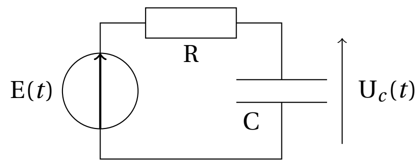
Résolution numérique d'équations différentielles

1 Introduction

On s'intéresse à présent au problème de la résolution d'équations différentielles, c'est-à-dire la détermination d'une fonction solution d'une équation faisant intervenir la fonction et ses dérivées. Parfois, de telles équations peuvent être résolues analytiquement, mais bien souvent, c'est une tâche impossible, et il faut se résoudre à chercher numériquement une approximation de la solution, ce qui sera le but de ce chapitre.

C'est un problème qui, lui aussi, trouve de nombreuses applications pratiques dans tous les domaines des sciences, les systèmes dont l'évolution étant décrite par une ou plusieurs équations différentielles étant monnaie courante.

Considérons par exemple le circuit électrique suivant (où le générateur est un générateur de tension parfait), dans l'approximation des régimes quasi-stationnaires :



La loi des mailles permet de relier les différentes tensions dans le circuit :

$$E(t) = U_r(t) + U_c(t)$$

Les caractéristiques des dipôles permettent de lier les tensions $U_r(t)$ et $U_c(t)$ au courant $I(t)$ circulant dans le circuit :

$$U_r(t) = R \cdot I(t) \quad \text{et} \quad I(t) = C \cdot \frac{dU_c(t)}{dt}$$

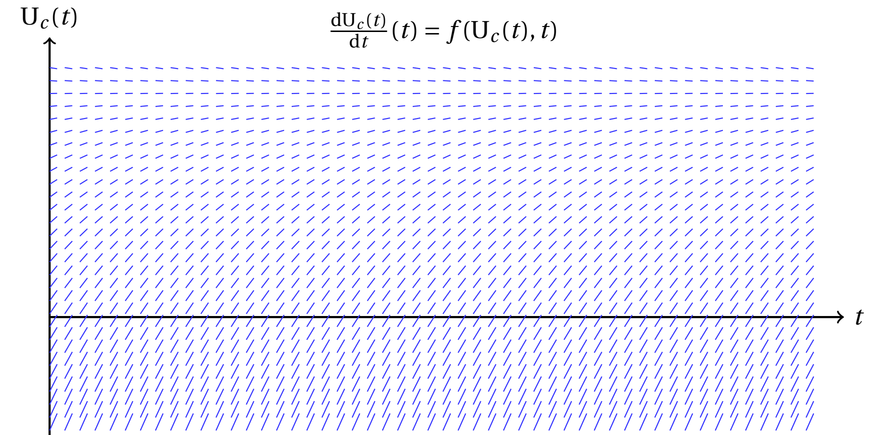
En éliminant $I(t)$ et $U_r(t)$ de ce système de trois équations, on peut obtenir une équation différentielle sur la tension $U_c(t)$ aux bornes du condensateur :

$$\frac{dU_c}{dt} + \frac{1}{RC} U_c(t) = \frac{1}{RC} E(t)$$

Supposons $E(t)$, R et C connus. On peut chercher à trouver l'évolution temporelle de la tension $U_c(t)$. Sa dérivée peut être isolée :

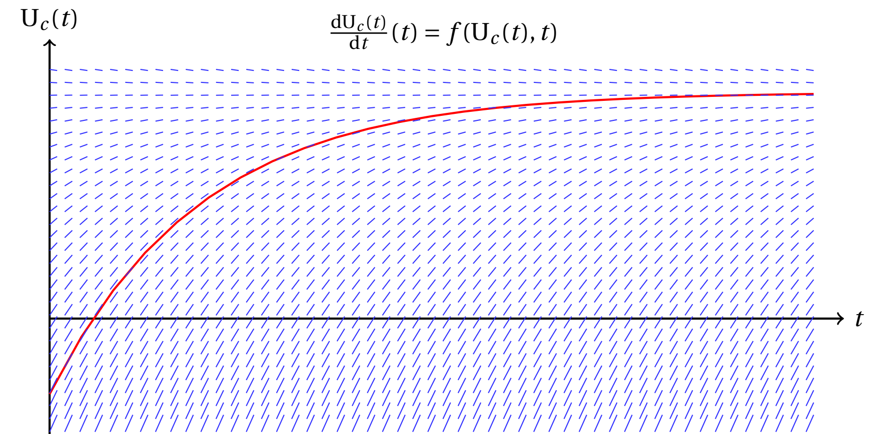
$$\frac{dU_c}{dt} = \frac{1}{RC}\big(E(t) - U_c(t)\big)$$

Dès lors, pour toute valeur de t et $U_c(t)$, on peut en déduire la valeur de la dérivée de $U_c(t)$. Dans le plan (t, U_c), cela revient à dire que l'on connaît en tout point M la tangente à la solution $U_c(t)$ si elle passe par ce point M. Par exemple, pour notre problème, en supposant $E(t) = E_0 = 3$ V pour $t \geq 0$, on a « champ de tangentes » de ce genre :



Il existe une infinité de solutions à cette équation différentielle (une infinité de courbes qui soient en tout points tangentes aux vecteurs tracés ci-dessus). Pour sélectionner celle qui correspond à l'évolution réelle de $U_c(t)$, il nous faut connaître un point par lequel passe la courbe, en général une « condition initiale ».

On peut par exemple supposer que la tension $U_c(t)$ aux bornes du condensateur en $t = 0$ est égale à $U_0 = -1$ V. Il ne reste alors qu'une seule et unique solution à l'équation différentielle, représentée ci-dessous :



Bien évidemment, dans le cas présent, le fait que $E(t)$ soit une constante permet de

résoudre analytiquement l'équation différentielle, dont la solution est

$$\mathrm{U}_c(t) = \mathrm{E}_0 + (\mathrm{U}_0 - \mathrm{E}_0)\,\mathrm{e}^{-t/\mathrm{RC}}$$

D'autres cas permettent une résolution analytique : lorsque $\mathrm{E}(t)$ est une fonction affine, polynomiale, sinusoïdale, etc [1]. Dans le cas général, toutefois, il n'existe pas de méthode universelle pour obtenir une solution analytique (c'est encore plus vrai dans le cas où l'équation différentielle est non-linéaire).

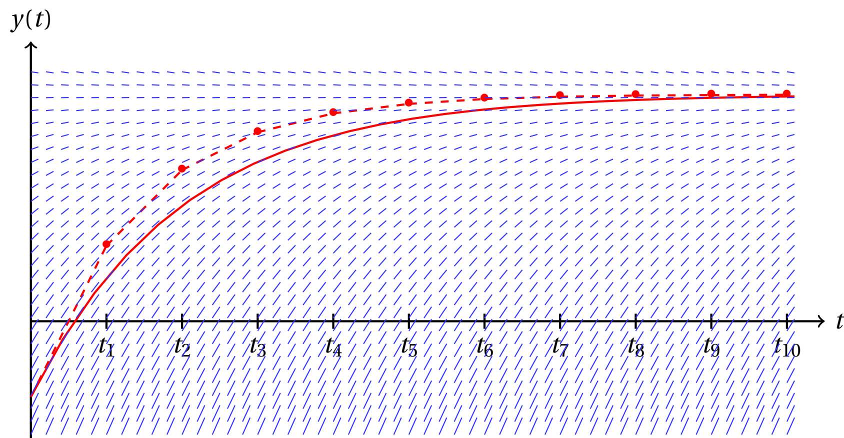
2 Principe de l'intégration numérique

2.1 Formalisation du problème

Une méthode numérique ne pourra pas retourner une fonction car une fonction représente possiblement une infinité de données (la valeur de $\mathrm{U}_c(t)$ pour toutes les valeurs de t qui nous intéressent), même si les fonctions courantes peuvent être définies avec beaucoup moins de données.

Supposons que l'on cherche la fonction $y : [t_0, t_0 + \mathrm{T}] \mapsto \mathbb{R}$ vérifiant, pour tout t dans l'intervalle $[t_0, t_0 + \mathrm{T}]$, une équation différentielle de la forme $y'(t) = f(y(t), t)$ et pour laquelle $y(t_0) = y_0$.

Une méthode numérique d'intégration fournira, pour un ensemble de t_k choisi dans $[t_0, t_0 + \mathrm{T}]$, un ensemble de valeurs y_k telles que les y_k soient aussi proches que possible des $y(t_k)$. Le résultat ne sera donc pas une fonction mais un ensemble de points (t_k, y_k), comme ci-dessous, aussi proches que possible de la fonction solution (traits pleins).



On dit que l'on utilise un *pas régulier* (noté h dans la suite) lorsque l'intervalle $[t_0, t_0 + \mathrm{T}]$ est subdivisé en n intervalles égaux, $t_{k+1} - t_k = h = \mathrm{T}/n$.

1. Ou bien définie par morceaux avec de telles fonctions.

2.2 Schéma d'intégration d'Euler explicite

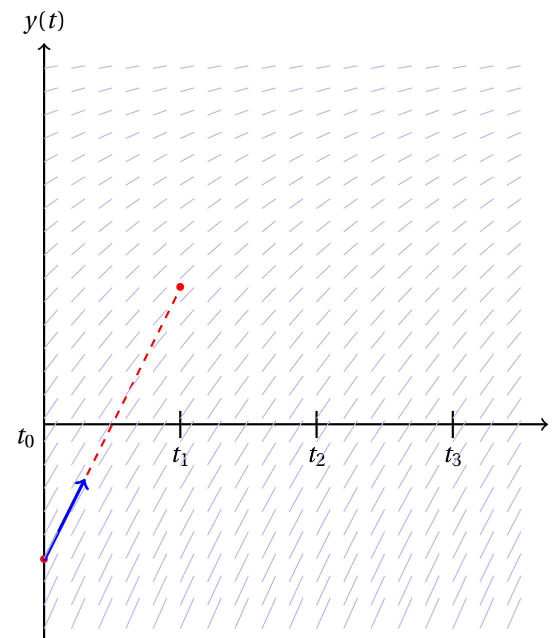
Un *schéma d'intégration* est une méthode qui calcule, successivement, les différents y_k à partir des y_j (avec $j < k$) et de la fonction f.

Il existe de très nombreux schémas d'intégration, qui diffèrent les uns des autres de par leur complexité, leur précision, leur stabilité numérique, etc. Le plus simple de tous ces schémas d'intégration est le schéma dit d'*Euler explicite*.

L'idée est simple. Connaissant on calcule y_k à partir de y_{k-1} via la relation suivante :

$$y_k = y_{k-1} + f(y_{k-1}, t_{k-1}) \times (t_k - t_{k-1})$$

Cela revient à considérer la tangente au point (t_{k-1}, y_{k-1}) obtenue grâce à la fonction f, et se diriger dans cette direction jusqu'à l'instant t_k. Ainsi, on obtient y_1 à partir de y_0 de la sorte :



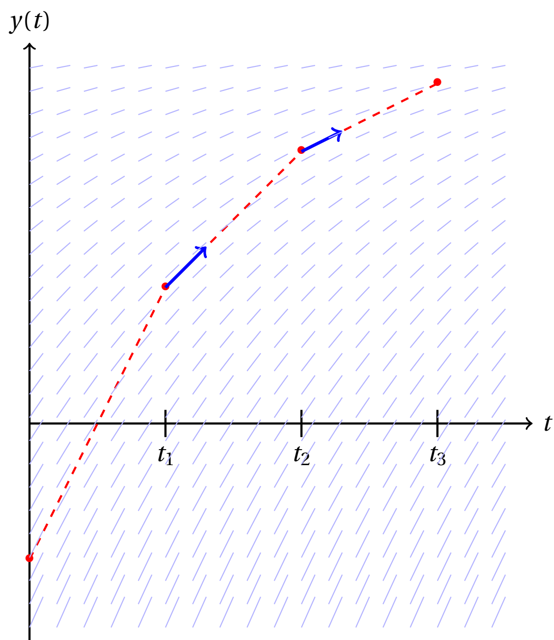
On peut voir cette relation entre y_k et y_{k-1} comme liée au développement de Taylor de $y(t)$ au premier ordre en $y(t_k)$:

$$y(t_k) = y(t_{k-1}) + y'(t_{k-1}) \times (t_k - t_{k-1}) + \mathrm{O}\big((t_k - t_{k-1})^2\big)$$

On ne conserverait donc ici que les deux premiers termes du développement.

Si c'est bien l'idée sous-jacente au schéma d'intégration d'Euler explicite, il faut être très prudent, car y_k et y_{k-1} ne correspondent pas exactement à $y(t_k)$ et $y(t_{k-1})$, ils n'en sont que des approximations.

Les différents y_k sont ainsi calculés de proche en proche. Par exemple pour y_2 et y_3 sur notre exemple :



2.3 Erreur de consistance

Évidemment, supposer que la fonction est un segment entre t_k et t_{k+1} dont la pente est estimée au seul instant t_k ne peut pas donner une solution exacte. L'intégration numérique de notre équation différentielle en utilisant une subdivision régulière du temps de pas 0.5RC ($\forall k, (t_k - t_{k-1}) = h = 0.5\text{RC}$) donne le résultat illustré à la page précédente : bien qu'on ne soit pas trop loin de la solution, on s'en éloigne quand même quelque peu.

On définit *l'erreur de consistance* e_k, pour un schéma d'intégration et une équation différentielle donnés, comme l'écart $|y_k - y(t_k)|$ en supposant que y_{k-1} était égal à $y(t_{k-1})$.

Une méthode de pas régulier h est dite *d'ordre* p lorsque $e_k = O(h^{p+1})$ lorsque h tend vers 0.

Elle est dite *consistante* lorsque $\sum_{k=1}^{n} e_k \xrightarrow[n\to\infty]{} 0$.

On souhaite évidemment utiliser des schémas d'intégration consistants, et dans la mesure du possible avec un ordre le plus grand possible, afin d'éviter les imprécisions.

Attention, l'erreur commise à l'issue de l'intégration numérique sur $[t_0, t_0 + T]$ ne correspond pas à la somme des erreurs de consistance.

Lorsque l'on utilise un pas h avec une méthode d'ordre p, on effectue T/h pas d'intégration, chacun ayant une erreur de consistance en $O(h^{p+1})$. On s'attend donc à une erreur de l'ordre de $T/h \times O(h^{p+1}) = O(h^p)$. Seulement, on s'écarte progressivement de la solution à chaque étape, et les erreurs commises peuvent être beaucoup plus grandes que les erreurs de consistance e_k. Il n'est pas possible de montrer que l'erreur sur l'ensemble de l'intervalle est en $O(h^p)$ sans faire d'autres hypothèses sur la fonction f.

Dans le cas du schéma d'intégration d'Euler explicite, on a

$$e_k = \Big|y(t_k) - \Big(y(t_{k-1}) + f(y(t_{k-1}), t_k) \times (t_k - t_{k-1})\Big)\Big|$$

Si f est $\mathscr{C}^1$, alors le schéma régulier d'Euler implicite est consistant d'ordre 1. En effet, l'erreur de consistance e_k correspond très précisément au terme que l'on a négligé dans le développement de Lagrange, $O\big((t_k - t_{k-1})^2\big)$, ce qui en fait une méthode d'ordre 1.

Pour prouver sa consistance, puisque f est $\mathscr{C}^1$, on peut en déduire que y est $\mathscr{C}^2$, et donc que, selon l'égalité de Taylor-Lagrange, pour tout intervalle $[t_{k-1}, t_k]$ il existe un $\tau_k \in [t_{k-1}, t_k]$ vérifiant

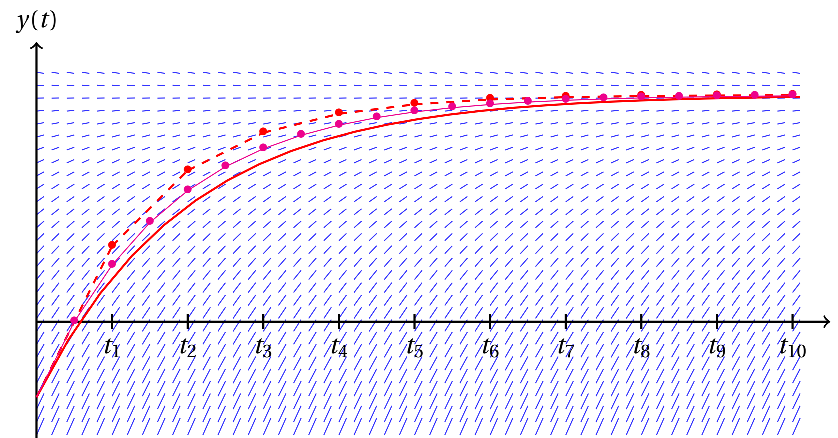
$$y(t_k) = y(t_{k-1}) + y'(t_{k-1}) \times h + y''(\tau_k) \times \frac{h^2}{2} \quad \text{où} \quad h = \frac{T}{n} = t_k - t_{k-1}$$

Autrement dit, l'erreur de consistance vérifie $e_k = \left|\frac{y''(\tau_k)}{2}\right| \times \frac{h^2}{2}$.

y étant $\mathscr{C}^2$, notons M_2 un majorant de $|y''|$ sur l'intervalle $[t_0, t_0 + T]$. On a alors

$$\left|\sum e_k\right| \leqslant \sum |e_k| \leqslant \sum M_2 \times \frac{h^2}{2} = n \times M_2 \times \frac{h^2}{2} = \frac{M_2 T^2}{2n} \xrightarrow[n\to\infty]{} 0$$

Le schéma d'intégration d'Euler implicite est donc bien consistant. En diminuant le pas h (en augmentant le nombre de subdivisions), on tend donc à s'approcher de la solution comme on peut le voir ci-dessous :



2.4 Implémentation

L'implémentation de la méthode d'Euler explicite est des plus simple. Elle prend trois arguments : la fonction f, la valeur y_0 de la fonction y à l'instant t_0 et la liste des t_k (pour $0 \leqslant k < n$) pour lesquels on cherche les y_k.

Elle retourne une liste des y_k (toujours pour $0 \leqslant k < n$), lesquels sont calculés de proche en proche grâce à la relation

$$y_k = y_{k-1} + f(y_{k-1}, t_{k-1}) \times (t_k - t_{k-1})$$

On a donc :

```
def EulerExplicite(f, y0, listeT) :
    listeY = [ y0 ]

    for k in range(1, len(listeT)) :
        pas = listeT[k] - listeT[k-1]
        yk = listeY[k-1] + f( listeY[k-1], listeT[k-1] ) * pas
        listeY.append(yk)

    return listeY
```

Pour l'utiliser, on crée la fonction f et la liste des instants t_k [2], avant d'appeler la fonction `EulerExplicite` :

```
R, C, Eo = 1.0e3, 1.0e-6, 3.0

def f(y, t) :
    return (Eo-y) / (R*C)

N = 10
pas = 5*R*C/N
listeT = [ 0.0 + i*pas for i in range(N+1) ]

y0 = -1.0
listeY = EulerExplicite(f, y0, listeT)
```

On peut ensuite afficher le résultat grâce à la commande `plot` en écrivant par exemple

```
from matplotlib.pyplot import plot, show

plot(listeT, listeY, 'ro')
show()
```

2. On aurait pu également utiliser `numpy.linspace(0.0, 5.0*R*C, N+1)` pour construire la liste des instants t_k.

2.5 Méthode de Runge-Kutta point milieu

Le schéma d'intégration numérique d'Euler explicite est le seul qui soit à connaître. C'est aussi, malheureusement, un des plus inefficaces, exigeant des pas très petits pour que le résultat soit de bonne qualité.

De la même façon qu'il existe de nombreuses méthodes pour estimer numériquement l'intégrale d'une fonction, il existe de très nombreux schémas d'intégration. Le schéma d'intégration d'Euler explicite s'apparente à la méthode des rectangles gauches : la valeur de la fonction f en t_{k-1} est utilisée pour tout l'intervalle $[t_{k-1}, t_k]$.

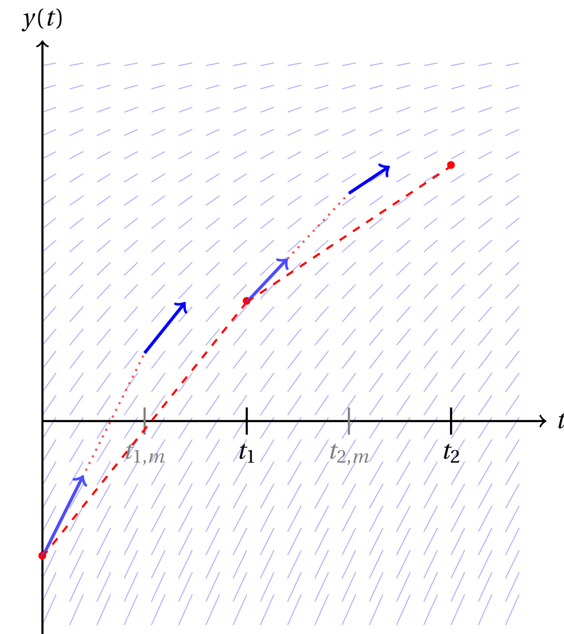
Nous avions vu que la méthode du point milieu donnait une meilleure estimation de l'intervalle, pour un même pas. On souhaiterait donc estimer la valeur de la pente, pour le calcul de y_k à partir de y_{k-1}, non pas en t_{k-1}, mais en $t_{k,m} = (t_{k-1} + t_k)/2$.

Seul ennui, on ne sait pas par où passe la fonction y à l'instant $t_{k,m}$!

On va donc travailler en deux temps :

- on estime un $y_{k,m}$ correspondant à un instant $t_{k,m} = (t_{k-1} + t_k)/2$, à partir de y_{k-1} grâce au schéma d'intégration d'Euler explicite ;
- on se sert de la pente $f(y_{k,m}, t_{k,m})$ pour calculer y_k, en repartant du point (t_{k-1}, y_{k-1}).

Un schéma valant mieux qu'un discours, on construit par exemple y_1 à partir de y_0, puis y_2 à partir de y_1, de la sorte [3] :



3. Attention, pour plus de lisibilité, les intervalles $[t_0, t_1]$ et $[t_1, t_2]$ sont un peu plus larges que précédemment.

Pour déterminer y_k à partir de y_{k-1}, cela revient à écrire :

$$y_k = y_{k-1} + f\left(y_{k-1} + f(y_{k-1}, t_{k-1}) \times \frac{(t_k - t_{k-1})}{2}, \frac{(t_{k-1} + t_k)}{2}\right) \times (t_k - t_{k-1})$$

L'implémentation n'est pas beaucoup plus compliquée que pour le schéma d'Euler explicite, et reprend les deux étapes (calcul de $y_{k,m}$, puis y_k) :

```
def RK_Milieu(f, y0, listeT) :
    listeY = [ y0 ]

    for k in range(1, len(listeT)) :
        pas = listeT[k] - listeT[k-1]
        ym = listeY[k-1] + f( listeY[k-1], listeT[k-1] ) * (pas/2.0) )
        yk = listeY[k-1] + f( ym, listeT[k-1]+(pas/2.0) ) * pas )
        listeY.append(yk)

    return listeY
```

On peut montrer que le schéma régulier de Runge-Kutta point milieu est un schéma consistant d'ordre 2.

En fait, il existe une collection de schémas dits de Runge-Kutta, et il est possible d'en fabriquer avec un ordre quelconque. Le calcul de y_k est cependant d'autant plus compliqué que l'on choisit un ordre élevé, aussi est-il parfois plus intéressant de réduire le pas h plutôt que d'utiliser un schéma d'intégration d'un ordre plus élevé.

2.6 Schémas d'intégration à pas adaptatif

Comme on l'a vu, le choix du pas $t_k - t_{k-1}$ est crucial pour obtenir autant de précision que possible. Cependant, un pas plus petit nécessite, naturellement, davantage de calculs.

Le choix du pas $t_k - t_{k-1}$ n'est donc pas simple. Par ailleurs, utiliser un pas régulier n'est pas toujours la meilleure solution possible. En effet, pour certaines valeurs de t, un pas important peut convenir, tandis que pour d'autres valeurs de t, il conviendrait d'utiliser des plus petits pas. De la même façon que, lorsque l'on conduit un véhicule sur une route, on peut aller (raisonablement) vite dans les lignes droites mais on est contraint de ralentir pour négocier les virages délicats.

Aussi peut-on envisager de laisser à l'intégrateur le soin de choisir les pas d'intégration h_k, potentiellement différents à chaque étape. Pour ce faire, il est besoin de pouvoir estimer l'erreur `err` que l'on commet lorsque l'on effectue une étape de l'intégration, afin de le comparer à une erreur « acceptable » `eps` : si `err` ≤ `eps`, on peut augmenter le pas d'intégration, et si au contraire `err` > `eps`, la dernière étape effectuée n'est pas acceptable, et il convient de la refaire avec un pas plus petit.

Pour estimer l'erreur `err`, on peut calculer la valeur y_k de deux façons différentes. Par exemple, en utilisant le schéma d'Euler et le schéma de Runge-Kutta point milieu, plus précis. On peut espérer que l'erreur commise lors d'une étape d'intégration soit liée à la différence entre la valeur y_k fournie par l'intégrateur d'Euler, et la valeur y_k fournie par le schéma de Runge-Kutta, plus précis.

Pour choisir un nouveau pas h', connaissant `err` et `eps`, on peut utiliser

$$h' = \sqrt{\frac{\text{eps}}{\text{err}}} \times h$$

En effet, de la sorte, si `err<eps`, le pas va augmenter, et inversement, si `err>eps`, on obtient une diminution du pas.

En Python, cela donne [4], avec une signature légèrement différente pour la fonction par rapport aux précédentes (on passe t_0 et `duree` plutôt qu'une liste d'instants t_k, et également une erreur acceptable `eps`) :

```
def EulerAdapt(f, y0, t0, duree, eps) :
    Y, T = [ y0 ], [ t0 ]

    h = 1.0                     # Un pas initial, choisi arbitrairement

    while T[-1] < t0+duree :    # Tant que l'on n'a pas atteint t0+duree
        # On estime un y_k avec le schéma d'Euler explicite :
        y_E = Y[-1] + f(Y[-1], T[-1]) * h

        # On fait de même avec le schéma de Runge-Kutta point milieu :
        y_R = Y[-1] + f(Y[-1] + f(Y[-1], T[-1])*h/2.0, T[-1]+h/2.0) * h

        # L'erreur est assimilée à la différence entre les estimations
        err = abs(y_E - y_R)

        # Si l'erreur est raisonnable, on a un nouveau point y_k, t_k
        if err < eps :
            T.append(T[-1] + h)
            Y.append(y_R)      # On préfère conserver l'estimation de RK

        # On adapte le pas pour la suite
        h = 0.95 * math.sqrt(eps/err) * h

    # On retourne les y_k mais aussi les t_k, choisis par l'algorithme
    return T, Y
```

4. On remarquera un coefficient 0.95 dans le calcul du pas qui, sans être indispensable, apporte parfois un peu plus de stabilité numérique.

3 Utilisation de scipy

3.1 Équations différentielles du premier ordre

Nous avons décrit dans ce chapitre le principe de quelques schémas d'intégration numérique dans le but d'illustrer le principe de la résolution numérique d'une équation différentielle. Dans la pratique, **on ne réécrit jamais un schéma d'intégration** lorsqu'il s'agit de résoudre une équation différentielle !

En effet, comme nous l'avons vu, les méthodes les plus simples ne sont pas les plus précises ni les plus stables numériquement. Des spécialistes de la question ont passé de longues heures à mettre au point des algorithmes efficaces (utilisant des schémas d'ordre élevés, une adaptation du pas si besoin, etc.) et il convient de s'en servir.

D'autant que cela ne représente aucune difficulté supplémentaire : la fonction `odeint` du module `scipy.integrate`, qui permet de résoudre une équation différentielle du premier ordre de la forme $y' = f(y, t)$, attends très exactement les mêmes arguments que notre fonction `EulerExplicite` par exemple !

Pour le problème qui nous occupe, on écrira donc très simplement :

```
from scipy.integrate import odeint

listeY = odeint(f, y0, listeT)
```

(où `f`, `y0` et `listeT` ont été définis de façon identique à ce que l'on a fait dans la section précédente).

`odeint` ne renvoie pas une liste de y_k, mais un `numpy.array`, qui toutefois se comporte de la même façon pour la plupart des utilisations (notamment, on peut tracer le résultat avec la même commande `plot(listeT, listeY)` que précédemment).

3.2 Systèmes d'équations différentielles du premier ordre

Parfois, on doit résoudre des équations différentielles du premier ordre couplées, par exemple un système de type :

$$\begin{cases} y_1'(t) = f_1(y1(t), y_2(t), t) \\ y_2'(t) = f_2(y1(t), y_2(t), t) \end{cases}$$

Il n'est bien entendu pas possible de résoudre les équations différentielles séparément. On les résoud donc simultanément, en calculant $y_{1,k}$ et $y_{2,k}$ correspondant à un instant t_k à partir, par exemple, des $y_{1,k-1}$, $y_{2,k-1}$ et t_{k-1} (en utilisant le schéma d'Euler explicite pour chacune des deux estimations, ou tout autre schéma d'intégration).

La fonction `odeint` permet naturellement de résoudre de tels systèmes, pour peu que l'on fournisse une fonction `f` « vectorielle », qui prenne pour premier argument une liste (ou un vecteur, un tuple...) des y_i et pour second argument t et retourne une liste (ou un vecteur, un tuple...) contenant les $y_i' = f_i(y_i, t)$ correspondant.

Par exemple, pour le système

$$\begin{cases} u'(t) = 2u(t) + v(t) \\ v'(t) = -u(t) + 2v(t) \end{cases}$$

on écrira

```
def f(Y, t) :
  u, v = Y                        # On extrait u et v de Y
  return [ 2*u+v, -u+2*v ]        # On retourne la liste des dérivées
```

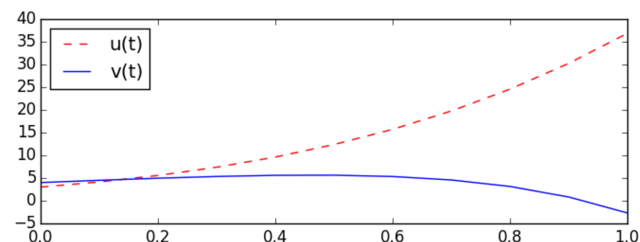
Et pour la résolution proprement dite, sur $[0, 1]$ avec par exemple $u(0) = 3$ et $v(0) = 4$:

```
T = [ 0.1*i for i in range(11) ]
Y0 = [ 3.0, 4.0 ]

from scipy.integrate import odeint
Y = odeint(f, Y0, T)
```

Le résultat retourné (ici `Y`) sera une *matrice* (sous la forme d'un `numpy.array`) où chaque ligne correspond aux différents instants de `T`, et les colonnes respectivement à $u(t)$ et $v(t)$.

Ainsi, les u_k peuvent être obtenus en écrivant `Y[:, 0]` et les v_k, `Y[:, 1]`. On pourra donc effectuer le tracé de $u(t)$ et $v(t)$ de la sorte :

```
from matplotlib.pyplot import plot, legend, show

plot(T, Y[:, 0], 'r--', label="u(t)")
plot(T, Y[:, 1], 'b', label="v(t)")
legend(loc="upper left")
show()
```



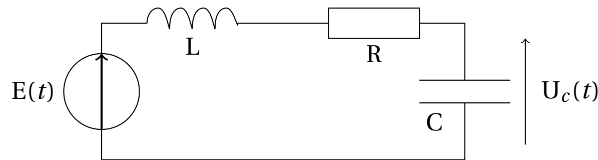
4 Équations différentielles du second ordre

4.1 Mise en équation

On s'intéresse à présent à des équations différentielles du second degré, c'est-à-dire à la recherche de fonction y solutions d'équations du type

$$y''(t) = f\left(y'(t), y(t), t\right)$$

Par exemple, on peut s'intéresser à la détermination de $U_c(t)$ dans un circuit de type « RLC série » tel que celui ci-dessous :



pour lequel on peut établir aisément, grâce à la loi des mailles et aux caractéristiques des dipôles,

$$E(t) \;=\; U_l(t) + U_r(t) + U_c(t) \;=\; LC\frac{d^2U_c}{dt}(t) + RC\frac{dU_c}{dt}(t) + U_c(t)$$

ce qui conduit à

$$\frac{d^2U_c}{dt}(t) \;=\; -\frac{R}{L}\frac{dU_c}{dt}(t) - \frac{1}{LC}U_c(t) + \frac{1}{LC}E(t)$$

Seulement, les schémas d'intégration que l'on a vu ne permettent pas de résoudre des équations différentielles du second ordre. On pourrait créer spécifiquement des schémas d'intégration pour cette tâche [5] mais on préférera transformer de tels problèmes en problème du premier ordre.

Lorsque l'on étudie une équation différentielle comme celle du dessus, la condition initiale $U_c(t_0)$ n'est plus suffisante : il faut une seconde condition initiale, par exemple $U_c'(t_0)$ (ou $I(t_0)$, ce qui revient exactement au même, au coefficient C près).

L'état du système n'est donc plus défini seulement par $U_c(t)$ mais par le « couple » $U_c(t), U_c'(t)$. De la même façon qu'en mécanique, l'état d'un mobile ponctuel n'est pas seulement défini par sa position mais par sa position et sa vitesse.

On peut donc envisager notre équation différentielle de la sorte :

$$\frac{d}{dt}\begin{pmatrix} y(t) \\ y'(t) \end{pmatrix} = \begin{pmatrix} y'(t) \\ y''(t) \end{pmatrix} = \begin{pmatrix} y'(t) \\ f(y'(t), y(t), k \end{pmatrix}$$

5. ils seraient même, pour un pas comparable, vraisemblablement plus précis !

On a donc converti une équation différentielle du second ordre en dimension 1 en une équation différentielle du premier ordre, certes en dimension 2, mais qui pourra être résolue par les outils que l'on a déjà vus.

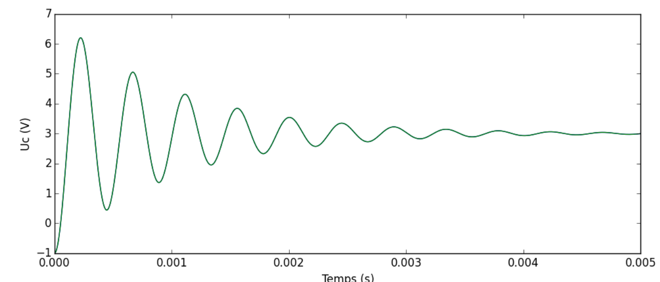
4.2 Résolution numérique avec `odeint`

Pour notre problème, on définira donc la fonction f de la sorte si $E(t) = E_0 = \text{cste}$:

```
def f(Y, t) :
    Uc, DerUc = Y
    return [ DerUc, -(R/L)*DerUc + (1/(L*C))*(Eo-Uc) ]
```

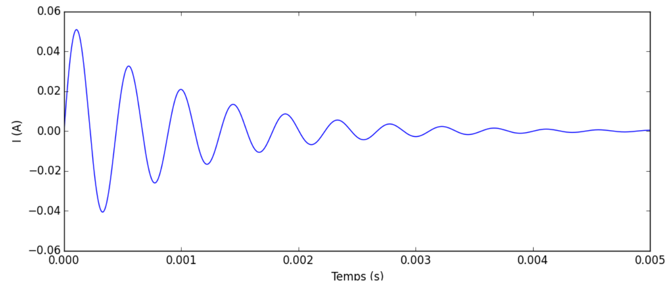
Il ne reste alors qu'à choisir les paramètres (par exemple $L = 5$ mH, $C = 1$ µF, $R = 10\,\Omega$ et $E(t) = E_0 = 3$ V) et les conditions initiales (ici $U_c(t_0) = -1$ V et $U_c'(t_0) = I(t_0)/C = 0$), un intervalle de temps ($[0, 500RC]$ ici) ainsi qu'un nombre de subdivisions ($N = 1000$), faire appel à `odeint` et tracer le résultat :

```
R, L, C, Eo = 1.0e1, 5.0e-3, 1.0e-6, 3.0

N = 1000
pas = 500*R*C/N
listeT = [ 0.0 + i*pas for i in range(N+1) ]

y0 = [ -1.0, 0.0 ]

from scipy.integrate import odeint
listeY = odeint(f, y0, listeT)

from matplotlib.pyplot import plot, xlabel, ylabel, show

plot(listeT, listeY[:,0])
xlabel("Temps (s)")
ylabel("Uc (V)")
show()
```



On peut également tracer aisément la courbe $I(t) = C\frac{dU_c}{dt}(t)$ du courant dans le circuit :

```
plot(listeT, C * listeY[:,1])
xlabel("Temps (s)")
ylabel("I (A)")
show()
```



Avec un facteur de qualité $Q = \frac{1}{R}\sqrt{\frac{L}{C}} \simeq 7.1$ on a (fort heureusement) bien obtenu un régime pseudo-périodique.