Résolution numérique d'équations différentielles linéaire du 1er ordre par la méthode d'Euler

I Capacité numérique :

Mettre en œuvre la méthode d'Euler à l'aide d'un langage de programmation pour simuler la réponse d'un système linéaire du premier ordre à une excitation de forme quelconque.

II Méthode d'Euler

II.1 Équation différentielle à second membre quelconque

Considérons le cas d'une équation différentielle linéaire faisant intervenir les dérivées temporelles d'une grandeur $x(t)$ quelconque.

On recherche numériquement, sur un intervalle de temps choisi, une approximation de la solution à une équation différentielle de la forme :

$$\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{x}{\tau} = f(t) \quad \text{vérifiant la condition initiale : } x(t_0) = x_0.$$

Sa solution est l'unique fonction $x(t)$ telle que $x(t_0) = 0$ et vérifiant cette équation différentielle, dans laquelle $f(t)$ est une fonction quelconque, **connue**, du temps.

II.2 Approximation numérique

La méthode d'Euler consiste à ;

- choisir un intervalle de temps $[t_{\min}; t_{\max}]$ sur lequel on souhaite obtenir une approximation de $x(t)$;
- le subdiviser en N intervalles (équidistants la plupart du temps) pour définir les instants t_i avec $i \in [0; N-1]$;
- rechercher des valeurs approchées de $x(t_i)$.

Pour cela, on approxime sur chaque intervalle t_i, t_{i+1} la variation de la fonction $x(t)$ inconnue par sa dérivée en t_i, donnée par l'équation différentielle, selon :

$$x(t_{i+1}) \simeq x(t_i) + \frac{\mathrm{d}x}{\mathrm{d}t}(t_i)\,(t_{i+1} - t_i)$$

La connaissance de la valeur de la condition initiale $x(t_0)$ permet alors, de calculer par récurrence, $x(t_1)$, puis $x(t_2)\ldots$

III Algorithme

III.1 Modules nécessaires

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

```
%matplotlib notebook
```

La ligne précédente ne doit apparaître que dans les notebooks `Jupyter`, pas dans un fichier python.

III.2 Fonctions définies par morceau

La fonction `piecewise` du module `numpy` permet de définir des fonctions par morceau. Elle permet en particulier de définir la fonction discontinue de Heaviside, utile pour étudier la réponse à un échelon.

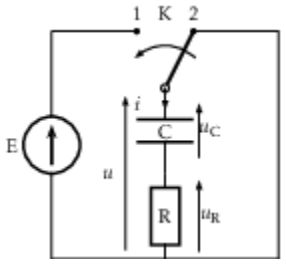
On peut l'appliquer à un tableau (cas `y1`) ou à un scalaire (cas `y2`).

```
figHeaviside,axHeaviside = plt.subplots()

x = np.linspace(-5,5,200)
y1 = np.piecewise(x,[x<2,x>=2],[2,3])

def Heaviside(x,params):
    x0, yavant, yapres = params
    return np.piecewise(x,[x<x0,x>=x0],[yavant,yapres])

params = [1,-4,6]
y2 = Heaviside(x,params)

axHeaviside.set_xlabel('abscisse')
axHeaviside.set_ylabel('ordonnée')
axHeaviside.set_title('titre')
axHeaviside.plot(x,y1,'b',label='Heaviside')
axHeaviside.plot(x,y2,'r',label='Heaviside2')
axHeaviside.legend(loc='best',shadow=True)

figHeaviside.show()
```

IV Dipôle RC série soumis à un échelon de tension

IV.1 Équation différentielle canonique adimensionnée

On étudie le cas classique d'un dipôle RC série soumis à un échelon de tension



La tension u_C est solution de l'équation :

$$\frac{\mathrm{d}u_C}{\mathrm{d}t} + \frac{u_C}{\tau} = \frac{E}{\tau} H(t),$$

avec $H(t)$ la fonction de Heaviside, $\tau = RC$. On peut étudier numériquement cette équation avec des valeurs de t\$ et u_C respectivement dimensionnées en millisecondes et en Volts.

Il est cependant possible de les **adimensionner** en définissant les nombres sans dimension :

- $U = u_C/E$;
- $t' = t/\tau$.

L'équation devient alors :

$$\frac{\mathrm{d}U}{\mathrm{d}t'} + U = H(t).$$

On souligne ainsi le caractère universel de cette équation et on ne manipule que des nombres de l'ordre de l'unité, ce qui est souvent plus facile à interpréter.

Dans l'exemple ci-dessous, on a $C = 1\mu$F, $R = 1$kΩ, $E = 5$V. Le symbole `t` désigne la variable adimensionnée t/τ et `U` la variable adimensionnée u_C/E.

```
1 NombrePoints = 2000
2
```

```
3  #  Paramètres
4  E = 5 # en V
5  R = 1e3 # en ohm
6  C = 1e-6 # en F
7
8  tau = R*C # en s
9  print(f'constante de temps: tau= {1e3*tau} ms\n valeur asymptotique: uCinf = {E} V')
10
11 tmin = -2 # en unités de tau
12 tmax = 6
13 t = np.linspace(tmin,tmax,NombrePoints)
14 Deltat = (tmax-tmin)/(NombrePoints -1)
15
16 def dxoverdt(t,x,params):
17     t0,tau,xinfty = params
18     return np.piecewise(t,[t<t0,t>=t0],[-x/tau,(xinfty-x)/tau])
19
20 x = np.zeros(NombrePoints)
21
22 params = [0,1,3]
23
24 for i in range(NombrePoints-1):
25     x[i+1] = x[i]+ Deltat*dxoverdt(t[i],x[i],params)
26
27 figequadiff,axequadiff = plt.subplots()
28
29 axequadiff.set_xlabel('t/tau')
30 axequadiff.set_ylabel('u/E')
31 axequadiff.set_title('Réponse d\'un dipôle RC à un échelon de tension')
32 axequadiff.plot(t,x,'b',label='uc/E')
33 axequadiff.legend(loc='best',shadow=True)
34
35 figequadiff.show()
```

IV.2 Module `scipy.integrate`

La méthode d'Euler est le plus simple des algorithmes de résolution d'équations numériques d'équations différentielles et ne sera pas le plus performant pour certaines équations différentielles plus délicates à manipuler que l'exemple présenté ci-dessus, en cherchant des approximations plus précises de la variation $x(t_{i+1}) - x(t_i)$, au prix d'un algorithme plus compliqué.

Le module `scipy.integrate` permet de les utiliser sans avoir à coder la boucle `for`. On présente ci-après un exemple de l'utilisation de sa fonction `solve_ivp` (pour Solve Initial Value Problem). Elle peut résoudre un système d'équations différentielles couplées. Elle prend pour arguments nécessaires :

- le tableau des dérivées des fonctions recherchées (comme dans le cas de la méthode d'Euler) en fonctions des valeurs des différentes fonctions,

- le tableau de l'intervalle de temps sur lequel rechercher la solution
- le tableau des conditions initiales,

Notons que le pas d'intégration qui était constant à `Deltat` dans la méthode d'Euler précédente sera ici variable et géré par l'algorithme.

Comme arguments optionnels, citons :

- la liste `t_eval` des instants auxquels évaluer la fonction recherchée
- des paramètres `args` pour le calcul des dérivées,
- l'algorithme `method` utilisé (voir la documentation)
- des évènements `events` particuliers (valeurs de la fonction, de sa dérivée…)
- un intervalle `max_step` imposant une taille maximale pour les intervalles de temps considérés : il est préférable de l'utiliser dans cet exemple où l'équation différentielle fait intervenir une fonction discontinue

Elle retourne :

- les instants où a été évaluée la fonction recherchée,
- ses valeurs en ces instants,
- une interpolation de la fonction recherchée, si l'option `dense_output=True` a été utilisée,

Si on a utilisé l'option `events`, elle retourne aussi les instants et valeurs de la fonction des évènements recherchés

Dans l'exemple précédent du dipôle RC, on n'utilise pas de tableau mais de simples scalaires car on ne recherche qu'une seule fonction $u_C(t\backslash)$. On utilisera en revanche un tableau pour :

- traiter un circuit à plusieurs mailles,
- traiter un circuit à une seule maille régi par une équation différentielle d'ordre strictement supérieur à 1.

```python
from scipy.integrate import solve_ivp

# paramètres de l'échelon
t0 = 0
xavant = 2
xapres = 5

def dxoverdt(t,x,t0,xavant,xapres):
    return np.piecewise(t,[t<t0,t>=t0],[xavant-x,xapres-x])

# intervalle de temps
tmin = -2 # en unités de tau
tmax = 5

# évènements à rechercher: ici passage par x=4
def passage(t,x,t0,xavant,xapres):
    return x[0]-4

# Condition initiale (à t = tmin)
CI = [0]

# Résolution numérique de l'équation différentielle
solution = solve_ivp(dxoverdt,[tmin,tmax],CI,args=(t0,xavant,xapres),events=passage,dense_output

print(f'passage à {solution.y_events} en t={solution.t_events}')

figSolveIVP,axSolveIVP = plt.subplots()

axSolveIVP.set_xlabel('t/tau')
axSolveIVP.set_ylabel('u/E')
axSolveIVP.set_title('Résolution par SolveIVP')
axSolveIVP.plot(solution.t,solution.y[0],'b+',label='Valeurs évaluées')

NombrePoints = 200
instants = np.linspace(tmin,tmax,NombrePoints)
valeurs = solution.sol(instants)
axSolveIVP.plot(instants,valeurs.T,'r',label='Fonction interpolée')
axSolveIVP.legend(loc='best',shadow=True)

figSolveIVP.show()
```

V Questions du DM03

V.1 III.1.a

On a maintenant :

```python
NombrePoints = 2000

```

```
#  Paramètres
E = 10 # en V
R = 500 # en ohm
L = 2.9e-2 # en H

uR = E # en V
tau = L/R # en s
print(f'constante de temps: tau= {1e3*tau} ms\n valeur asymptotique: uR = {uR} V')

tmin = -.2 # en unités de tau
tmax = 6
t = np.linspace(tmin,tmax,NombrePoints)
Deltat = (tmax-tmin)/(NombrePoints -1)

def dxoverdt(t,x,params):
    t0,tau,xinfty = params
    return np.piecewise(t,[t<t0,t>=t0],[-x/tau,(xinfty-x)/tau])

x = np.zeros(NombrePoints)

params = [0,1,E]

for i in range(NombrePoints-1):
    x[i+1] = x[i]+ Deltat*dxoverdt(t[i],x[i],params)

figequadiff,axequadiff = plt.subplots()

axequadiff.set_xlabel('t/tau')
axequadiff.set_ylabel('uR (V)')
axequadiff.set_title('Réponse d\'un dipôle RL à un échelon de tension')
axequadiff.plot(t,x,'b',label='uR')
axequadiff.legend(loc='best',shadow=True)

figequadiff.show()
```

V.2 III.1.b

On trace plus précisément la courbe entre τ et 1.3τ à l'aide de l'option `set_xlim`, on rajoute une grille pour guider l'œil avec l'option `grid`, qu'on affine avec `[x/y]axis.set_minor_locat`

```
from matplotlib.ticker import (MultipleLocator, AutoMinorLocator)
figzoom,axezoom = plt.subplots()
axezoom.xaxis.set_minor_locator(MultipleLocator(0.01))
axezoom.yaxis.set_minor_locator(MultipleLocator(0.005))

axezoom.set_xlabel('t/tau')
axezoom.set_ylabel('uR (V)')
axezoom.set_title('Réponse d\'un dipôle RL à un échelon de tension')
axezoom.plot(t,x,'b',label='uR')
axezoom.legend(loc='best',shadow=True)
axezoom.set_xlim(.9,.93)
axezoom.set_ylim(5.8,6.2)
axezoom.grid(which='both')

figzoom.show()
```

On y vérifie que $u_R = 6\text{V}$ pour $t = 9.16e-1\tau = 5.3e-5\text{s}$.

V.3 III.2.a

Ici on n'adimensionne pas l'équation car on doit comparer deux échelles de temps, τ et la période de commutation. On définit la fonction `dxoverdtCommutation` qui retourne :

$$-\frac{i}{\tau} \pm \frac{i_\infty}{\tau}.$$

selon la demi-période dans laquelle elle se trouve. Pour ce faire, on utilise l'opérateur `//` qui donne le quotient entier et l'opérateur `%` qui donne le reste (de la division par 2 ici pour tester la parité).

Le reste du code est similaire au précédent à ceci près que cette fonction dépend de deux paramètres temporels : la période de commutation et la constante de temps du dipôle RL.

```
def dxoverdtCommutation(t,x,params):
    periode, tau, ybas, yhaut = params
    demiperiode = periode/2
    parite = (np.floor(t//demiperiode))%2 # vaut 0 ou 1 selon la 1/2 période dans laquelle on est
    return (np.piecewise(t,[parite != 0, parite == 0 ],[ybas,yhaut]) -x)/tau

iLinf = 1000*E/R # en mA
tau = 1000*L/R # en ms
tmin = 0
periode = 1/5 # en ms
NombrePeriodes = 10
tmax = NombrePeriodes * periode
t = np.linspace(tmin,tmax,NombrePoints)
Deltat = (tmax-tmin)/(NombrePoints -1)
params = [periode,tau,-iLinf,iLinf] # t ms
x = np.zeros(NombrePoints)

for i in range(NombrePoints-1):
    x[i+1] = x[i]+ Deltat*dxoverdtCommutation(t[i],x[i],params)

figCommutation,axCommutation = plt.subplots()

axCommutation.set_xlabel('t (ms)')
axCommutation.set_ylabel('iL (mA)')
axCommutation.set_title('Réponse d\'un dipôle RL à un créneau de tension')
axCommutation.plot(t,x,'b',label='iL')
```

```
axCommutation.legend(loc='best',shadow=True)

figCommutation.show()
```

On sélectionne ensuite des tranches (”slices” en python) du tableau `x` correspondant à chaque demi-période et on en prend les valeurs moyenne, minimale et maximale. On peut le faire en calculant combien de points compte chaque tranche comme dans l'exemple suivant :

```
NombrePointsDemiPeriode = int((periode/(2*Deltat)))+1

iLmoyenneA = np.zeros(NombrePeriodes) # 1/2 périodes impaires
iLminA = np.zeros(NombrePeriodes)
iLmaxA = np.zeros(NombrePeriodes)

for i in range(NombrePeriodes):
    iLmoyenneA[i] = np.average(x[2*i*NombrePointsDemiPeriode:(2*i+1)*NombrePointsDemiPeriode])
    iLminA[i] = np.min(x[2*i*NombrePointsDemiPeriode:(2*i+1)*NombrePointsDemiPeriode])
    iLmaxA[i] = np.max(x[2*i*NombrePointsDemiPeriode:(2*i+1)*NombrePointsDemiPeriode])

print('1/2 périodes impaires')
print(f'valeurs moyennes de iL (en mA): {iLmoyenneA}\nvaleurs max: {iLmaxA}\nvaleurs min {iLminA}')


iLmoyenneB = np.zeros(NombrePeriodes) # 1/2 périodes paires
iLminB = np.zeros(NombrePeriodes)
iLmaxB = np.zeros(NombrePeriodes)

for i in range(NombrePeriodes):
    iLmoyenneB[i] = np.average(x[(2*i+1)*NombrePointsDemiPeriode:(2*i+2)*NombrePointsDemiPeriode])
    iLminB[i] = np.min(x[(2*i+1)*NombrePointsDemiPeriode:(2*i+2)*NombrePointsDemiPeriode])
    iLmaxB[i] = np.max(x[(2*i+1)*NombrePointsDemiPeriode:(2*i+2)*NombrePointsDemiPeriode])

print('1/2 périodes paires')
print(f'valeurs moyennes de iL (en mA): {iLmoyenneB}\nvaleurs max: {iLmaxB}\nvaleurs min {iLminB}')
```

La détermination des indices à sélectionner peut cependant être délicate si la durée `Deltat` et la demi-période de commutation ne sont pas commensurables. On préférera utiliser les possibilités plus avancées de sélection selon :

```
iLmoyenneC = np.zeros(NombrePeriodes)
iLminC = np.zeros(NombrePeriodes)
iLmaxC = np.zeros(NombrePeriodes)
for i in range(NombrePeriodes):
    xTemp = [x[j] for j in range(len(x)) if t[j] >= i*periode and t[j] < (i+1/2) * periode]
    iLmoyenneC[i] = np.average(xTemp)
    iLminC[i] = np.min(xTemp)
    iLmaxC[i] = np.max(xTemp)

print('1/2 periodes impaires')
print(f'valeurs moyennes de iL (en mA): {iLmoyenneC}\nvaleurs max: {iLmaxC}\nvaleurs min {iLminC}')
```

V.4 III.2.b

On définit la fonction en «dents de scie» à utiliser dans l'équation différentielle, en utilisant le reste `t % demiperiode` et la parité `(t//demiperiode)%2` pour distinguer les demiperiodes paire ou impaire.

```
def dxoverdtTriangle(t,x,params):
  periode, tau, ybas, yhaut = params
  demiperiode = periode/2
  reste = t % demiperiode
  pente = (yhaut-ybas)/demiperiode
  parite = (t//demiperiode)%2 # vaut 0 ou 1 selon la 1/2 période dans laquelle on est
  return  (np.piecewise(t,[(t//demiperiode)%2 != 0, (t//demiperiode)%2 == 0 ],[lambda t: ybas+pe
```

Remarque : la syntaxe en `lambda t:`, définissant d'une certaine manière une fonction «en ligne» est celle attendue par `np.piecewise()` dans le cas général. Dans le cas de la fonction créneau, on pouvait mettre seulement la valeur de la fonction car elle était constante et dans ce cas `np.piecewise()` interprète par défaut `constante` en `lambda t: constante`. On peut par ailleurs se passer de l'appel à `np.piecewise()` et utiliser une structure conditionnelle en `if (t//demiperiode)%2 != 0` mais la fonction dxoverdtTriangle renverra alors un scalaire. C'est suffisant pour la méthode d'Euler mais cette syntaxe permet de définir une fonction qui renvoie un tableau `numpy`, ce qui allège le code pour tracer des courbes par exemple.

Le reste du code a la même structure :

```
x = np.zeros(NombrePoints)
periode = 1/5 # en ms
tau = 1000*L/R # en ms
params = [periode,tau,-E/L,E/L] # E/L en A/s soit en mA/ms

for i in range(NombrePoints-1):
    x[i+1] = x[i]+ Deltat*dxoverdtTriangle(t[i],x[i],params)

figTriangle,axTriangle = plt.subplots()

axTriangle.set_xlabel('t (ms)')
axTriangle.set_ylabel('iL (mA)')
axTriangle.set_title('Réponse d\'un dipôle RL à une fonction triangle')
axTriangle.plot(t,x,'b',label='iL')
axTriangle.legend(loc='best',shadow=True)

figTriangle.show()
```