Objectifs *(extrait du programme) Problème dynamique à une dimension, linéaire ou non, conduisant à la résolution approchée d'une équation différentielle ordinaire par la méthode d'Euler. On compare les résultats obtenus avec les fonctions de résolution approchée fournies par une bibliothèque numérique. On met en évidence l'impact du pas de discrétisation et du nombre d'itérations sur la qualité des résultats et sur le temps de calcul.*

1. ÉQUATIONS DIFFÉRENTIELLES D'ORDRE 1

1.1. Point de terminologie et de forme

a) terminologie

Une équation différentielle d'une fonction à une variable est une *Équation Différentielle Ordinaire*, en anglais *Ordinary Differential Equation*. Pour sa résolution, Scilab dispose donc d'une fonction qui se nomme `ode`.
Avec des fonctions à plusieurs variables, les équations différentielles comportant des dérivées partielles sont appelées *Équations aux Dérivées Partielles*, en anglais *Partial Differential Equation*. Exemple : l'équation de propagation d'une onde transversale le long d'une corde. La fonction correspondante, pour info, est `pde`.

b) mise en forme d'une équation différentielle

Une équation différentielle du premier ordre est une relation entre une fonction et sa dérivée. Cette fonction, qui est l'inconnue, sera noté ici y ; elle s'exprime en fonction d'une variable que nous noterons t, car en physique il s'agit le plus souvent du temps ⇒ $y = y(t)$. Une équation différentielle du premier ordre est donc une relation du type : $\frac{dy}{dt} = f(t, y)$ c'est-à-dire $y'(t) = f(t, y(t))$.

- Exemple 1, tiré de l'aide intégrée à Scilab : $\frac{dy}{dt} = y^2 - y\sin t + \cos t$

Cette équa. diff. s'écrira $y'(t) = f(t, y(t))$ avec $f(t, y(t)) = y^2 - y\sin t + \cos t$

Dans Scilab, on déclarera :

```
function yprime = yprime(t,y)  // le nom "yprime" est plus parlant que "f".
   yprime = y^2 - y*sin(t) + cos(t)
endfunction
```

Remarque : cette équa. diff. n'est pas linéaire.

- Exemple 2, chute amortie par frottements fluides (cas linéaire) : $\frac{dy}{dt} + \frac{y}{\tau} = K$

y est ici la vitesse, fonction du temps t ; τ et K sont deux constantes, τ est la constante de temps.

Cette équa. diff. s'écrira $y'(t) = f(t, y(t))$ avec $f(t, y(t)) = K - \frac{y(t)}{\tau}$

On remarque qu'ici, $f(t, y(t)) = f(\not{t}, y(t))$: f ne dépend pas explicitement de t. *(On dit que l'équation est autonome : ces équations ont des propriétés particulières)*. Mais Scilab impose une contrainte :

Pour définir une équation différentielle en y dans Scilab, la fonction qui définit la dérivée première y' doit toujours être déclarée avec les deux variables t et y (dans cet ordre : variable puis fonction) même si elle ne dépend que de y.

Par conséquent, on déclarera que y' est fonction de t et y, ou bien, si on reprend des notations plus spécifiques, que v' est fonction de t et v :

```
tau = 1;  // paramètres (exemple)
K = 0;
function vprime = vprime(t,v)  // on aurait également pu choisir "vpoint"
   vprime = K -v/tau
endfunction
```

Cette équa. diff. est linéaire : nous savons résoudre ce type d'équations "à la main". De plus, ses coefficients sont constants.

1.2. Résolution numérique : position du problème

Soit une équation différentielle pour la fonction $y(t)$ de la forme suivante : $y'(t) = f(t, y(t))$
Conditions aux limites : définie par $(t_0, y_0) \in \mathbb{R}^2$, abscisse t_0 et ordonnée y_0 telle que $y_0 = y(t_0)$. Dans le cas où la variable t désigne vraiment le temps, il s'agira généralement d'une condition initiale ($t_0 = 0$), d'où $y_0 = y(0)$.
La solution de cette équa diff est la fonction $y : t \mapsto y(t)$.

La résolution numérique d'une équation différentielle de ce type consiste, à partir d'un ensemble de N valeurs de t suffisamment proches, à calculer des valeurs approchées de $y(t)$.
Les N valeurs de t sont des instants t_n sur un intervalle $[0,T]$. On divise cet intervalle en N sous-intervalles de longueur $h=T/N$ et on définit les instants : $t_n = nh$, où l'entier n varie de 0 à N. Le terme h se nomme le pas : $t_{n+1} = t_n + h$.
Les valeurs approchées de $y(t)$ sont alors les termes d'une suite $y_n \approx y(t_n)$ avec $n \in [0,N]$.
L'accès à la solution se fait alors en reliant graphiquement par `plot2d` les différents y_n : le résultat est sensé être une bonne approximation du graphe de $y(t)$.

Problème : comment calculer les y_n ?
Scilab dispose d'une fonction toute faite, `ode`, comme nous l'avons dit. Cette fonction étant très performante, elle pourra nous servir de référence. L'utilisation de cette commande sera développée dans un TP ultérieur.
Ce qui nous intéresse ici est de *construire* une fonction remplissant le même rôle, pour comprendre "comment ça marche".
Nous nous limiterons à la méthode dite d'Euler, basée sur la formule de Taylor.

1.3. Développement de Taylor

On rappelle le développement de Taylor à l'ordre 1 de la fonction $y(t)$ au voisinage de t_0 :
$y(t) = y(t_0) + (t - t_0)y'(t_0) + (t - t_0)\varepsilon(t - t_0)$, soit en négligeant ε, si t et t_0 sont suffisamment proches :
$y(t) \approx y(t_0) + (t - t_0)y'(t_0)$

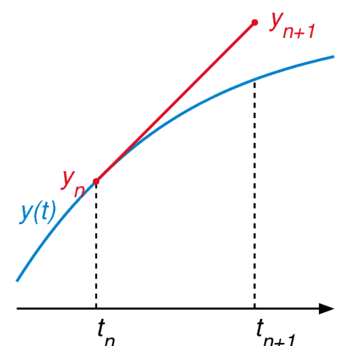
➢ *Remarque :*

l'équation précédente s'écrit aussi $y'(t_0) \approx \dfrac{y(t) - y(t_0)}{(t - t_0)}$ *qui n'est autre qu'une simplification de l'écriture* $y'(t_0) = \lim\limits_{t \to t_0} \dfrac{y(t) - y(t_0)}{(t - t_0)}$

Appliqué entre deux dates t_n et t_{n+1}, on obtient : $y(t_{n+1}) \approx y(t_n) + (t_{n+1} - t_n)y'(t_n)$
Soit, en introduisant le pas $h = t_{n+1} - t_n$: $y(t_{n+1}) \approx y(t_n) + h\, y'(t_n)$

1.4. Méthode d'Euler

Cette méthode consiste à utiliser le résultat précédent pour calculer successivement l'ensemble des termes de la suite y_n à partir de la valeur initiale y_0. Ainsi, le terme y_{n+1} se calcule à partir du précédent par :
$y_{n+1} = y_n + h\, y'(t_n)$.
Illustration sur le schéma ci-contre de l'avancée par approximation sur la tangente au point initial.



L'équation différentielle nous indiquant que $y'(t) = f(t, y(t))$, on obtient finalement :

$$\boxed{y_{n+1} = y_n + h\, f(t_n, y_n)}$$

2. MISE EN ŒUVRE DANS SCILAB

Nous allons distinguer plusieurs étapes :

- définition de l'équation différentielle par une première fonction,
- traduction de la méthode d'Euler par une seconde fonction,
- tracé de la solution,
- test de la validité de la méthode.

2.1. Définition de l'équation différentielle

Supposons que nous cherchions la solution de l'équa diff suivante : $\frac{dy}{dt}+\frac{y}{\tau}=0$

La première chose à faire est de la traduire sous forme d'une fonction Scilab, dans un script, en partant de la forme équivalente $y' = K - \frac{y}{\tau}$, avec $K = 0$ (l'introduction de K pourra permettre des variantes plus facilement) :

```
//équadiff yprime+y÷tau=K
// paramètres
tau =  3;
K =  0;
// équadiff
function yprime = yprime(t, y)
        yprime = K -y/tau
endfunction
```

équadiff yprime+ysurtau=0.sce

NB : l'utilisation de ’ dans un nom de script pose problème au moment son exécution.

2.2. Fonction `euler`

Nous allons procéder par étape, en partant d'un programme "basique" que nous enrichirons petit à petit.

a) <u>cœur du programme : instruction itérative</u>

L'itération de la méthode d'Euler s'écrit : `y=y+h*yprime(t,y)`

Reste à définir quels arguments donner à la fonction `euler` :

- la position initiale `y0` : nécessaire puisque d'un problème à l'autre elle changera.
- la date initiale sera supposé nulle quel que soit le problème, donc on ne l'entrera pas.
- la date finale `T` : nécessaire, pour définir les limites de l'étude.
- le pas `h` ou le nombre `N` de sous-intervalles : on a le choix. On optera pour `N` car il est indépendant de `T` et qu'on pourra l'utiliser directement comme borne finale pour la récurrence. On en déduira le pas par `h = T/N`

Cela conduit alors à une fonction `euler(y0,N,T)`

- on pourrait également ajouter l'équation différentielle, c'est-à-dire la fonction `yprime`.

On aurait alors la fonction `euler(yprime,y0,N,T)` . On envisagera cette option à la fin seulement.

Une première version de la méthode d'Euler est donc la suivante :

```
// euler version 1
function y = euler(y0,N,T)
    y = y0;  //initialisation de y
    t = 0;   //initialisation de t
    h = T/N;  //calcul du pas
    for i=1:N
        y = y+h*yprime(t,y);    //on calcule y(i) en fonction de y(i-1) et t(i-1)
        t = t+h;                //on calcule t(i) en fonction de t(i-1)
    end
endfunction
```

euler v1.sce

👁 *Instruction itérative. Ici l'utilisation d'une boucle* `for`.

→ utilisation dans la console (en ligne de commande Scilab), avec valeur initiale $y(0)=10$, nombre d'intervalles de temps $N=5$, instant final $T=10$:

```
--> euler(10,5,10)
 ans  =
   0.0411523
```

b) renvoi des arguments

Nous constatons que l'appel de la fonction précédente `euler v1` ne fournit que l'approximation finale de $y(T)$, sans les valeurs intermédiaires, c'est normal. On y remédie en créant la liste complète `liste_y` , et en renvoyant cette valeur :

```
// euler version 2 (en gras les changements)
function liste_y = euler(y0,N,T)
    y = y0;
    liste_y =[y0];
    t = 0;
    h = T/N;
    for i=1:N
        y = y+h*yprime(t,y);
        t = t+h;
        liste_y = [liste_y,y];  //construction du vecteur y
    end
endfunction
```

euler v2.sce

→ utilisation dans la console :

```
--> euler(10,5,10)
 ans  =
   10.   3.3333333   1.1111111   0.3703704   0.1234568   0.0411523
```

On obtient un vecteur-ligne. Normal, on a écrit le tableau sous la forme `[a,b]`. Pour obtenir un vecteur-colonne, préférable pour les graphes, il aurait fallu écrire `[a;b]` ou `[a,b]'`. Nous ferons la modification dans la version suivante.

Il peut être aussi commode, pour le tracé par exemple, que la fonction renvoie également les instants successifs où sont effectuées les approximations. Il est vrai que le vecteur temps correspondant serait directement accessible par `linspace(0,T,N+1)`, mais dans le cas où l'on fera varier le pas h, pour tester la validité de la méthode d'Euler, il sera plus simple de construire le vecteur temps en même temps que le vecteur "approximation de $y(t)$".

```
// euler version 3
function [liste_t,liste_y] = euler(y0,N,T)
    y = y0;
    liste_y =[y0];
    t = 0;
    liste_t =[0];
    h = T/N;
    for i = 1:N
        y = y+h*yprime(t,y);
        t = t+h;
        liste_y = [liste_y;y];  // construction du vecteur y
        liste_t = [liste_t;t];  // construction du vecteur t
    end
endfunction
```

euler v3.sce

→ utilisation dans la console :

Si on appelle la fonction comme précédemment, sans précision ou avec un seul argument de sortie, elle renverra le premier, donc ici le temps :

```
--> euler(10,5,10)
 ans  =
   0.
   2.
   4.
   6.
   8.
   10.
```

Si l'on souhaite avoir accès aux deux arguments de sortie, il faut effectuer l'appel comme suit, en leur donnant des noms, placés entre crochets et séparés par une virgule (un tableau à N lignes et 2 colonnes), par exemple `t` pour le temps et `y` pour la solution :

```
--> [t,y]=euler(10,5,10)
 y  =

   10.
   3.3333333
   1.1111111
   0.3703704
   0.1234568
   0.0411523
 t  =

   0.
   2.
   4.
   6.
   8.
   10.
```

Est-ce bien un tableau ? Pour s'en convaincre, il suffit de rappeler `[t,y]` dans la console :

```
--> [t,y]
 ans  =
   0.    10.
   2.    3.3333333
   4.    1.1111111
   6.    0.3703704
   8.    0.1234568
   10.   0.0411523
```

c) Passage de fonction comme paramètre

À chaque nouvelle équation différentielle, le nom de la dérivée 1e devra être modifié. Il peut être plus commode d'entrer ce paramètre dans la fonction `euler`. Nommons `fprime` le paramètre désignant cette dérivée 1e et réécrivons le script :

```
// euler version finale
// paramètres :
// nom de la dérivée 1e = fprime
// valeur initiale = y0
// nombre de dates = N
// instant final = T
function [liste_t,liste_y]=euler(fprime,y0,N,T)
    y = y0;
    liste_y =[y0];
    t = 0;
    liste_t =[0];
    h = T/N;
    for i = 1:N
        y = y+h*fprime(t,y);
        t = t+h;
        liste_y = [liste_y;y];
        liste_t = [liste_t;t];
    end
endfunction
```

euler.sce

→ L'appel s'effectue alors par

```
--> [t,y]=euler(yprime,10,5,10)
```

2.3. tracé de la courbe

a) depuis la console

→ directement dans la console, à condition d'avoir les variables en mémoire :

```
--> plot2d(t,y)
```

Avec N=5, ce n'est pas extraordinaire…

b) avec un script

Écrivons un nouveau script pour le tracé ; nous en profiterons pour entrer les 4 paramètres par une invite à l'aide de la commande `input` :

```
// Tracé de la solution d'une équadiff par la méthode d'Euler
mode(0)        // Affichage des résultats dans la console (facultatif ici)
clear          // Réinitialisation de toutes les variables
xdel(winsid()) // Fermeture de toutes les figures
// Appel de l'équa diff
exec('P:\chemin…\équadiff yprime+ysurtau=0.sce');          // corriger le chemin
// Appel du module de résolution euler
exec('P:\chemin…\euler.sce');                               // corriger le chemin
// Paramètres
fprime = input('équadiff : nom de la dérivée première ? ');
y0 =  input('valeur initiale de la fonction ? ');
N =  input('nombre de dates pour la méthode d''Euler ? ');
T =  input('instant final (s) ? ');
// Résolution
[t,y] = euler(fprime,y0,N,T);
// Graphique
plot2d(t,y,style=2)
```

c) habillage graphique

Exécuter '*options_graphes.sce*'.
Alternative : ajouter simplement les lignes suivantes au script :

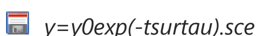
```
title ('Resolution par la methode d''Euler', 'fontsize',3)
xlabel ('temps t','fontsize',3)
ylabel ('y(t)','fontsize',3)
```

2.4. Test de la validité de la méthode

a) solution exacte

Dans le cas d'une équation différentielle non linéaire, il conviendrait ici de comparer le résultat obtenu par la méthode d'Euler avec le résultat fourni par la commande `ode` de Scilab. Dans le cas présent, la solution exacte est connue, on va donc faire plus simple. Définissons alors la fonction y_{exacte} notée `yex` :

```
// fonction y=y0exp(-t÷tau)
tau = 3;
function yex = yex(t)
    yex = y0*exp(-t/tau)
endfunction
```



b) <u>comparaison des tracés</u>

→ dans la console, avec `N=20` :

```
--> [t,y]=euler(yprime,10,20,10);
--> plot2d(t,y)
--> plot2d(t,yex(t), style=5)
--> legend ('Solution numerique','Solution exacte');
```

Conclusion : la solution numérique avec la méthode d'Euler a bonne allure, mais on voit qu'elle n'est qu'approximative.

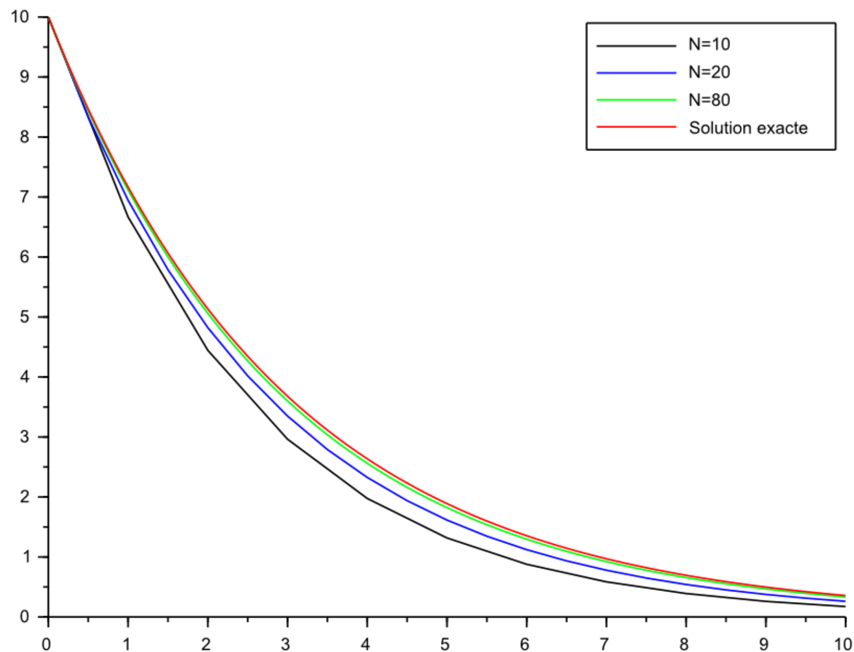
c) <u>impact du pas de discrétisation et du nombre d'itérations</u>

Reprenons ce tracé dans un script, en superposant les graphes obtenus avec `N=10`, `N=20`, `N=80` :

```
// on fait varier N
clf()          // efface la figure courante
[t,y]=euler(yprime,10,10,10);
plot2d(t,y)
[t,y]=euler(yprime,10,20,10);
plot2d(t,y,style=2)
[t,y]=euler(yprime,10,80,10);
plot2d(t,y,style=3)
plot2d(t,yex(t), style=5)
legend ('N=10','N=20','N=80','Solution exacte');
```

euler comparaison avec solution exacte.sce

Le résultat est clair : plus le pas est petit et donc plus le nombre d'itérations est grand, plus on tend vers la limite exacte.



Source principale : https://perso.univ-rennes1.fr/benjamin.boutin/planning_agregCS/Data/prg_scilab.pdf