TE 2 : Résolution numérique d'équations différentielles ordinaires (EDO)

Objectif On se propose de résoudre numériquement des équations différentielles en commençant par une équation scalaire du premier ordre, via des méthodes à un seul pas. L'objectif est d'analyser l'influence de l'ordre de la méthode et de la valeur du pas sur l'erreur commise. Dans un second temps, on convertira le code scalaire en vectoriel pour traiter un système d'équations différentielles couplées représentant une EDO d'ordre supérieur.

1 EDO scalaire du premier ordre

1.1 Position du problème

On recherche la solution de l'équation différentielle (1) avec la condition initiale $y(t_0) = y_0$. On suppose que $y_0 > 0$ et $a > t_0$.

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\frac{(t-a)\,y}{\tau^2} \tag{1}$$

Montrer que la solution analytique se met sous la forme (2) d'une gaussienne centrée en a et de largeur τ. Ne pas programmer directement l'expression (2), mais la reformuler pour limiter les erreurs d'arrondi et les dépassements de capacité.

$$y(t) = y_0 \exp\left[\frac{1}{2}\left(\frac{t_0-a}{\tau}\right)^2\right] \exp\left[-\frac{1}{2}\left(\frac{t-a}{\tau}\right)^2\right] \tag{2}$$

1.2 Organisation des codes

Le programme comporte quatre fichiers source principaux (voir schéma 1.2.1, p. 2) :

1. Un programme principal (fichier `main.cpp`) assurant le dialogue avec l'utilisateur pour choisir le pas d'intégration et la méthode. L'abscisse de début, de fin et la valeur initiale de la solution y_0 [1], qui ne varieront pas pendant le TE, seront définis au début du programme. `main` **définit** aussi les valeurs des paramètres (`a` et `tau` par exemple) des fonctions à tester. Ce programme crée alors le vecteur des abscisses et calcule les valeurs de la solution analytique en ces points. Il lance ensuite la boucle qui appelle le solveur choisi pour calculer de proche en proche la solution numérique. Enfin, il transmet à la procédure `ecrit` toutes les informations utiles pour écrire le fichier de résultats.
2. Un fichier `fcts.cpp` définissant les différentes fonctions second membre `dydt(t,y)` à intégrer, ainsi que les solutions analytiques `sol(t,t0,y_0)` éventuellement associées. C'est dans ce fichier que sont **déclarés** (mais non définis) les paramètres communs des fonctions (`a` et `tau` notamment) comme variables globales.
3. Un fichier `methodes.cpp` hébergeant les fonctions d'intégration des différents ordres qui permettent de progresser d'un pas, de t_i à t_{i+1}. Ces fonctions seront nommées selon la méthode d'intégration : `euler`, `milieu`, etc. Elles respecteront toutes le même prototype, par exemple pour `euler` :

   ```
   float euler(float u_i, float t_i, float h, function<float(float, float)> dydt)
   ```

   où `u_i` est la valeur de la solution numérique à l'instant `t_i`, `h` est le pas d'intégration, et `dydt` la fonction second membre à intégrer.
4. Un module utilitaire (fichier `util.cpp`) contenant la procédure `ecrit` chargée de créer le fichier formaté des résultats. Le nom du fichier sera choisi en fonction de la méthode (`euler.txt` par exemple signifiant méthode d'Euler). Le corps du fichier des résultats comportera une ligne par point de grille sous la forme :

abscisse	ordonnée numérique	ordonnée analytique	écart
t_i	u_i	$y(t_i)$	$u_i - y(t_i)$

   On spécifiera un format compatible avec la dynamique des valeurs et assurant une précision de l'ordre de 10^{-7}.
   Une fois la méthode mise au point, on ajoutera dans l'entête du fichier de résultats :
   - une première ligne avec le pas et la méthode utilisée,
   - et une deuxième ligne avec le maximum de la valeur absolue de l'écart et l'instant où cet écart est maximal.

1. Attention : en C++, `y0`, `y1` et `yn` sont des fonctions de Bessel, ainsi que `j0`, `j1` et `jn`. On évitera donc de nommer ainsi des variables.

1.2.1 Les quatre fichiers sources principaux

`main.cpp`
Choix des paramètres
(début `t0`, fin `tfin`, valeur initiale `y_0`, `a` et `tau`)
Choix du pas par l'utilisateur
Dans un deuxième temps, choix de la méthode `m` par l'utilisateur et détermination du nom du fichier de sortie
`...`
Création et remplissage des `Eigen::VectorXf t` des abscisses
et `exact` de la fonction analytique.
Boucle de calcul pas à pas

> Appel de la méthode d'intégration d'ordre `m` pour calculer u_{i+1}
> Remplissage du `Eigen::VectorXf estime` avec l'ordonnée estimée u_{i+1}.

`...`
Appel de `ecrit(t, estime, exact, nom_fich)`

`methodes.cpp`
Contient les 3 fonctions d'intégration de type
`nom_methode(u_i, t_i, h, dydt)`

1. Euler progressive
2. Point milieu
3. Runge Kutta d'ordre 4

La fonction `dydt` (de deux variables réelles et à valeur réelle) passée en argument des méthodes d'intégration sera déclarée sous la forme `function<float(float, float)> dydt` dans le prototype de la méthode.

`util.cpp`
Contient la procédure d'écriture du fichier
`ecrit(t, estime, exact, m) ...`

1. écriture du fichier des résultats
2. **dans un deuxième temps**, mise en place de l'entête.

`fcts.cpp`
Déclare (sans les définir) les paramètres communs des fonctions comme variables globales
Définit la fonction second membre
`dydt(t, y)`
`...`
et la solution analytique associée
`sol(t, t0, y_0)`
`...`

À ces fichiers principaux, viennent s'ajouter les fichiers d'entête (`.h`) pour le langage C++.

1.3 Graphiques sous `python`

Copier le script python du précédent TE dans le répertoire consacré à ce TE. Le faire évoluer afin de tracer, pour chaque méthode, un graphique comprenant les solutions analytique et numérique, et un autre graphique avec l'écart entre ces deux solutions. Exploiter ensuite l'entête lu comme un tableau de chaînes de caractères pour construire un titre identifiant la méthode et le pas. On sauvegardera les figures au format `pdf` directement sous `python` avec `plt.savefig("fig.pdf")`.

1.4 Plan de travail détaillé

Chaque module sera compilé séparément à l'aide de l'outil `make`.
On cherche à retrouver la solution gaussienne dans l'intervalle $[0,\ 20]$ avec $a = 4.$, $\tau = \sqrt{2.}$, $t_0 = 0$ et $y_0 = 0.5$

1. Rédiger le programme complet avec seulement la méthode d'Euler sans le calcul d'erreur. Visualiser les résultats pour un pas de 0.2
2. Reprendre l'étude avec un pas de 0.1 et commenter le rapport des erreurs.
3. Ajouter le calcul d'erreur et visualiser l'erreur [2] en fonction du temps. Contrôler l'erreur maximale en valeur absolue et sa position calculées par le programme.
4. Programmer progressivement les méthodes d'ordre supérieur et tester les pas de 0,2 et de 0,1. Produire les graphiques correspondant. Avec chaque méthode, calculer le rapport entre les erreurs maximales obtenues pour un pas de 0.2 et un pas de 0.1 Le comparer avec le rapport attendu en fonction de l'ordre de la méthode.
5. Rependre ces calculs entre les pas de 0.1 et 0.01 Avec quelle(s) méthode(s) obtient-on encore une amélioration en passant à un pas de 0.01, puis de 0.001 ?

2. Ne pas tracer sa valeur absolue : le signe de l'erreur est essentiel pour l'interprétation.

2 Mise en œuvre vectorielle des méthodes explicites à un pas

On souhaite maintenant intégrer numériquement l'équation différentielle ordinaire qui régit le mouvement du pendule : $y'' = -k^2 \sin y$. Cette EDO d'ordre 2 est à transformer en une EDO vectorielle du premier ordre avec `p=2` composantes, représentant la position et la vitesse du pendule.
On continue à noter `n` le nombre d'instants pour lequel est estimée la solution numérique.

2.1 Conseils techniques en C++

1. **Programme principal**
   Le nombre `p` de composantes, défini par le problème à traiter, sera initialisé en même temps que les autres paramètres, au début du programme principal. Les pas de temps restent associés à une variable de type `VectorXf`, comme dans la version scalaire. Par contre, les solutions analytique (`exact`) et numérique (`estime`) seront cette fois-ci des `MatrixXf`. Ces variables seront déclarées dans le programme principal après le choix du pas par l'utilisateur (qui fixe le nombre `n` de pas de temps). Pour l'efficacité numérique, il est préférable de consacrer la première dimension (lignes) au nombre de composantes, et la seconde (colonne) aux pas de temps.
   Comme dans le cas scalaire, les méthodes d'intégration ne font progresser que d'un pas de temps, de t_i à t_{i+1}. On trouvera ainsi dans la boucle d'intégration de l'EDO une instruction de la forme :

   ```
   // appel de la méthode du point milieu par exemple
    estime.col(i+1) = milieu(estime.col(i), t(i), h, dydt);
   // estime.col(i) et estime.col(i+1) : vecteurs à p composantes
   // t(i) : scalaire
   ```

   On notera que la condition initiale $\vec{\mathbf{y}}(t_0)$ est également **un vecteur de taille** `p`.
2. **Fonction second membre et solution analytique de l'équation différentielle**
   Ces fonctions calculent à l'instant t respectivement le second membre de l'EDO vectorielle $\overrightarrow{\mathbf{dy/dt}}(t, \vec{\mathbf{y}}(t))$, et la solution analytique pour une condition initiale donnée $\vec{\mathbf{y}}(t_0)$. Elles ont donc le même prototype. Celui de `dydt` est fourni à titre d'exemple :

   ```
   Eigen::VectorXf dydt(float t, Eigen::VectorXf y);
   ```

   On utilisera la fonction membre `size()` de la classe `VectorXf` pour déterminer automatiquement la taille du vecteur à renvoyer.
3. **Chacune des méthodes à un pas** est cette fois-ci une fonction retournant un `VectorXf`. Le prototype commun de ces fonctions est :

   ```
   //cas de la méthode d'Euler
   VectorXf euler(VectorXf u_i, float t_i, float h, function<VectorXf(float, VectorXf)> dydt);
   ```

   Les pentes locales $\overrightarrow{\mathbf{k_i}}$ seront des `VectorXf`, dont le nombre dépend de la méthode. On mettra à profit pour les calculer les opérations arithmétiques sur les `VectorXf`, ainsi que les opérations d'affectation, définies par la librairie `Eigen`.

2.2 Plan de travail pour les EDO vectorielles

1. Créer un répertoire spécifique pour implémenter les codes d'EDO vectorielles du premier ordre en gardant la même structure que pour le scalaire. Commencer par mettre au point le programme principal (`ppal.cpp`) et les fonctions de `fcts.cpp` avant d'aborder la partie résolution numérique.
2. Traiter d'abord le problème du pendule avec un second membre linéarisé $y'' = -k^2 y$.
   (a) Déterminer l'expression **générale** de la solution analytique en fonction des conditions initiales y_0 (position) et y'_0 (vitesse). Coder cette solution et la tracer. Choisir $k = 1$, $y_0 = 0$ et $y'_0 = 0.2$, et un pas de $h = 0.1$ On conservera $t_0 = 0$ et $t_f = 20$.
   (b) Écrire le système vectoriel du premier ordre de dimension 2 représentant l'EDO scalaire du second ordre. Implémenter la résolution numérique vectorielle. Calculer les deux composantes (position et vitesse) de l'écart avec la solution analytique et vérifier l'effet d'un changement de pas, par exemple $h = 0.01$
3. Compléter avec le second membre exact, non-linéaire.

(a) Comparer d'abord la solution numérique avec second membre non-linéaire pour des amplitudes très faibles, avec la solution analytique dans le cas linéaire.

(b) Tester ensuite l'évolution de la solution numérique quand y'_0 augmente. Repérer en particulier le passage en régime apériodique pour $|y'_0| > 2k$.

(c) Calculer l'énergie mécanique du système à chaque pas de temps et vérifier dans quelle mesure elle se conserve. On représentera l'écart relatif de l'énergie par rapport à la valeur initiale.