dérivée de y au point $x[\,t\,+\,h/2 * f(\,t)\,]$.

d- Les Erreurs

Contrairement à la méthode d'Euler, la marge d'erreur est bien plus faible.

Ainsi, ce modèle permet d'affiner le résultat et donne un résultat plus proche de la réalité. Nous obtenons donc maintenant un modèle plus stable pour nos simulations.



Toutefois, nous travaillons encore sur un modèle d'ordre 2 qui apporte ses limites avec ses 2 erreurs de troncatures :

- Aire du trapèze
- Propagation d'erreur par l'estimation de y_{n+1} par $y_n + hf(t_n, y_n)$

e- Conclusion de la méthode de Runge-Kutta

Nous pouvons conclure que la méthode de Runge-Kutta apporte moins d'erreur au résultat et donne de la une stabilité aux systèmes dynamiques. Personnellement, nous trouvons que c'est un bon compromis entre la précision et la complexité des calculs.

Si la simulation doit être encore plus proche de la réalité, le modèle de Runge-Kutta 4 sera parfait. En effet, au lieu d'utiliser des trapèzes, la méthode RK4 est basée sur la méthode de Simpson.

4- Exercice 1 – LE PENDULE

Résolution :

Nous travaillons à partir de l'équation : $ml^2 * \ddot{\theta} = -mgl * \sin(\theta) - p\dot{\theta} + u$

a. <u>Le vecteur d'état :</u>

$X = \begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix}$ soit la dérivé : $\dot{X} = \begin{bmatrix}\dot{\theta} \\ \ddot{\theta}\end{bmatrix}$ par identification : $X = \begin{bmatrix}x_1 \\ x_2\end{bmatrix}$ et $\dot{X} = \begin{bmatrix}\dot{x_1} \\ \dot{x_2}\end{bmatrix}$

b. <u>L'entrée :</u>

$U = u$

c. <u>Equation d'Etats :</u>

Equation d'Evolution : $\dot{X} = f(x, u)$

Ainsi : $\dot{X} = \begin{bmatrix}\dot{x_1} \\ \dot{x_2}\end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{mgl}{ml} * \sin(x_1) - \frac{p}{ml^2}x_2 - \frac{1}{ml^2}u \end{bmatrix}$ dans notre programme : $\begin{cases} x_1 = X[0] \\ x_2 = X[1] \end{cases}$

Simulation :

A partir des valeurs calculées ci-dessus, nous pouvons lancer la simulation sur notre programme et comparer les méthodes d'Euler et Runge Kutta d'ordre 2.

Vous retrouverez le résultat de la simulation en annexe 1.

Lors d'une augmentation de la valeur de mu, nous observons que la courbe issue de la méthode de Runge Kutta à l'ordre 2 reste stable tandis que les valeurs prises par la méthode d'Euler sont fausses. En revanche lorsque nous diminuons mu, la méthode d'Euler offre une plus grande précision que la méthode de Runge Kutta.

Conclusion :

Notre étude du pendule permet de visualiser les avantages et les inconvénients des deux modèles, la méthode de Runge Kutta est plus adaptée à un mu élevé tandis que celle d'Euler convient à des mu faibles. De plus, Euler est plus simple à mettre en œuvre et demande moins de ressources.

5- Exercice 2 – VAN DER POL

Résolution :

Nous travaillons à partir de l'équation : $\ddot{y} + (y^2 - 1) * \dot{y} + y = 0$

a. <u>Le vecteur d'état :</u>

$X = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}$ soit la dérivé : $\dot{X} = \begin{bmatrix} \dot{y} \\ \ddot{y} \end{bmatrix}$ par identification : $X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ et $\dot{X} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix}$

b. <u>Equation d'Etats :</u>

Equation d'Evolution : $\dot{X} = f(x, u)$

Avec $\begin{cases} \dot{x1} = x2 \\ \dot{x2} = -(x1^2 - 1) * x2 - x1 \end{cases}$

Point d'équilibre :

⇨ $\dot{X} = 0$

⇨ $\begin{cases} \dot{x1} = x2 = 0 \\ -(x1^2 - 1) * x2 - x1 = 0 \end{cases}$

⇨ $x1 = 0$

Le point d'équilibre est $(0, 0)$ donc $\overline{x1} = 0$ et $\overline{x2} = 0$.

$\dot{X} = A * (X - \bar{X}) + B * (Y - \bar{Y})$

$A = \begin{pmatrix} 0 & 1 \\ -2 * \overline{x2} - 1 & -\overline{x1}\text{^}2 \end{pmatrix}$

$A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$

Avec la fonction eig() de Numpy nous obtenons les valeurs propres suivantes : 0 et 1.

Nous obtenons deux valeurs propres distinctes mais un est égal à 1, le système est donc instable (partie réelle positive).

Simulation :

Nous avons comparé le résultat entre le système linéarisé et le système dynamique. Le résultat est sans appel, nous avons donc un système instable (cf Annexe 3). Du coup, nous ne pouvons pas avoir de solutions analytiques et l'unique solution est de l'intégrer numériquement avec un Runge-Kutta.

Conclusion :

Cet exercice nous a permis de comprendre que tous les systèmes n'ont forcément de solutions analytiques , et que nous pouvons résoudre ceci uniquement qu'à partir de l'équation exacte et l'intégrer numériquement.

6- Exercice 3 – Conduite automatique d'une voiture

Simulation :

Cette partie est consacrée à la simulation d'une voiture sans régulation.

Nos variables d'état sont les suivants : $x, y, teta, v, delta$
A partir d'une planche en annexe 6, il est possible de connaitre l'intérêt de chaque variable.

Nous établissons les équations suivantes :

a. <u>Le vecteur d'état :</u>

$$\begin{cases} x = d * \cos(x3) = x1 \\ y = d * \sin(x3) = x2 \\ teta = x3 \\ V = x4 \\ delta = x5 \end{cases}$$

b. <u>Equation d'Etats :</u>

$$\dot{X} = \begin{cases} \dot{x1} = x4 * \cos(x3 + x5) \\ \dot{x2} = x4 * \sin(x3 + x5) \\ \dot{x3} = 0 \\ \dot{x4} = 0 \\ \dot{x5} = 0 \end{cases}$$

Dans le cadre de notre étude, nous imposons une des conditions constantes sur l'angle de la voiture et de ses roues. La vitesse restera, elle aussi, constante sur l'ensemble de la simulation.

Il est important de noter qu'aucune régulation n'est envisageable à ce moment de l'année et que cette voiture n'aura absolument pas de variation ou de perturbation lors de sa simulation. Mais toutefois, notre véhicule tourne parfaitement bien (annexe 5).

Pour aller un peu plus loin :

En revenant sur le sujet, nous avons pensé que la trajectoire de la voiture devait être identique lors de la simulation. Or, nous savons que la méthode d'Euler, ici employé, avait tendance à amplifier les erreurs et de les accumuler.

C'est à ce moment, que nous nous sommes posés de la question : « Si vous laissez la voiture tourner dans un temps bien supérieur à 40s, va-t-elle toujours emprunter la même trajectoire ».

Au lieu de laisser 40 secondes, nous avons fait une simulation de 15 minutes (cf : annexe 6). Nous pouvons observer que plus le temps avance, plus la voiture s'écarte de son axe d'origine.

Conclusion :

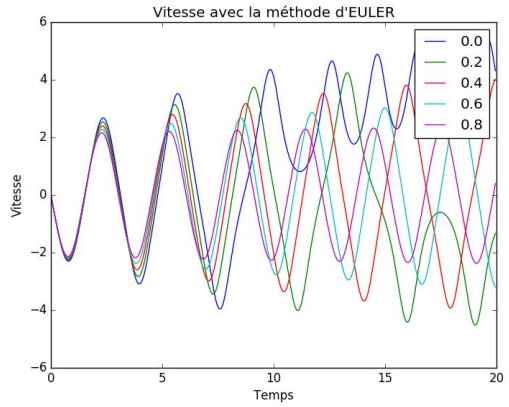
Nous avons vraiment apprécié d'appliquer de l'automatique sur un système concret.

7- Conclusion

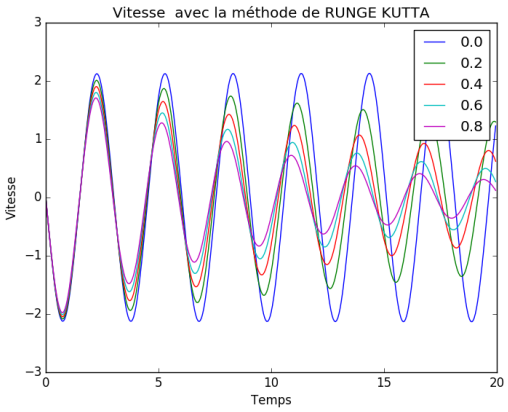
Ces séances nous ont permis d'apprendre à maîtriser les bases de l'automatique avec python.

Il était pour nous inimaginable de simuler des systèmes dynamiques et aujourd'hui, nous avons pu mettre en application les cours d'analyse numérique et d'automatique. Nous avons compris les limites qu'apporte la méthode d'Euler face au Runge-Kutta et que tous les systèmes ne sont pas forcement linéarisable.

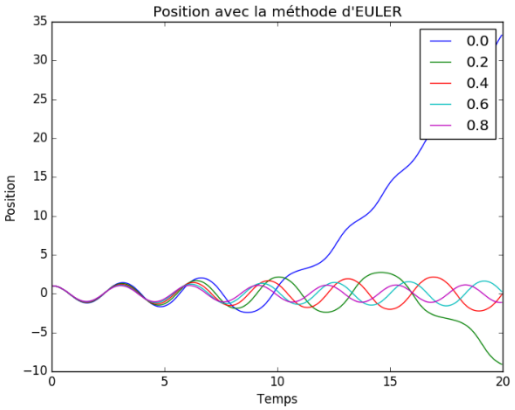
ANNEXE 1 : EXERCICE 1 – Simulation

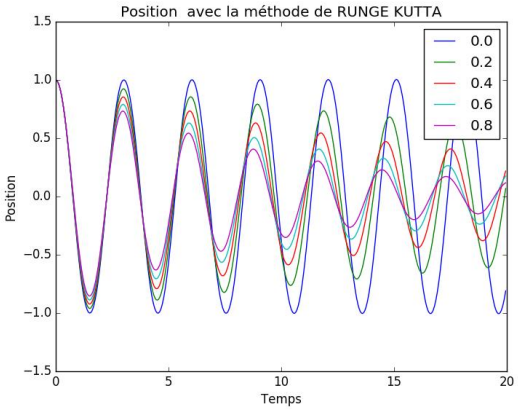


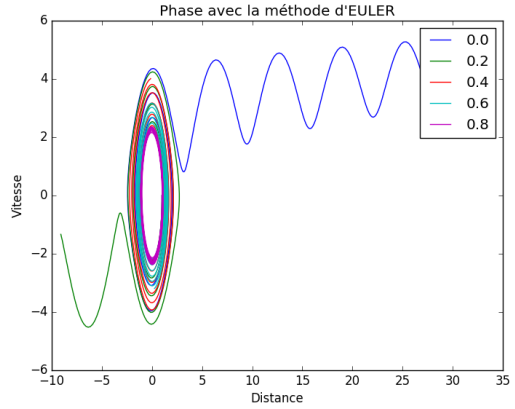
Nous pouvons constater qu'avec un mu élevé (0.8) la méthode d'Euler reste stable mais sa précision décroit au fur et à mesure que mu décroit.

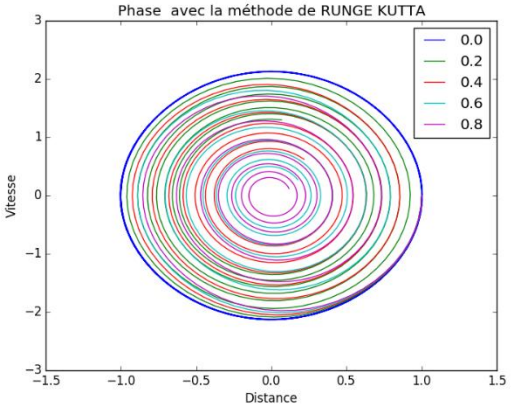


Runge Kutta est une méthode plus stable qu'Euler car elle semble rester précise pour différentes valeurs de mu. En revanche, plus le mu augmente, plus elle perd en précision.







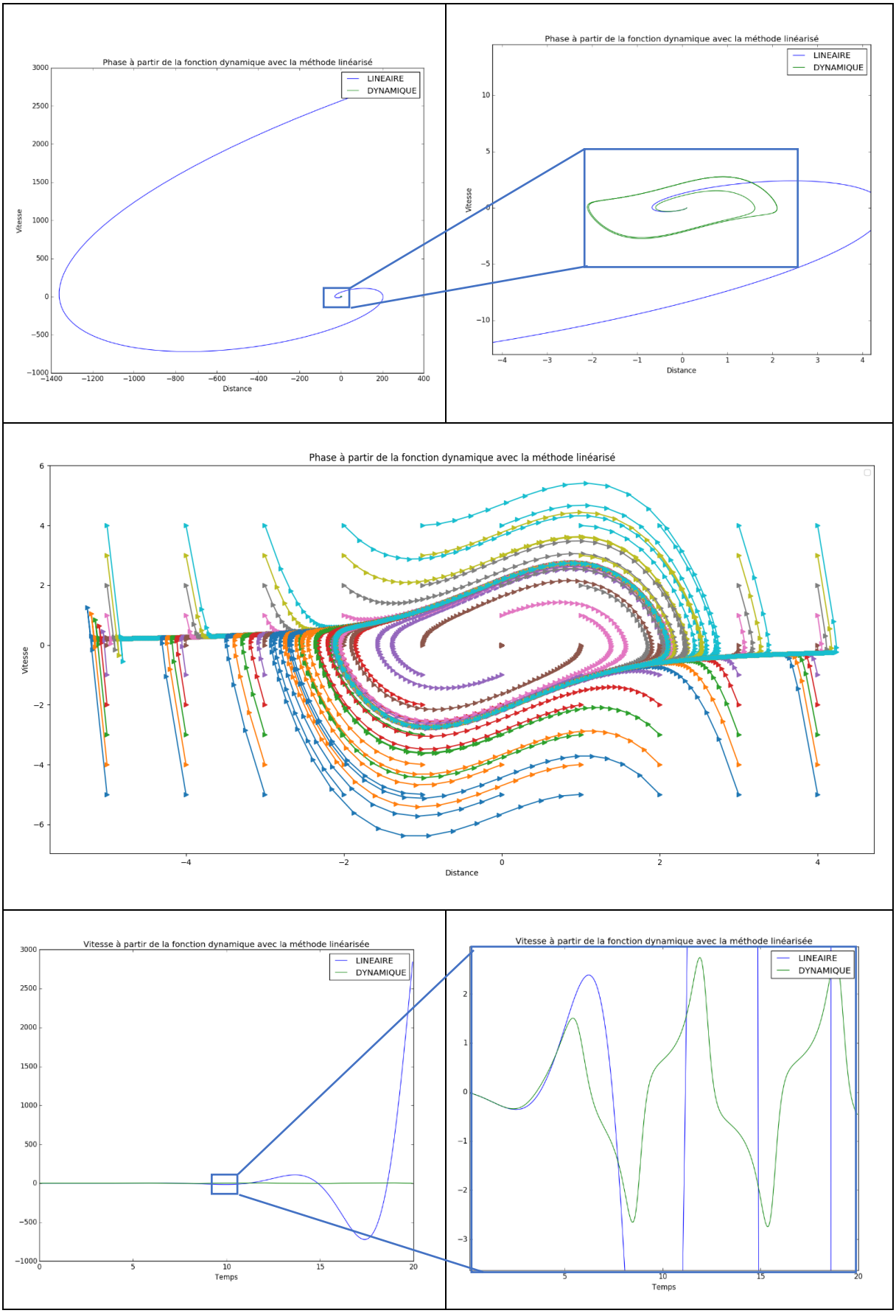


ANNEXE 2 : EXERCICE 1 – Programme Python

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.
4.   def f(x,u):
5.       m=1.0
6.       g=9.81
7.       l=2.0
8.       Xdot = [x[1], -g*np.sin( x[0] )/l - mu*x[1]/(m*l**2) + u/(m*l**2) ]
9.  return np.array(Xdot)
10.
11. # METHODE RK2 - Runge Kutta
12. def methode_RK(t, X0, h, entreetype): #CONDITION INITIALES
13. u = 0
14. X = np.zeros((len(X0), len(t)), 'float')
15. X[: , 0] = X0
16.
17. for i in range(0, len(t) - 1):
18.     X[: , i + 1] = X[: , i] + h * f(X[: , i] + h / 2 * f(X[: , i], u, entreetype), u
    , entreetype)
19.
20. return X
21.
22. def creationgraphique(X, Y, graphiquenumero, titre, labelX, labelY, nomcourbe):
23.     plt.figure(graphiquenumero)
24.     plt.plot(X, Y, label = nomcourbe)
25.     plt.title(titre)
26.     plt.xlabel(labelX)
27.     plt.ylabel(labelY)
28.     plt.legend()
29.     plt.show()
30.
31.
32. # MAIN
33. if __name__ == "__main__":
34.     print("Test Modele")
35.
36.     X0=np.array([ 1.0 , 0]) # Valeur initiale de X0
37.
38.     #PARAMETRE DE SIMULATION
39.     h = 0.05
40.     duree = 20.0
41.     t=np.arange( 0.0, duree, h)
42.     print("taille: ", len(t),"duree: ", t[-1])
43.
44.     for i in  range( 0, 10, 2 ) :
45.         mu=i/10
46.         #METHODE EULER
47.         X=methode_euler(t, X0, h)
48.         creationgraphique(t, X[1], 1, "Vitesse avec la méthode d'EULER", "Temps",
    "Vitesse", mu)
49.         creationgraphique(t, X[0], 2, "Position avec la méthode d'EULER", "Temps",
    "Position", mu)
50.         creationgraphique(X[0], X[1], 3, "Phase avec la méthode d'EULER",
    "Distance", "Vitesse", mu)
51.
52.         #METHODE RUNGE KUTTA
53.         X=methode_RK(t, X0, h)
54.         creationgraphique(t, X[1], 4, "Vitesse  avec la méthode de RUNGE KUTTA",
    "Temps", "Vitesse",mu)
55.         creationgraphique(t, X[0], 5, "Position  avec la méthode de RUNGE KUTTA",
    "Temps", "Position", mu)
```

```
56.         creationgraphique(X[0], X[1], 6, "Phase  avec la méthode de RUNGE KUTTA",
    "Distance", "Vitesse", mu)
57.         # mu élevé pour Euler et mu faible pour Runge Kuta
58.
59.     #RESULTATS
60. print(t,",",X0)
```

ANNEXE 3 : EXERCICE 2 – Simulation

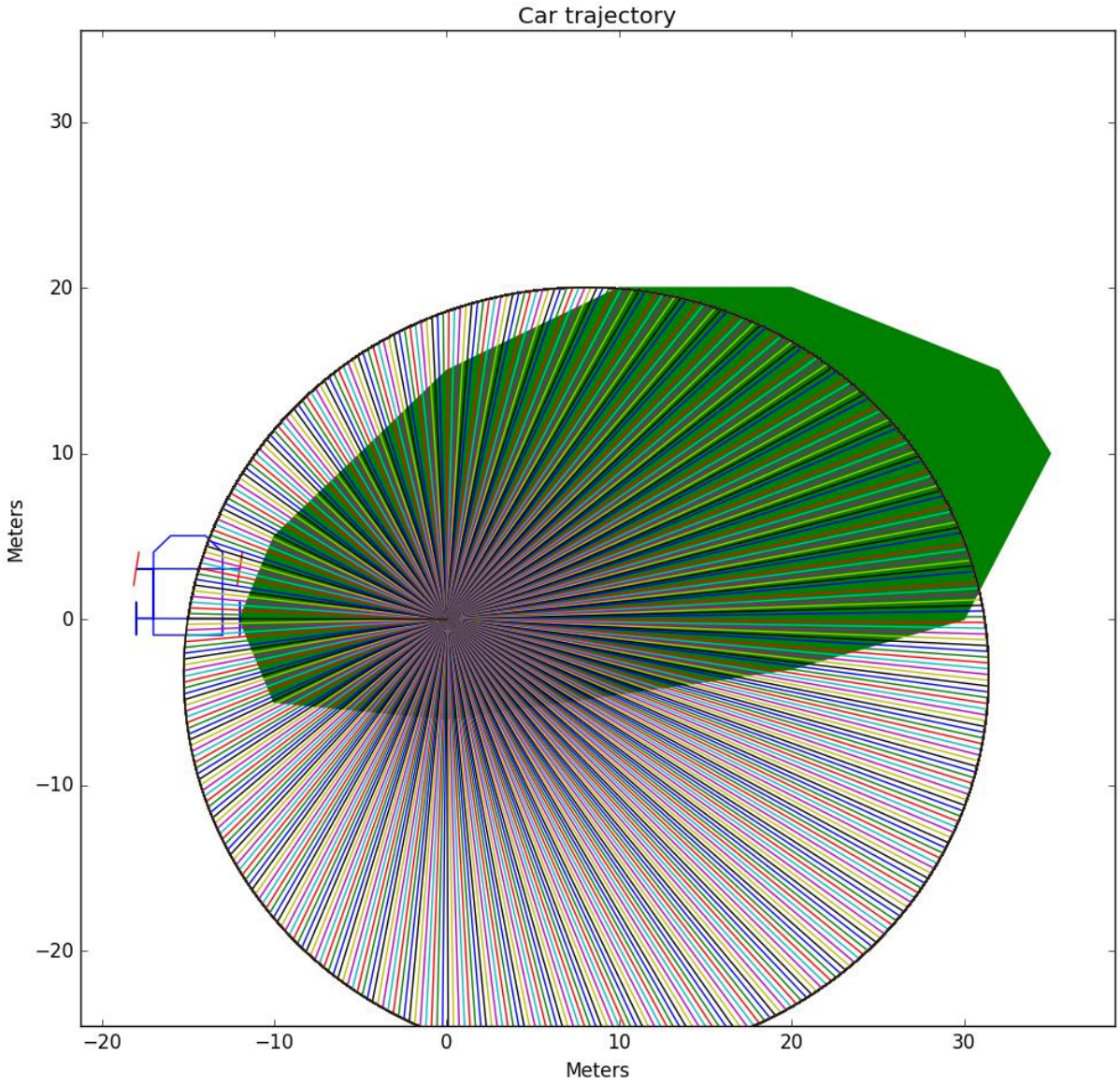


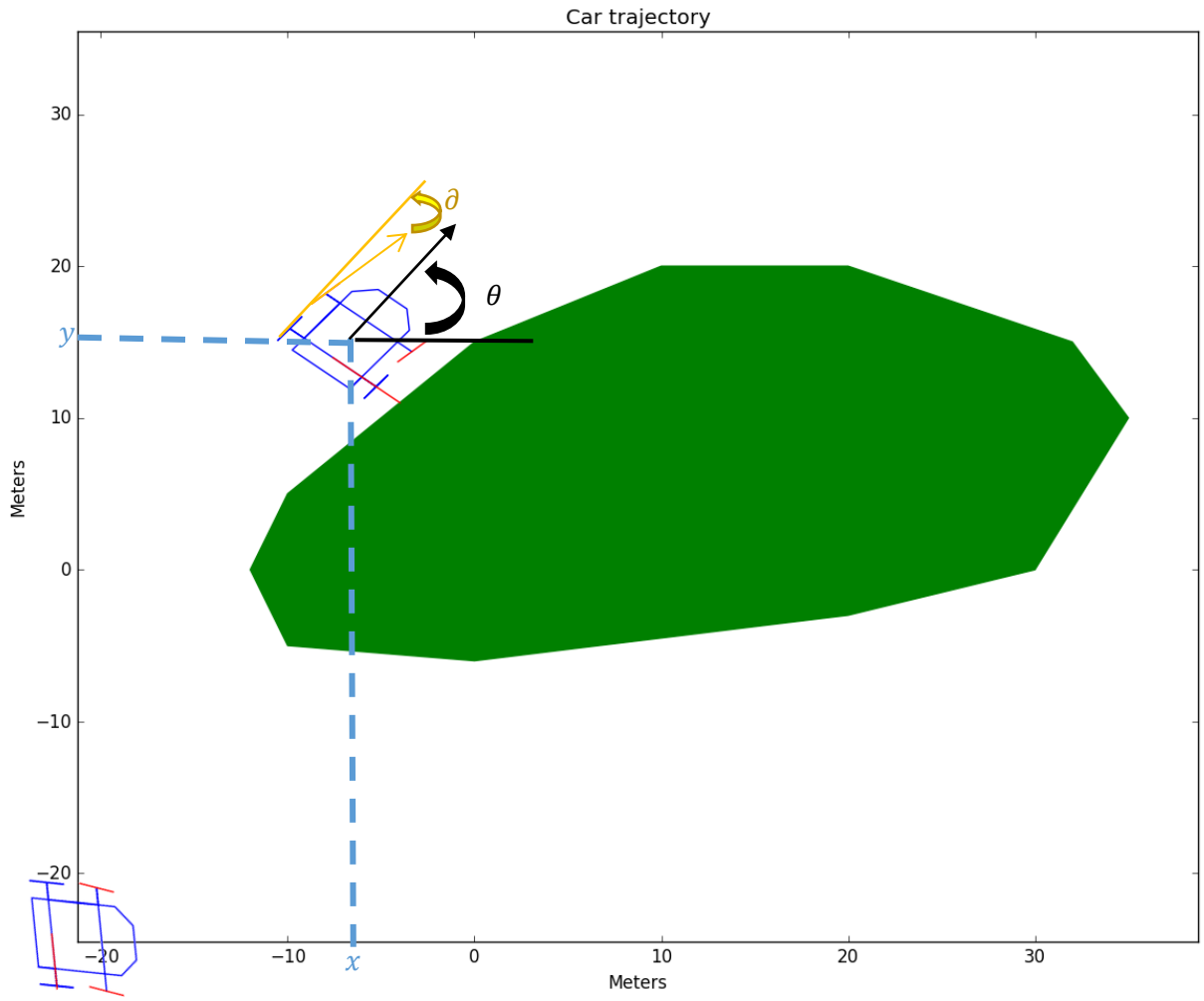
ANNEXE 4 : EXERCICE 2 – Programme Python

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.
4.
5.  # def f(x, u): #alpha = 0.1# Xdot = (x[2] + alpha * ((x[0] * * 2)- 1) * x[1] + x[0])
    # return np.array(Xdot)
6.
7.  def f(x, u, entreetype):
8.      coeff = (x[0], x[1])
9.      fonctionlinearise = np.dot(matricejacob, coeff)
10. return np.array(fonctionlinearise)
11.
12. def f2(x, u, entreetype):
13.     return -(x[0] * * 2 - 1) * x[1] - x[0]
14.
15. # METHODE RK2 - Runge Kutta
16. def methode_RK(t, X0, h, entreetype): #CONDITION INITIALES
17.     u = 0
18.     X = np.zeros(len(t), 'float')
19.     X[0] = X0[0]
20.
21.     vitesse = np.zeros(len(t), 'float')
22.     vitesse[0] = X0[1]
23.
24.     for i in range(0, len(t) - 1):
25.         vitesse[i + 1] = f([X[i], vitesse[i]] + h / 2 * f([X[i], vitesse[i]], u,
    entreetype), u, entreetype)[1]
26.
27. X[i + 1] = X[i] + h * f([X[i], vitesse[i]] + h / 2 * f([X[i], vitesse[i]], u,
    entreetype), u, entreetype)[0]
28.
29. return X, vitesse
30.
31. def methode_Euler(t, X0, h, entreetype): #CONDITION INITIALES
32.     u = 0
33.     X = np.zeros(len(t), 'float')
34.     Xdot = np.zeros(len(t), 'float')
35.
36.     X[0] = x0_0
37.     Xdot[0] = x1_0
38.
39.
40.     for i in range(0, len(t) - 1):
41.         Xdot[i + 1] = Xdot[i] + h * f2([X[i], Xdot[i]], u, entreetype)
42.         X[i + 1] = X[i] + h * (Xdot[i])
43. return X, Xdot
44.
45. def creationgraphique(X, Y, graphiquenumero, titre, labelX, labelY, nomcourbe):
46.     plt.figure(graphiquenumero)
47.     plt.plot(X, Y, label = nomcourbe)
48.     plt.title(titre)
49.     plt.xlabel(labelX)
50.     plt.ylabel(labelY)
51.     plt.legend()# plt.axis([0, 20, -500, 2500])
52.     plt.show()
53.
54.
55. # MAIN
56. if __name__ == "__main__":
57.     print("Test Modele")
58.
59.     X0 = np.array([0.1, 0])# Valeur initiale de X0
```

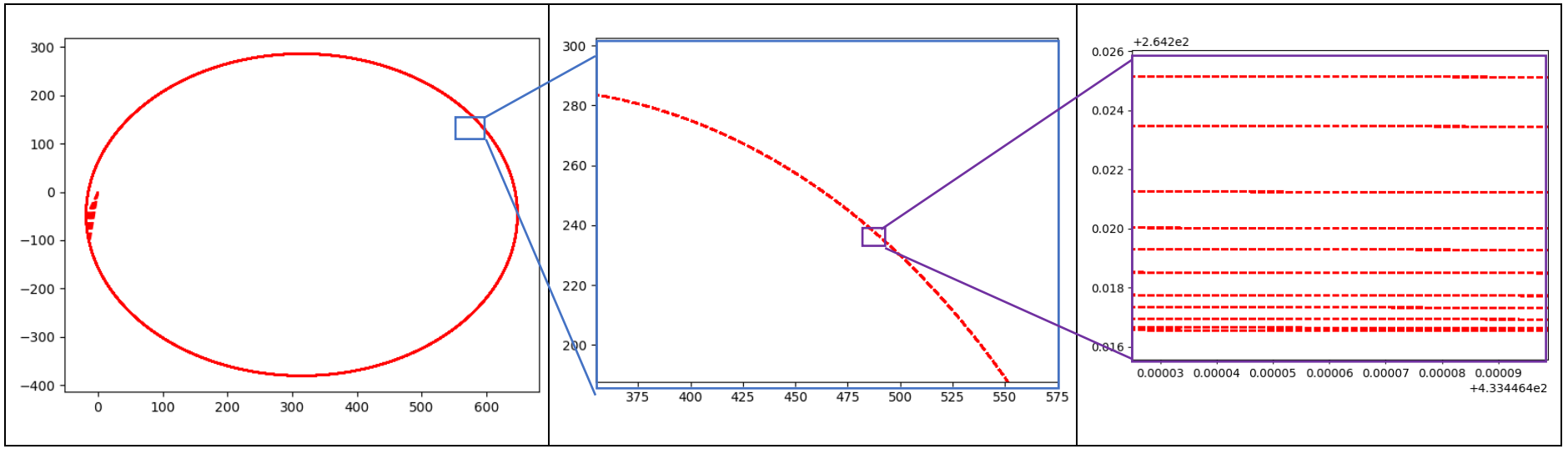
```
60.
61.     # PARAMETRE DE SIMULATION
62.     x0_0 = 0.1
63.     x1_0 = 0
64.     h = 0.05
65.     duree = 20.0
66.
67.     matricejacob = np.array([[0, 1],
68.                              [-1, 1]])
69.
70.     t = np.arange(0.0, duree, h)
71.     print("taille: ", len(t), "duree: ", t[-1])
72.
73.     for i in range(0, 10, 2): #METHODE FONCTION AVEC MATRICE JACOBIENNE
74.         Z = methode_RK(t, X0, h, "lineaire")
75. creationgraphique(t, Z[1], 1, "Vitesse à partir de la fonction linéarisée avec la mé
    thode de RUNGE KUTTA", "Temps", "Vitesse", "1")
76. creationgraphique(t, Z[0], 2, "Position à partir de la fonction linéarisée avec la m
    éthode de RUNGE KUTTA", "Temps", "Position", "2")
77. creationgraphique(Z[0], Z[1], 3, "Phase à partir de la fonction linéarisée avec la m
    éthode de RUNGE KUTTA", "Distance", "Vitesse", "3")
78.
79.     # METHODE FONCTION AVEC LA FONCTION INITIALE
80.     Y = methode_Euler(t, X0, h, "dynamique")
81. creationgraphique(t, Y[1], 1, "Vitesse à partir de la fonction dynamique avec la mét
    hode linéarisée", "Temps", "Vitesse", "1")
82. creationgraphique(t, Y[0], 2, "Position à partir de la fonction dynamique avec la mé
    thode linéarisée", "Temps", "Position", "2")
83. creationgraphique(Y[0], Y[1], 3, "Phase à partir de la fonction dynamique avec la mé
    thode linéarisé", "Distance", "Vitesse", "3")
84.
85.     # RESULTATS

86.     print(t,",",X0)
```

ANNEXE 5 : EXERCICE 3 – Simulation




Car trajectory
30
20
10
0
−10
−20
Meters
−20
−10
0
10
20
30
Meters
y
x
θ
∂

ANNEXE 6 : EXERCICE 3 – Accumulation des erreurs avec la méthode d'Euler



ANNEXE 7 : EXERCICE 3 – Programme Python

```
#- * -coding: utf - 8 - * -
import car# module pour afficher la petite voiture
import numpy as np# module pour la manipulation de matrice
import pylab as pl# pour la manipulation graphique

def car_dyn(X, U):
    return [X[3] * np.cos(X[2] + X[4]), X[3] * np.sin(X[2] + X[4]), -
0.30, u[0], u[1]]

###################### Fonction d 'observation de la voiture
def observation_func(X, track):
    X = X.flatten()
    d = car.dist_to_track(X, track)
return np.array([

######################################## methode main pour tester notre code
if __name__ == "__main__":
    print("Test du modele voiture")

    # Track(Coordonnées de la forme autour duquelle doit tourner la voiture)
    track = np.array([
    [-10, -12, -10, 0, 10, 20, 32, 35, 30, 20, 0, -10],
    [-5, 0, 5, 15, 20, 20, 15, 10, 0, -3, -6, -5]])

    # Conditions initiales pour la simulation du mobile
    X_0 = np.array([-15.0, 0.0, np.pi / 2.0, 7.0, -
0.15])# Etat initial du véhicule[x, y, theta, v, delta]
    X_0 = X_0.reshape((5, 1))# Vecteur colonne!
    U_0 = np.array([0.0, 0.0])# Consigne initiale U pour le mobile[u1, u2]
    U_0 = U_0.reshape((2, 1))# Vecteur colonne!
    Y_0 = observation_func(X_0, track)# Observation initiale du mobile à t = 0
    print('X_0 : \n', X_0, '\nU_0 : \n ', U_0, '\nY_0 : \n', Y_0)

    print("Estimation position à t=0")
    d_0 = Y_0[0]# distance du mobile / à la forme
    car.draw(X_0, d_0, track)# Pour dessiner la petite voiture et le polygone
    pl.show()

    tmin = 0.0
    tmax = 40.0
    L = 3
    d = Y_0[0]
    h = 0.05
    t = np.arange(tmin, tmax, h)
    Xc = np.zeros(len(t))
    Yc = np.zeros(len(t))
    teta = np.zeros(len(t))
    vitesse = np.zeros(len(t))
    delta = np.zeros(len(t))
    u = U_0
    Xc[0] = X_0[0]
    Yc[0] = X_0[1]
    teta[0] = X_0[2]
    vitesse[0] = X_0[3]
    delta[0] = X_0[4]

    for i in range(len(t) - 1):

        vecteur = np.array([Xc[i], Yc[i], teta[i], vitesse[i], delta[i]])
    vecteur = vecteur.reshape((5, 1))

    d = observation_func(vecteur, track)[0]
    u[0] = observation_func(vecteur, track)[1]
    u[1] = observation_func(vecteur, track)[2]
```

```
63.
64.
65. Xc[i + 1] = Xc[i] + h * car_dyn([Xc[i], Yc[i], teta[i], vitesse[i], delta[i]], u)[0]
66. Yc[i + 1] = Yc[i] + h * car_dyn([Xc[i], Yc[i], teta[i], vitesse[i], delta[i]], u)[1]
67. teta[i + 1] = teta[i] + h * car_dyn([Xc[i], Yc[i], teta[i], vitesse[i], delta[i]], u
    )[2]
68.     vitesse[i + 1] = u[0]
69.     delta[i + 1] = u[1]
70.
71.     car.draw(vecteur, d, track)# Pour dessiner la petite voiture et le polygone
72.
73.     pl.show()
```

ANNEXE - Sources :

Jimmy ROUSSEL. 2018. “Runge Kutta.” Physique. http://perso.ensc-rennes.fr. 2018. https://www.femto-physique.fr/omp/runge_kutta.php.

Jimmy ROUSSEL. 2018. “Euler.” Physique. http://perso.ensc-rennes.fr. 2018. https://www.femto-physique.fr/omp/euler.php.

Marc BUFFAT. 2008. “Différence Finies.” Physique. http://www.ufrmeca.univ-lyon1.fr/~buffat/COURS/COURSDF_HTML/node16.html

Jean-Denis DUROU. “Discrétisation.” Physique. https://www.irit.fr/~Jean-Denis.Durou/ENSEIGNEMENT/TS/COURS/co03.html