Chapitre 4 - Matlab et analyse numérique

IV

Dans ce chapitre nous abordons les notions centrales de l'analyse numérique. En effet, beaucoup de méthodes numériques conduisent à la résolution d'un système d'équations (linéaire ou non linéaire). Quelques méthodes de résolution des équations sont détaillées ici, ainsi que des méthodes d'intégration et manipulation des polynômes.

1. Fonctions "numériques"

Les fonctions numériques de Matlab généralisent les fonctions numériques usuelles, avec une différence cependant, elles sont vectorisées. C'est à dire qu'elles s'appliquent aussi bien à des nombres qu'à des tableaux. Lorsqu'une de ces fonctions à pour argument un tableau, la fonction est appliquée à chacun de ces éléments. Le résultat obtenu est donc un tableau du même format que le tableau donné comme argument.

2. Polynômes

Sur Matlab, les vecteurs ligne sont utilisés pour représenter les polynômes, en effet, un polynôme étant déterminé par ses coefficients, il suffit de stocker ceux-ci. Le coefficient du monôme de degré 0 dans la première coordonnée du vecteur et le coefficient du monôme de plus haut degré dans la dernière. Par exemple pour représenter le polynôme `P(x) = 2x² - 3x + 1` on utilise le vecteur `[2 -3 1]`.

2.1. Manipuler les polynômes

La fonction la plus utilisée sur les polynômes est évidemment celle qui permet d'évaluer la valeur d'un polynôme `P` pour une valeur `x` donnée. La fonction `polyval` prend comme arguments un polynôme `p` (sous forme d'un vecteur) et un nombre `x`, et fournit comme résultat la valeur `p(x)`.

Exemple : Évaluer un polynôme

Dans cet exemple nous allons évaluer la valeur du polynôme `P(x) = 2x² - 3x + 1` pour `x = 2`.

```
>> P = [2 -3 1];
>> polyval(P, 2)

ans =

     3
```

Complément

D'autres fonctions qui sont assez répandues dans Matlab pour manipuler les polynômes sont listées dans le tableau suivant :

Fonction	Opération à effectuer
`poly(r)`	Créer un polynôme à partir de ces racines (dans le vecteur `r` passé comme argument).
`polyfit(x, y, n)`	Retourner un polynôme `p` de degré `n` donnant la meilleurs approximation (en méthode des moindres carrés) pour les points définis par les vecteurs `x` et `y`.
`roots(p)`	Retourner les racines du polynôme `p` passé comme argument.
`conv(p, q)`	Effectuer une multiplication de `p` par `q`.
`deconv(p, q)`	Effectue une division euclidienne du polynôme `p` par `q`. Il est aussi possible de demander deux sorties à la fonction `deconv` pour calculer aussi le résidu de la division polynomiale. En d'autres termes, la commande `[u, v] = deconv (p, q)` donne deux polynômes `u` et `v` tel que : `p = conv(u, q) + v`.
`polyint(p)`	Retourne l'intégrale du polynôme `p` (en utilisant la valeur 0 comme constante d'intégration). Il est aussi possible de spécifier la constante d'intégration `k` en la passant comme deuxième argument de la fonction : (`polyint(p, k)`).
`polyder(p)`	Retourner le polynôme dérivé de `p`.

Tableau 4 : Les fonctions les plus utilisées sur les polynômes

Exemple

Nous montrons dans cet exemple quelques commandes effectuant des manipulations sur les polynômes

```
>> P = [1 -1];       % P(x) = x - 1.
>> Q = [-1 2];       % Q(x) = x - 1.
>> W = conv(P, Q)    % W(x) = P(x) * Q(x).

W =

    -1     3    -2

>> r = roots(W)      % Calculer les racines de W

r =

     2
     1

>> V = poly(r)       % créer un polynôme à partir de ces racines

V =

     1    -3     2
```

```
>> U = deconv(W, P) % U(x) = W(x) / P(x)

U =

    -1     2

>> A = polyder(V)   % Calculer le polynôme A(x) = V'(x)

A =

     2    -3

>> polyint(A)       % Calculer l'intégral de A(x)

ans =

     1    -3     0

>> polyint(A, 2)    % Calculer l'intégral de A(x) avec la constante d'intégration
2

ans =

     1    -3     2
```

3. Calcul matriciel

Matlab est spécialement conçu pour manipuler des matrices. Il reconnaît et manipule les variables matricielles à coefficients réels ou complexes, denses ou creuses. Nous avons déjà montré dans le premier chapitre comment initialiser, effectuer des calculs et appliquer des fonctions sur des matrices. Nous montrons dans cette section comment effectuer des calculs matriciels plus avancés en utilisant Matlab.

3.1. Les matrices creuses

On appelle matrice creuse une matrice comportant une forte proportion de coefficients nuls. L'intérêt de telles matrices résulte principalement de la réduction de la place mémoire (on ne stocke pas les zéros) et aussi de la réduction du nombre d'opérations (on n'effectuera pas les opérations portant sur les zéros). Par défaut dans MATLAB une matrice est considérée comme pleine, c'est-à-dire que tous ses coefficients sont mémorisés. Si `M` est une matrice, la commande `sparse(M)` permet d'obtenir la même matrice mais stockée sous la forme creuse. Si l'on a une matrice stockée sous la forme creuse, on peut obtenir la même matrice, stockée sous la forme ordinaire par la commande `full`.

Exemple : Avantage des matrices creuses

Dans cet exemple nous allons utiliser la commande `whos` pour avoir des informations sur l'espace mémoire occupé par chaque variable, puis nous comparons l'espace mémoire occupé par des matrices creuses stockées sous différents formats.

```
>> A = diag(1:10)

A =

     1     0     0     0     0     0     0     0     0     0
```

```
     0     2     0     0     0     0     0     0     0     0
     0     0     3     0     0     0     0     0     0     0
     0     0     0     4     0     0     0     0     0     0
     0     0     0     0     5     0     0     0     0     0
     0     0     0     0     0     6     0     0     0     0
     0     0     0     0     0     0     7     0     0     0
     0     0     0     0     0     0     0     8     0     0
     0     0     0     0     0     0     0     0     9     0
     0     0     0     0     0     0     0     0     0    10

>> B = sparse(A)

B =

   (1,1)         1
   (2,2)         2
   (3,3)         3
   (4,4)         4
   (5,5)         5
   (6,6)         6
   (7,7)         7
   (8,8)         8
   (9,9)         9
  (10,10)       10

>> infoA = whos('A'); % Récupérer des informations sur la variable A
>> infoA.bytes        % Afficher le nombre de bytes (octets) occupés par A

ans =

   800

>> infoB = whos('B'); % Récupérer des informations sur la variable B
>> infoB.bytes        % Afficher le nombre de bytes (octets) occupés par B

ans =

   164
```

3.2. Valeurs et vecteurs propres d'une matrice

Soit A une matrice carrée d'ordre n. On dit que x est un vecteur propre de la matrice A pour la valeur δ si Ax = x. Pour trouver les valeurs propres de la matrice A, il faut résoudre le système $Ax=\lambda x \iff (A-\lambda I_n)x=0$. Avec I_n une matrice identité de taille *n* (matrice carrée de taille *n* dont tous les éléments de la diagonale valent "1" et tous les autres éléments valent "0").

À l'exclusion de la solution triviale x = 0, le système ci-dessus admet des solutions si le déterminant de $(A-\lambda I_n)$ vaut 0.

Complément : Calculer le vecteur propre associé à chaque valeur propre

Un vecteur x est associé comme vecteur propre à la valeur λ s'il vérifie la relation $(A-\lambda I_n)x=0$.

Complément : Calculer les vecteurs et les valeurs propres sur Matlab

Pour calculer les valeurs propres d'une matrice carrée `A` nous utilisons la fonction `eig(A)` qui retourne un vecteur colonne contenant toutes les valeurs propres de `A`. Il est aussi possible de calculer les valeurs et les vecteurs propres de `A` en demandant deux sorties à la fonction `eig` (`[vec, val] = eig(A)`). la deuxième sortie de la fonction `eig` dans ce cas donne une matrice diagonale contenant les valeurs propre de `A`, alors que la première sortie donne une matrice telle que chaque colonne est un vecteur propres de `A`.

Plus formellement, la commande `[vec, val] = eig(A)` assure que `A * vec = vec * val`.

Exemple : Utilisation de la fonction eig

Nous montrons dans cet exemple comment utiliser la fonction `eig` de Matlab pour calculer les vecteurs et valeurs propres d'une matrice

```
>> A = [5 -3; 6 -4];
>> [vec, val] = eig(A)

vec =

    0.7071    0.4472
    0.7071    0.8944


val =

     2     0
     0    -1

>> A * vec

ans =

    1.4142   -0.4472
    1.4142   -0.8944

>> vec * val

ans =

    1.4142   -0.4472
    1.4142   -0.8944
```

3.3. Méthodes itératives pour la résolution des systèmes linéaires

Un système d'équations linéaires est un ensemble d'équations portant sur les mêmes inconnues. Nous considérons ici que les systèmes contenant autant d'équations que de variables. En général, un système de m équations linéaires à n inconnues peut être écrit sous la forme suivante :

$$\begin{cases} a_{11}x_1 - a_{12}x_2 + ... + a_{1n}x_n & = b_1 \\ a_{21}x_1 - a_{22}x_2 + ... + a_{2n}x_n & = b_2 \\ \cdots & . \\ \cdots & . \\ \cdots & . \\ a_{n1}x_1 - a_{n2}x_2 + ... + a_{nn}x_n & = b_n \end{cases}$$

Où $x_1, x_2 ... x_n$ sont des inconnus.

Un système d'équations linéaires peut aussi s'écrire sous la forme matricielle : $Ax = b$ tel que :

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} et\ x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Si la matrice A est inversible, il est possible de calculer $x = A^{-1} * b$. Plusieurs méthodes directes permettant de résoudre un système linéaire existent : Décomposition LU, Gauss (avec ou sans pivot), calculer l'inverse de A ... etc. Dans cette section nous nous intéressons plus aux méthodes itératives.

L'idée derrière les méthodes itératives est de construire une suite de vecteur $x^{(k)}$ qui converge vers le vecteur x (la solution du système $Ax = b$). Plus formellement $\lim_{k\to\infty} x^{(k)} = x$.

L'intérêt principal des méthodes itératives est d'être plus rapide en terme de temps de calcul (particulièrement dans les cas ou le point de départ x^0 est bien choisi). Nous présentons ici deux méthodes itératives pour la résolution des systèmes linéaires : la méthode de Jacobi et la méthode de Gauss-Seidel.

3.3.1. Méthode de Jacobi

Soit un système linéaire $Ax = b$. Si les éléments diagonaux de A sont non nuls (il est possible de permuter les lignes de A pour obtenir un diagonal qui ne comporte pas de zéro), alors le système est équivalent à :

$$x_i = \frac{1}{a_{ii}} \times \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j \right) \qquad (2)$$

Pour un vecteur $x^{(k)}$ connu, on peut calculer $x^{(k+1)}$ avec la formule suivante. $x^{(0)}$ est appelée "solution devinée", elle peut être donnée par l'utilisateur, initialisée par des valeurs aléatoires ou autre. Généralement, on arrête les itérations lorsque :

- $b - Ax^{(k)} < \varepsilon$, qui signifie que la solution $x^{(k)}$ est assez précise ;
- Ou $|x^{(k+1)} - x^{(k)}| < \varepsilon$, qui signifie que les solutions ne convergent plus.

Le schéma numérique de la méthode de Jacobi est le suivant :

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \times \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right) pour\ k \geq 0 \qquad (3)$$

Implémentation Matlab

Nous montrons ici l'implémentation Matlab d'une fonction permettant de résoudre un système d'équations linéaires donné sous sa forme matricielle. Pour simplifier l'implémentation, nous allons décomposer la matrice A en deux sous-matrices, la matrice diagonale D contenant les éléments diagonaux de A, et R = A - D. Dans ce cas, le système précédent est équivalent à :

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \ pour \ k \geq 0 \qquad (4)$$

```
% Fonction - jacobi.m
function x = jacobi(A, b, X0)
D = diag(diag(A));                % D prend les éléments diagonaux de A
invD = D^(-1);                    % nous calculons D^-1
R = A - D;                        % R contient les éléments non diagonaux de A
Xk = X0;
while norm(b - A * Xk) > eps      % condition d'arrêt
    Xkplus1 = invD * (b - R * Xk); % calculer Xkplus1 en fonction de k
    Xk = Xkplus1;
end;
x = Xk;
```

3.3.2. Méthode de Gauss-Seidel

La méthode de Gauss-Seidel est très similaire à celle de Jacobi, la seule différence est qu'elle propose d'introduire au fur et à mesure dans le calcul, sans attendre l'itération suivante, les composantes de $x^{(k+1)}$ qui viennent d'être calculées. Le schéma numérique de la méthode de Jacobi est le suivant :

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \times \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \ pour \ k \geq 0 \qquad (5)$$

L'algorithme de Gauss-Seidel est en général plus rapide que l'algorithme de Jacobi, et donc préférable. Par contre l'algorithme de Jacobi est plus adapté aux environnements parallèles.

4. Fonction d'une variable

Nous montrons dans ce chapitre les méthodes numériques pour le calcul d'intégrale, et pour la résolution des équations non linéaires. On ne considère dans cette section que les fonctions à une seule variable.

4.1. Calcul d'intégrale

Très souvent le calcul explicite de l'intégrale, d'une fonction *f* continue sur un intervalle [a, b], définie par : $I(f) = \int_a^b f(x)\ dx$ peut être très difficile à atteindre. Par conséquent, on fait appel à des méthodes numériques afin de calculer une approximation de *I(f)*. Nous présentons ici deux méthodes numériques pour le calcule d'intégrale : méthode du point milieu et méthode du trapèze.

4.1.1. Méthode du point milieu

Soit *f* une fonction continue sur l'intervalle *[a, b]*. L'idée de cette méthode est de subdiviser l'intervalle *[a, b]* en *n* sous intervalles $I_k = [x_k, x_{k+1}]$ tel que $x_1 = a$ et $x_{n+1} = b$. Le schéma numérique de la méthode du point milieu est le suivant :

$$I(f)=\sum_{k=1}^{n}(x_{k+1}-x_k)\times f(\bar{x}_k) \qquad avec \qquad \bar{x}_k=\frac{x_{k+1}-x_k}{2} \tag{6}$$

En effet, *I(f)* représente l'aire comprise entre la courbe *y = f(x)* et l'axe des abscisses entre les droites *x = a* et *x = b*. Alors que comme le montre la figure ci-dessous, $(x_{k+1}-x_k)\times f(\bar{x}_k)$ calcule l'aire entre la droite $y=f(\bar{x}_k)$ et l'axe des abscisses sur l'intervalle I_k. Plus les intervalles I_k sont réduits (et par conséquent nombreux), plus l'approximation du calcul d'intégrale est précise.

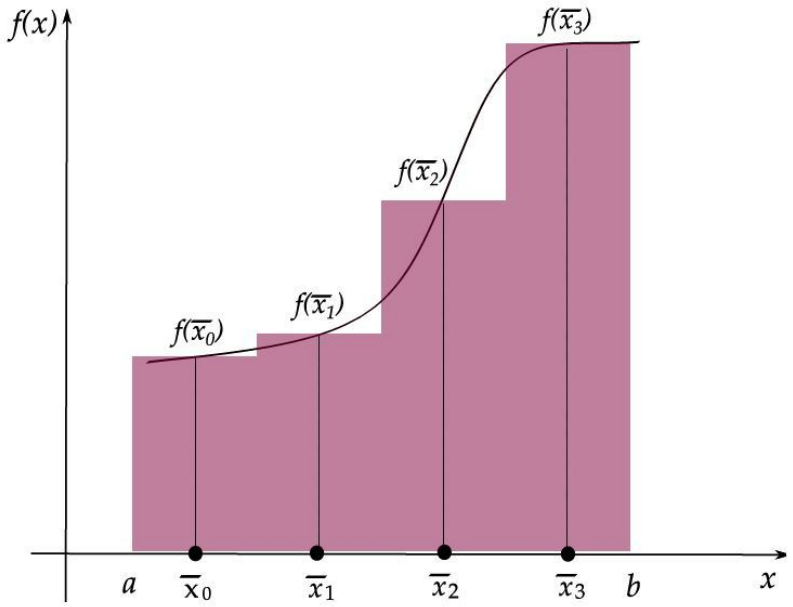

Figure 2 : Représentation de la méthode du point milieu

Si tous les intervalles I_k sont équidistants, le schéma numérique peut être simplifié comme suit :

$$I(f)=\Delta x\times\sum_{k=1}^{n}f(\bar{x}_k) \qquad avec \qquad \Delta x=\frac{b-a}{n} \tag{7}$$

Implémentation Matlab

Nous montrons ici l'implémentation Matlab d'une fonction permettant de calculer l'intégrale d'une autre fonction (passée comme paramètre) en utilisant la méthode du point milieu. Pour que notre calcul d'intégrale soit assez dynamique, il doit prendre en charge toutes les fonctions qui ont un seul paramètre `x`. pour cela, la fonction pour laquelle nous allons calculer l'intégrale est passée aussi comme paramètre, ainsi que les bornes de l'intervalle `a` et `b` (nous divisons l'intervalle *[a, b]* par la suite en 100 sous intervalles équidistants). La fonction est donnée sous forme d'une chaîne de caractères, puis elle est convertie à une fonction en appelant `str2func`.

```
% Fonction - pointMilieu.m
function res = pointMilieu(strFunc, a, b)
integFunc = str2func(strFunc);
Xk = linspace(a, b, 101);            % Générer tous les Xk distribués
uniformement sur l'intervalle [a, b]
dx = (b - a) / 100;                  % Calculer Δx
mp = (Xk(2:end) + Xk(1:end-1)) ./ 2; % Calculer les centres des sous intervalles
res = sum(integFunc(mp)) * dx;       % Calculer l'integrale
```

4.1.2. Méthode du trapèze

La méthode du trapèze consiste aussi à subdiviser l'intervalle *[a, b]* en *n* sous intervalles I_k. Cette méthode est basée sur l'interpolation linéaire de chaque intervalle. En d'autres termes, sur chaque intervalle $[x_k, x_{k+1}]$ est substitué par la droite joignant les points $(x_k, f(x_k))$ et $(x_{k+1}, f(x_{k+1}))$. Le schéma numérique de la méthode du trapèze est le suivant :

$$I(f)=\sum_{k=1}^{n}\left(\frac{f(x_k)+f(x_{k+1})}{2}\times(x_{k+1}-x_k)\right) \qquad (8)$$

La figure ci-dessous montre que $((f(x_{k+1})+f(x_k))/2)\times(x_{k+1}-x_k)$ calcule la surface de la trapézoïdale (d'où le nom trapèze) délimitée par la droite $((x_k,f(x_k)),(x_{k+1},f(x_{k+1})))$ et l'axe des abscisses sur l'intervalle I_k. Il est aussi possible d'augmenter la précision de la méthode du trapèze en réduisant la taille des intervalles I_k.

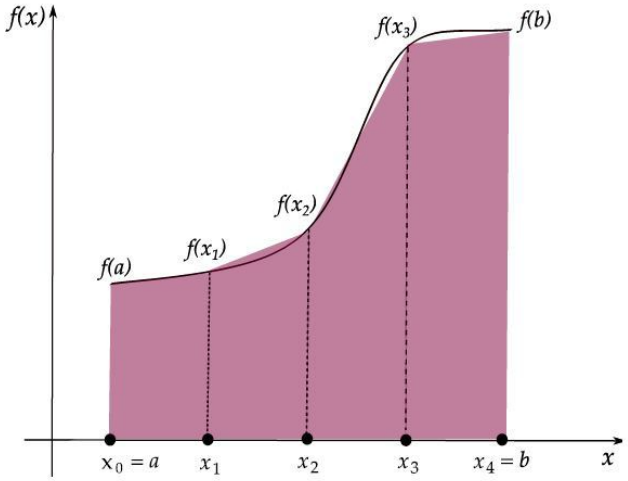

Figure 3 : Représentation de la méthode du trapèze

Si tous les intervalles I_k sont équidistants, le schéma numérique peut être simplifié comme suit :

$$\begin{aligned}
I(f) &= \Delta x\times\sum_{k=1}^{n}\frac{f(x_k)+f(x_{k+1})}{2} \qquad avec \qquad \Delta x=\frac{b-a}{n}\\
&= \frac{\Delta x}{2}\times\sum_{k=1}^{n}(f(x_k)+f(x_{k+1}))\\
&= \frac{\Delta x}{2}\times\left(\sum_{k=1}^{n}f(x_k)+\sum_{k=1}^{n}f(x_{k+1})\right)\\
&= \frac{\Delta x}{2}\times\left(\sum_{k=1}^{n}f(x_k)+\sum_{k=2}^{n+1}f(x_k)\right)\\
&= \frac{\Delta x}{2}\times\left(f(x_1)+\sum_{k=2}^{n}f(x_k)+\sum_{k=2}^{n}f(x_k)+f(x_{n+1})\right)\\
&= \frac{\Delta x}{2}\times\left(f(x_1)+f(x_{n+1})+2\sum_{k=2}^{n}f(x_k)\right)\\
&= \frac{\Delta x}{2}\times\left(f(a)+f(b)+2\sum_{k=2}^{n}f(x_k)\right)\\
&= \Delta x\times\left(\frac{f(a)+f(b)}{2}+\sum_{k=2}^{n}f(x_k)\right)
\end{aligned} \qquad (9)$$

La méthode du trapèze est prédéfinie dans l'environnement Matlab par la fonction `trapz`. Elle est utilisée selon la syntaxe suivante : `trapz(x, y)`. L'argument `x` de la fonction `trapz` contient les x_k qui définissent les subdivisions de l'intervalle sur lequel on va calculer l'intégral, alors que l'argument `y` contient les $f(x_k)$.

4.2. Résolution des équations non-linéaires

Plusieurs méthodes numériques permettent de calculer une approximation des zéros d'une fonction *f*. Deux méthodes sont illustrées dans cette section : méthode du point fixe et de la dichotomie.

4.2.1. Méthode du point fixe

Le principe de la méthode du point fixe est que de transformer l'équation $f(x) = 0$ à une équation $g(x) = x$ (généralement $g(x) = f(x) + x$). Donc, résoudre l'équation $f(x) = 0$ revient à trouver α tel que $g(\alpha) = \alpha$. Dans le cas ou α existe, il est qualifié de point fixe de la fonction $g(x)$ qui est appelée *fonction d'itération*. Lé schéma numérique de la méthode du point fixe est le suivant :

$$x^{(k+1)}=g(x^{(k)}) \qquad pour \qquad k>0 \tag{10}$$

Notez que $x^{(0)}$ est généralement choisi d'une façon aléatoire sur un intervalle *[a, b]*.

Implémentation Matlab

Nous donnons ici une fonction qui prend les trois paramètres d'entrées suivants :

- *f* : Le nom d'une fonction sous forme d'une chaîne de caractères ;
- *a* et *b* : les bornes de l'intervalle sur lequel on doit chercher la solution.

Et qui retourne comme résultat α tel que $f(\alpha) = \alpha$ en utilisant la méthode du point fixe.

```
% Fonction - pointFixe.m
function x = pointFixe(strFunc, a, b)
f = str2func(strFunc);
g = @(x) f(x) + x;          % Définir la fonction g(x) = f(x) + x
x = rand() * (b - a) + a; % Choisir un point du départ aléatoire
while x ~= g(x)           % Répeter x = g(x) jusqu'à trouver le point fixe
    x = g(x);
end
```

⚠ *Attention : Test d'égalité*

Il est important de noter que les ordinateurs n'utilisent pas explicitement les nombres réels. Ils utilisent plutôt les nombres flottants, qui sont un sous-ensemble discret de l'ensemble des nombres réels. Donc il est possible que α (tel que $g(\alpha) = \alpha$) n'appartient pas à ce sous-ensemble. Pour cela il est recommandé d'éviter les tests explicites d'égalité (ou d'inégalité), et de les remplacer par des tests qui sont assez léger pour être salifiables (et en même temps assez exigeant pour que l'algorithme reste correct).

Dans l'exemple du script précédent, il est recommandé de remplacer la condition `x ~= g(x)` par `(x - g(x)) / x < eps`.

4.2.2. Méthode de dichotomie

Le principe de cette méthode est basé sur le théorème des valeurs intermédiaires.

Rappel : Théorème des valeurs intermédiaires

Ce théorème implique que si une fonction *f* est continue sur un intervalle *[a, b]*, et que *f(a)* et *f(b)* ont différents signes (*f(a) * f(b) < 0*) alors .il existe un nombre α qui appartient à *[a, b]* tel que *f(α) = 0*.

La méthode de dichotomie suit les étapes suivantes :

- Calculer *c = (a + b) / 2*.
- Si *f(c) = 0* retourner *c* comme résultat.
- Sinon, si *signe(c) = signe(a)* (*a* et *c* ont le même signe), alors continuer le travaille récursivement sur l'intervalle *[c, b]*, sinon continuer sur l'intervalle *[a, c]*.

5. Exercices

5.1. Exercice : Évaluation d'un polynôme

Question

[solution n°17 p.77*]*

Écrire une fonction qui prend comme argument un polynôme `p` (sous forme d'un vecteur ligne) et une valeur `x` , et qui retourne comme résultat `p(x)` (vous n'avez pas le droit d'utiliser la fonction `polyval`).

Indice :

Utilisez `polyval` pour tester si les résultats fournis par votre fonction sont justes

5.2. Exercice : Dérivé d'un polynôme

Question

[solution n°18 p.77*]*

Écrire une fonction qui prend comme argument un polynôme `p`, et qui retourne comme résultat `q` le polynôme dérivé de `p` (vous n'avez pas le droit d'utiliser la fonction `polyder`).

Indice :

Utilisez `polyder` pour tester si les résultats fournis par votre fonction sont justes

5.3. Exercice : Méthode de Gauss-Seidel

Question

[solution n°19 p.77*]*

Écrire une fonction qui prend comme argument une matrice `A`, un vecteur `b`, et qui retourne comme résultat un vecteur `x` tel que `Ax = b` en utilisant la méthode de Gauss-Seidel. Il faut aussi passer comme argument la solution devinée `X0`.

5.4. Exercice : Méthode du trapèze

Question

[solution n°20 p.77*]*

Écrire une fonction qui prend les arguments :

- `f` : le nom d'une fonction sous forme d'une chaîne de caractères ;
- `a` et `b` : deux nombres.

Et qui retourne l'intégrale de la fonction *f* sur l'intervalle *[a, b]* en utilisant la méthode du trapèze (sans utiliser la fonction `trapz` de Matlab).

5.5. Exercice : Méthode de dichotomie

Question

[solution n°21 p.77*]*

Écrire une fonction qui prend les arguments :

- `f` : le nom d'une fonction sous forme d'une chaîne de caractères ;
- `a` et `b` : deux nombres tels que *f* est continue sur *[a, b]* et *f(a) * f(b) < 0*.

Et qui retourne un zéro de *f* sur *[a, b]*.