Oscillateur à pont de Wien numérique

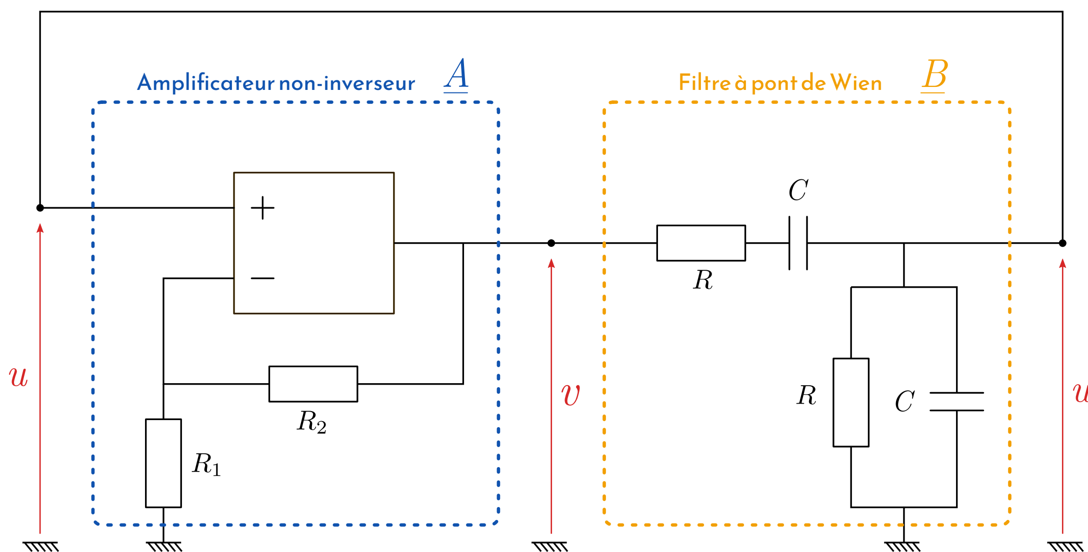
Capacités exigibles

- À l'aide d'un langage de programmation, simuler l'évolution temporelle d'un signal généré par un oscillateur

I Documents

Document 1 : Oscillateur à pont de Wien

L'oscillateur à pont de Wien est constitué d'un bloc amplificateur non-inverseur en bouvle rétroactive avec un filtre de Wien.



On peut alors écrire les relations suivantes entre les tensions u et v :

$$v = \begin{cases} (1+\alpha)\,u & \text{en régime linéaire} \\ \pm V_{sat} & \text{en régime saturé} \end{cases}$$

$$\underline{u} = \frac{1}{3 + j\left(\frac{\omega}{\omega_0} - \frac{\omega_0}{\omega}\right)}\,\underline{v} = \frac{j\omega_0\omega}{-\omega^2 + 3j\omega_0\omega + {\omega_0}^2}\,\underline{u}$$

Et on a posé les paramètres

$$\alpha = \frac{R_2}{R_1} \qquad \omega_0 = \frac{1}{RC}$$

II Énoncé

① Donner l'équation différentielle associée à la relation entrée-sortie du filtre de Wien.

② En déduire les deux équations différentielles de u, pour chacun des cas envisageable.

③ Retrouver la condition sur α pour avoir des oscillations naissantes.

4 Créer des variables pour chacun des paramètres physiques et numériques en leur donnant des valeurs qui vous semblent appropriées (quitte à les modifier plus tard) :

- `alpha` qui représente α ;
- `omega` qui représente ω_0 ;
- `Vsat` qui représente V_{sat} ;
- `bruit` un tableau de 2 valeurs aléatoires très faibles (cf. annexe) ;
- `T` la durée totale de la simulation ;
- `N` le nombre de points de la simulation ;
- `temps` la liste des temps sur lesquels on va mener l'intégration.

5 Écrire une fonction python `Wien(U, t)` prenant comme arguments

- `U` un vecteur (liste python) contenant pour un instant donné, la valeur de $u(t)$ et celle de $\dot{u}(t)$:

$$\dot{U} = [u, \dot{u}]$$

- `t` le temps auquel on mène le calcul

Et faire en sorte qu'elle renvoie la dérivée du vecteur d'entrée, savoir un nouveau vecteur contenant :

$$\dot{U} = [\dot{u}, \ddot{u}]$$

6 À l'aide de la fonction `odeint` (cf. annexe), compléter votre code afin de pouvoir afficher l'évolution de $u(t)$ dans un graphique, à partir d'une condition initiale stockée dans la variable `bruit`.

Faites varier la valeur de α et vérifiez se qu'il se passe quand on la choisit trop faible ou trop élevée.

7 Après l'utilisation de `odeint`, ajouter quelques lignes permettant de construire, à partir des solutions, le tableau de valeurs de $v(t)$.

Afficher ensuite $u(t)$ et $v(t)$ dans le même graph, puis visualiser le tracé de $v(u)$.

III Annexe

`np.linspace(xi, xf, N)`

Créer un tableau 1D allant de xi à xf en N points.
Exemple :

```
>>> np.linspace(0, 1, 11)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

`np.random.random(n)`

Créer un tableau 1D de n valeurs aléatoires choisies uniformément entre 0 et 1 :
Exemple :

```
>>> np.random.random(4)
array([0.85568752, 0.94219879, 0.01725846, 0.2365631 ])
```

`slicing`

Il est très facile d'extraire et de modifier certaines valeurs d'un tableau numpy.
Exemple : Si on a une matrice 2D notée M :

```
M = array([[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]])
```

- On peut extraire des valeurs :

```
>>> M[0]     # Renvoie la première ligne
array([1, 2, 3])
>>> M[0, 2]      # Renvoie la valeur en première ligne, troisième
    colonne
3
>>> M[:, 1]      # Renvoie toutes les valeurs de la deuxième colonne
array([2, 5, 8])
```

On remarque donc que les deux points ":" signifient "toutes les valeurs".

- On peut modifier certaines valeurs :

```
>>> M > 5   # Renvoie une matrice "calque" contenant un True
    à tout les emplacements où la valeur est supérieure à 5
array([[False, False, False],
       [False, False,  True],
       [ True,  True,  True]])
>>> M[M > 5] = 0 # Modifie toutes les valeurs de M
    supérieures à 5 en 0
>>> M
array([[1, 2, 3],
       [4, 5, 0],
       [0, 0, 0]])
```

odeint

Dans la bibliothèque `scipy.integrate`, se trouve une fonction `odeint` permettant de résoudre numériquement des équations différentielles, et de manière bien plus précise que la méthode d'Euler que l'on a implémenté l'année dernière :

```python
from scipy.integrate import odeint

...

solutions = odeint(F, Y0, X)
```

Les arguments doivent être les suivants :

- `F` la fonction qui encode l'équation différentielle (ici par exemple une équation d'ordre n sur une fonction $y : x \to y(x)$) :

$$Y'(x) = F(Y(x), x)$$

  Elle doit prendre elle-même en entrée :

  - `Y` un vecteur de taille n (liste python) contenant pour un x donné, les valeurs de $y(x)$ et ses $n-1$ premières dérivées :

$$Y(x) = \begin{pmatrix} y(x) \\ y'(x) \\ \vdots \\ y^{(n-1)}(x) \end{pmatrix}$$

  - `x` l'abscisse à laquelle on mène le calcul.

  Elle doit renvoyer le vecteur dérivé (toujours sous forme de liste python) :

$$Y'(x) = \begin{pmatrix} y'(x) \\ y''(x) \\ \vdots \\ y^{(n)}(x) \end{pmatrix}$$

- `Y0` les conditions initiales

$$Y(x_1) = \begin{pmatrix} y(x_1) \\ y'(x_1) \\ \vdots \\ y^{(n-1)}(x_1) \end{pmatrix}$$

- `X` La liste des abscisses sur lesquelles on va mener l'intégration :

$$X = [x_1, x_2, \dots, x_N]$$

  Souvent on utilise la fonction `np.numpy` (cf. précédemment), pour créer cette liste.

Alors `odeint` renvoie un tableau `numpy` de taille `N` × `n` avec

- `N` le nombre de points choisis pour intégrer l'équation
- `n` l'ordre de l'équation différentielle

$$\texttt{odeint} \implies \begin{pmatrix} y(x_1) & y'(x_1) & \dots & y^{(n-1)}(x_1) \\ y(x_2) & y'(x_2) & \dots & y^{(n-1)}(x_2) \\ \vdots & \vdots & & \vdots \\ y(x_N) & y'(x_N) & \dots & y^{(n-1)}(x_N) \end{pmatrix}$$