

**Exercice 1.** Soit  $f : \mathbb{R} \rightarrow \mathbb{R}$  une fonction numérique de classe  $C^\infty$  prenant des valeurs rationnelles aux points rationnels,  $\mathbf{a}$  et  $\mathbf{b}$  deux nombres rationnels, et  $(x_n)_{n \geq 1}$  la suite de nombres réels définie par  $x_1 = \mathbf{a}$ ,  $x_2 = \mathbf{b}$  et, pour tout  $n > 2$ ,

$$x_n = \begin{cases} x_{n-1} - f(x_{n-1}) \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} & \text{si } f(x_{n-1}) \neq f(x_{n-2}) \\ \frac{x_{n-1} + x_{n-2}}{2} & \text{sinon.} \end{cases}$$

1. Dites pourquoi la suite  $(x_n)_{n \geq 1}$  est une suite de nombres rationnels.

Ceci se voit par récurrence : en effet l'expression rationnelle de  $x_n$  en termes de  $x_{n-1}$  et  $x_{n-2}$  est une expression rationnelle à coefficients rationnels (dans les deux sous-cas de l'alternative proposée pour la définition de  $x_n$ ).

2. On suppose que  $f(\mathbf{a})f(\mathbf{b}) < 0$ . À quelle méthode algorithmique correspond la génération de cette suite  $(x_n)_{n \geq 1}$  ? Si cette suite converge vers un nombre réel  $\xi$  tel que  $f'(\xi) \neq 0$  et n'est pas stationnaire, pourquoi a-t-on  $f(\xi) = 0$  ? Quelle est l'ordre de convergence de cette méthode algorithmique (on pourra se contenter d'en donner une minoration) ?

La méthode algorithmique à laquelle correspond la génération de la suite  $(x_n)_{n \geq 1}$  est la méthode de la sécante (voir la section 2.1.3 du cours). On reconnaît en effet la formule inductive (2.2) sous-tendant cette méthode. Si la suite converge vers un nombre  $\xi$  tel que  $f'(\xi) \neq 0$ , il résulte de la formule des accroissements finis que

$$\lim_{n \rightarrow +\infty} \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} = \frac{1}{f'(\xi)}.$$

Le théorème de Rolle assure d'autre part que, puisque la suite  $(x_n)_{n \geq 1}$  n'est pas stationnaire, alors nécessairement  $f(x_n) \neq f(x_{n+1})$  pour  $n$  assez grand : si ce n'était pas le cas, on devrait avoir en effet, pour un tel  $n$ ,  $x_n = x_{n+1}$ , car, sinon, il y aurait (par Rolle) un zéro de  $f'$  entre  $x_n$  et  $x_{n+1}$ , ce qui est impossible du fait que la suite  $(x_n)_{n \geq 1}$  est censée

converger vers  $\xi$  tel que  $\mathbf{f}'(\xi) \neq 0$  et que  $\mathbf{f}'$  est continue en ce point  $\xi$ .  
 Pour  $n$  assez grand, on a donc

$$x_n - x_{n-1} = \mathbf{f}(x_{n-1}) \times \left( \frac{1}{\mathbf{f}'(\xi)} + o(1) \right) = o(1).$$

On en déduit que

$$\mathbf{f}(\xi) = \lim_{n \rightarrow +\infty} \mathbf{f}(x_{n-1}) = 0.$$

L'ordre de convergence de la méthode de la sécante est au moins égal au nombre d'or  $(1 + \sqrt{5})/2$  d'après le cours (Proposition 2.3) et les TP. En fait, il y a égalité.

3. Rédiger, en complétant le code **Maple** suivant, une procédure récursive qui calcule  $x_N$  pour  $N \geq 1$ ,  $\mathbf{a}, \mathbf{b}$  étant des rationnels fixés (la fonction  $\mathbf{f}$  ayant été préalablement déclarée sous **Maple**) :

```

algorithmme := proc(f,a,b,N) option remember;
local ... ;
if N=1 then
... ;
elif N=2 then
... ;
else
    if ... then
... ;
    else
... ;
... ;
... ;
    end if;
end if;
end proc;
    
```

*Expliquer pourquoi les deux boucles **if ... end if** sont essentielles. Que se passerait-il en particulier si l'on omettait la première d'entre elles ? Quels sont à la fois le sens et l'intérêt de l'instruction **option remember** ?*

Voilà comment compléter le synopsis proposé :

```

algorithmme := proc(f,a,b,N) option remember;
local u,v;
if N=1 then
    
```

```

a ;
elif N=2 then
b ;
else
u := algorithme (f,a,b,N-2);
v := algorithme (f,a,b,N-1);
    if f(u)=f(v) then
        (u+v)/2 ;
    else
        v - f(v) *(v-u)/(f(v)-f(u));
    end if;
end if;
end proc;

```

La première boucle `if ... end if` est essentielle car, si on l'omettait, la procédure récursive ne serait pas initialisée et bouclerait donc sur elle-même indéfiniment. La seconde boucle `if ... end if` est tout aussi essentielle, mais pour une autre raison : afin d'éviter cette fois une intempestive division par zéro lors de l'implémentation de la procédure (la procédure récursive générant la suite  $(x_n)_{n \geq 1}$  contient d'ailleurs dans sa définition même cette seconde boucle `if ... end if`). L'instruction `option remember` (dans une procédure récursive) permet de conserver en mémoire les résultats et donc de ne pas refaire un calcul déjà fait lors des appels successifs à la récursivité ; l'intérêt de cette instruction réside dans l'allègement qui en résulte concernant le temps d'exécution `time` du code (évalué en temps CPU).

## Exercice 2.

1. *Rappeler comment déclarer sous l'environnement **MATLAB** la fonction*

$$f : t \in \mathbb{R} \mapsto \frac{t^2 \times (\cos(t)) \times (\ln(t^2 + 1))}{2 + t^4 - (\sin(t))^2},$$

*de manière à cette fonction puisse aussi être directement évaluée aussi bien sur un vecteur ligne `t` que sur un simple scalaire `t`.*

On utilise pour cela la commande :

```

>> f =
inline ('(t.^2).*cos(t).*(log(t.^2+1))./(2+t.^4-(sin(t)).^2)','t');

```

2. *Rédiger un code (sous **MATLAB**) :*

```

function Im = trapezes (a,b,f,m) ;

```

qui, étant donnés deux réels  $a < b$  et un entier  $m \in \mathbb{N}^*$ , renvoie en sortie, une fois exécuté, l'intégrale approchée  $Im$  de  $f$  sur le segment  $[a, b]$ , calculée suivant la méthode des trapèzes composite, avec comme pas  $h_m = (b - a)/M$ , où  $M = 2^m$ . Que vaut (en vous reportant au cours) le plus grand entier  $k \in \mathbb{N}^*$  tel que l'on puisse affirmer :

$$\left| \int_a^b f(t) dt - Im \right| = O\left(\left(\frac{b-a}{2^m}\right)^k\right)$$

lorsque  $m$  croît vers l'infini ?

Il suffit d'utiliser sur chaque segment de la subdivision de  $[a, b]$  la formule (5.18), puis d'ajouter : les extrémités  $a$  et  $b$  sont prises en compte avec un facteur  $h_m/2$ , tandis que les  $2^m$  autres nœuds de la subdivision sont pris en compte avec un facteur  $h_m$ . Cela donne donc le code **MATLAB** suivant :

```
function Im = trapezes (a,b,f,m) ;
h = (b-a)/(2^m) ;
t = a:h:b ;
Im = h * ((f(t(1)) + f(t(2^m+1)))/2 + sum(f(t(2:2^m)))) ;
```

L'ordre de la méthode des trapèzes vaut  $p = 3$  (voir la formule (5.23) dans la sous-section 5.2.2 du cours). Lorsque cette méthode des trapèzes est utilisée de manière composite avec un pas  $h = (b - a)/M$ , l'erreur absolue commise entre l'intégrale exacte  $\int_a^b f(t) dt$  et sa version approchée est majorée en  $M \times O(((b - a)/M)^3) = O(((b - a)/M)^2)$ . Le plus grand entier  $k$  possible ici pour que l'assertion exigée soit valide est  $k = 2$ .

3. On admet (suivant en cela la formule d'Euler-MacLaurin mentionnée en cours), que, lorsque  $m \in \mathbb{N}^*$  croît vers l'infini,

$$Im = \int_a^b f(t) dt + \alpha_0[f] \left(\frac{b-a}{2^m}\right)^2 + O\left(\left(\frac{b-a}{2^m}\right)^4\right),$$

avec  $\alpha_0[f] = (f'(a) - f'(b))/12$ . Suivant le principe d'extrapolation de L. Richardson, modifier le code **trapezes** en un code :

```
ImBis = trapezes2 (a,b,f,m) ;
```

de manière à ce que, cette fois, on puisse affirmer :

$$\left| \int_a^b f(t) dt - ImBis \right| = O\left(\left(\frac{b-a}{2^m}\right)^4\right)$$

lorsque  $m$  croît vers l'infini.

4. L'idée est de combiner les deux formules :

$$\begin{aligned} \text{Im} &= \int_a^b f(t) dt + \alpha_0[f] h_m^2 + O(h_m^4) \\ \text{I}[m+1] &= \int_a^b f(t) dt + \alpha_0[f] h_{m+1}^2 + O(h_{m+1}^4) \\ &= \int_a^b f(t) dt + \alpha_0[f] \frac{h_m^2}{4} + O(h_m^4). \end{aligned}$$

En formant

$$\text{Imbis} = \frac{1}{3} \left( 4 \times \text{I}[m+1] - \text{Im} \right),$$

on obtient donc une approximation de l'intégrale exacte  $\int_a^b f(t) dt$  avec une erreur absolue en  $O(h_m^4)$  (au lieu de  $O(h_m^2)$  comme c'était le cas pour l'approximation par  $\text{Im}$ ). En termes de code **MATLAB**, cela s'écrit :

```
function Imbis = trapezes2 (a,b,f,m) ;
Im = trapezes(a,b,f,m);
Imaux = trapezes(a,b,f,m+1);
Imbis = (1/3)*(4*Imaux - Im);
```

5. Montrer que *Imbis* correspond au calcul approché de l'intégrale de  $f$  sur le segment  $[a, b]$ , calculée suivant cette fois la méthode de Simpson composite, avec comme pas toujours  $h_m = (b - a)/M$ , où  $M = 2^m$ .

On vérifie immédiatement que

$$\begin{aligned} \text{Imbis} &= \frac{h_m}{6} \left( f(a) + f(b) + 2 \sum_{j=1}^{M-1} f(a + j h_m) \right. \\ &\quad \left. + 4 \sum_{j=1}^M f\left(a + (2j-1) \frac{h_m}{2}\right) \right). \end{aligned}$$

Le membre de droite de cette formule correspond exactement à l'approximation de l'intégrale par la méthode de Simpson composite (méthode à 3 points avec précisément les pondérations  $L/6, 4L/6, L/6$ ,  $L$  désignant ici la longueur  $L = (b-a)/M$  de chaque segment de la subdivision).

**Exercice 3.** On rappelle (voir le cours d'Algèbre 2) que si  $A$  désigne une matrice réelle à  $m$  lignes et  $n$  colonnes et  $A'$  sa transposée, alors, pour tout vecteur colonne  $x \in \mathbb{R}^n$ , on a

$$\langle A' \cdot A \cdot x, x \rangle = \|A \cdot x\|^2, \quad (*)$$

où  $\| \cdot \|$  désigne la norme euclidienne sur  $\mathbb{R}^n$  et  $\langle \cdot, \cdot \rangle$  le produit scalaire usuel dans  $\mathbb{R}^n$ . Soit  $\mathbf{A}$  une telle matrice, avec de plus  $m \geq n$  et  $\text{rang}(\mathbf{A}) = n$ .

1. En utilisant (\*), vérifier que la matrice  $\mathbf{M} := \mathbf{A}' \cdot \mathbf{A}$  est une matrice symétrique réelle définie positive (c'est-à-dire dont toutes les valeurs propres sont strictement positives).

La relation (\*) implique que, si  $\mathbf{M} \cdot x = 0$ , alors  $\|\mathbf{A} \cdot x\| = 0$ . Comme le rang de  $\mathbf{A}$  est égal au nombre de colonnes  $n$ , l'endomorphisme de matrice  $\mathbf{A}$  relativement aux bases canoniques respectivement de  $\mathbb{R}^n$  et de  $\mathbb{R}^m$  est injectif. Si  $\mathbf{M} \cdot x = 0$ , on a donc  $x = 0$ . La matrice  $(n, n)$   $\mathbf{M}$  est donc inversible. Cette matrice est symétrique car

$$\mathbf{M}' = (\mathbf{A}' \cdot \mathbf{A})' = \mathbf{A}' \cdot \mathbf{A}'' = \mathbf{A}' \cdot \mathbf{A} = \mathbf{M}.$$

En tant que matrice symétrique réelle inversible,  $\mathbf{M}$  est diagonalisable et toutes ses valeurs propres  $\lambda$  sont réelles non nulles. Si  $\lambda$  est une telle valeur propre et  $v$  un vecteur propre (colonne) non nul associé à cette valeur propre, alors, il vient du fait de (\*) :

$$\langle \mathbf{M} \cdot v, v \rangle = \lambda \|v\|^2 = \|\mathbf{A} \cdot v\|^2 \geq 0.$$

On en déduit  $\lambda > 0$ . La matrice  $\mathbf{M}$  est donc bien symétrique définie positive.

2. Soit  $\mathbf{B}$  un vecteur colonne de  $\mathbb{R}^n$ . Écrire sous **MATLAB**<sup>1</sup> le code d'une procédure itérative (reposant sur le théorème du point fixe dans  $\mathbb{R}^n$ , au travers d'un algorithme vu en cours)

```
function XX = resolIterative(M,B,X,k) ;
```

qui, initiée au vecteur colonne  $\mathbf{X} \in \mathbb{R}^n$ , fournit au bout de  $k$  itérations une approximation  $\mathbf{XX}$  de la solution du système de Cramer  $\mathbf{M} \cdot \mathbf{XX} = \mathbf{B}$ .

D'après le cours (Proposition 4.5), l'algorithme de Gauß-Seidel converge lorsque la matrice  $\mathbf{M}$  est symétrique réelle définie positive. C'est donc la procédure itérative de Gauß-Seidel que l'on peut implémenter pour résoudre le système de Cramer  $\mathbf{M} \cdot \mathbf{XX} = \mathbf{B}$  de manière approchée. La syntaxe sous **MATLAB** de la procédure demandée ici est donc :

```
function XX = resolIterative(M,B,X,k);
T=tril(M);
F=tril(M)-M;
```

---

1. Dans cette question, comme dans la suivante, on pourra, faute d'exprimer le code sous la syntaxe du logiciel **MATLAB**, se contenter de le rédiger sous forme « pseudo-algorithmique », pourvu d'en sérier clairement les instructions.

```

XX=X;
for i=1:k
    XX=T^(-1)*F*XX+T^(-1)*B;
end

```

3. En y intégrant une boucle *while* (...) ~~&&~~ (...) ... *end*, modifier le code *resolIterative* rédigé à la question 2 en un code

```
function [XX,Niter] = resolIterative2 (M,B,X,epsilon,k);
```

de manière à ce que la procédure correspondante s'arrête automatiquement lors de son exécution dès qu'au terme d'un certain nombre d'itérations  $0 \leq Niter \leq k$ , on trouve pour la première fois

$$\|XX(Niter+1) - XX(Niter)\| \leq \epsilon,$$

où *epsilon* désigne un seuil strictement positif donné. On note ici  $XX(l)$ ,  $l = 0, \dots, k$ , l'état de la variable *XX* au terme de  $l$  itérations lors de l'exécution du code. Que signifie le fait de prendre *epsilon=eps* ?

Le travail demandé ici correspond à un travail fait nombre de fois en TP, soit sous Maple, soit (comme ici, cf. la feuille de TP 7) sous MATLAB. Voici la nouvelle routine :

```

function [XX,Niter] = resolIterative2 (M,B,X,epsilon,k);
T=tril(M);
F=tril(M)-M;
% on initialise XX et l'erreur err
% permettant le test d'erreur a venir :
XX=X;
err=2*epsilon;

% on initialise l'indice de comptage d'iterations Niter :
Niter=0;

while (err > epsilon) && (Niter<k)

    % on met en memoire le resultat de XX obtenu au terme des
    % iterations precedentes :
    XXold=XX;

    % on reactualise XX avec une nouvelle iteration :
    XX=T^(-1)*F*XX +T^(-1)*B;

```

```

% on calcule l'erreur entre ce nouvel XX et l'ancien,
% ce qui permet de reactualiser l'erreur :
err=norm(XX-XXold);

% on incremente l'indice de comptage
Niter=Niter+1;

end

```

Prendre `epsilon=eps` revient à décider que `epsilon` correspond à l'erreur machine ( $2^{-52}$  si l'on travaille en double précision). Ceci signifie que l'exécution du code s'arrête (si toutefois on y parvient en moins de `k` itérations) dès que la machine devient incapable de distinguer du point de vue numérique les états au cran `Niter` et au cran `Niter + 1` de la variable `XX`.