Présentation de Matlab

1. Introduction - Historique
2. Démarrage de MATLAB
3. Génération de graphique avec MATLAB

Systèmes d'équations linéaires

1. Matrices et Vecteurs dans MATLAB
2. Équations et systèmes linéaires dans MATLAB
3. Méthode directe (Méthode du pivot)
4. Méthodes itératives
   4.1. Méthode de Jacobi
   4.2. Méthode de Gauss-Seidel

Polynômes et interpolation polynomiale Résolution des équations non linéaires

1. Opérations sur les polynômes dans MATLAB
   1.1. Multiplication des polynômes
   1.2. Division des polynômes
2. Manipulation de fonctions polynomiales dans MATLAB
   2.1. Évaluation d'un polynôme
   2.2. Interpolation au sens des moindres carrés
3. Interpolation linéaire et non linéaire
4. Interpolation de Lagrange
5. Résolution d'équations et de Systèmes d'équations non Linéaire
   5.1. Résolution d'équations non Linéaires
   5.2. Résolution de Systèmes d'équations non Linéaires

Intégration numérique des fonctions

1. Introduction
2. Méthodes d'intégrations numériques
   2.1. Méthode des trapèzes
   2.2. Méthode de Simpson
3. Fonctions MATLAB utilisées pour l'intégration numérique

Résolution numérique des équations différentielles et des équations aux dérivées partielles

1. Introduction
2. Équations différentielles du premier ordre
3. Équations différentielles du second ordre
4. Méthode de Runge-Kutta
   4.1. Méthode de Runge-Kutta du second ordre
   4.2. Méthode de Runge-Kutta à l'ordre 4
5. Méthode Matricielle avec des "Conditions aux Limites"

Présentation de Matlab

1. Introduction - Historique

MATLAB est une abréviation de *Matrix LABoratory*. Écrit à l'origine, en Fortran, par *C. Moler*, MATLAB était destiné à faciliter l'accès au logiciel matriciel développé dans les projets LINPACK et EISPACK. La version actuelle, écrite en C par the MathWorks Inc., existe en version professionnelle et en version étudiant. Sa disponibilité est assurée sur plusieurs plates-formes : Sun, Bull, HP, IBM, compatibles PC (DOS, Unix ou Windows), Macintoch, iMac et plusieurs machines parallèles.

MATLAB est un environnement puissant, complet et facile à utiliser destiné au calcul scientifique. Il apporte aux ingénieurs, chercheurs et à tout scientifique un système interactif intégrant calcul numérique et visualisation. C'est un environnement performant, ouvert et programmable qui permet de remarquables gains de productivité et de créativité.

MATLAB est un environnement complet, ouvert et extensible pour le calcul et la visualisation. Il dispose de plusieurs centaines (voire milliers, selon les versions et les modules optionnels autour du noyeau Matlab) de fonctions mathématiques, scientifiques et techniques. L'approche matricielle de MATLAB permet de traiter les données sans aucune limitation de taille et de réaliser des calculs numériques et symboliques de façon fiable et rapide. Grâce aux fonctions graphiques de MATLAB, il devient très facile de modifier interactivement les différents paramètres des graphiques pour les adapter selon nos souhaits.

L'approche ouverte de MATLAB permet de construire un outil sur mesure. On peut inspecter le code source et les algorithmes des bibliothèques de fonctions (Toolboxes), modifier des fonctions existantes et ajouter d'autres.

MATLAB possède son propre langage, intuitif et naturel qui permet des gains de temps de CPU spectaculaires par rapport à des langages comme le C, le TurboPascal et le Fortran. Avec MATLAB, on peut faire des liaisons de façon dynamique, à des programmes C ou Fortran, échanger des données avec d'autres applications (via la DDE : MATLAB serveur ou client) ou utiliser MATLAB comme moteur d'analyse et de visualisation.

MATLAB comprend aussi un ensemble d'outils spécifiques à des domaines, appelés Toolboxes (ou Boîtes à Outils). Indispensables à la plupart des utilisateurs, les Boîtes à Outils sont des collections de fonctions qui étendent l'environnement MATLAB pour résoudre des catégories spécifiques de problèmes. Les domaines couverts sont très variés et comprennent notamment le traitement du signal, l'automatique, l'identification de systèmes, les réseaux de neurones, la logique floue, le calcul de structure, les statistiques, etc.

MATLAB fait également partie d'un ensemble d'outils intégrés dédiés au Traitement du Signal. En complément du noyau de calcul MATLAB, l'environnement comprend des modules optionnels qui sont parfaitement intégrés à l'ensemble :

1) une vaste gamme de bibliothèques de fonctions spécialisées (***Toolboxes***)
2) ***Simulink***, un environnement puissant de modélisation basée sur les ***schémas-blocs*** et de simulation de systèmes dynamiques linéaires et non linéaires
3) Des bibliothèques de blocs ***Simulink*** spécialisés (***Blocksets***)
4) D'autres modules dont un ***Compilateur***, un ***générateur*** de ***code C***, un ***accélérateur***,...
5) Un ensemble d'outils intégrés dédiés au Traitement du Signal : le ***DSP Workshop***.

Quelles sont les particularités de MATLAB ?

MATLAB permet le travail interactif soit en mode commande, soit en mode programmation ; tout en ayant toujours la possibilité de faire des visualisations graphiques. Considéré comme un des meilleurs langages de programmations (C ou Fortran), MATLAB possède les particularités suivantes par rapport à ces langages :

- la programmation facile,
- la continuité parmi les valeurs entières, réelles et complexes,
- la gamme étendue des nombres et leurs précisions,
- la bibliothèque mathématique très compréhensive,
- l'outil graphique qui inclus les fonctions d'interface graphique et les utilitaires,
- la possibilité de liaison avec les autres langages classiques de programmations (C ou Fortran).

Dans MATLAB, aucune déclaration n'est à effectuer sur les nombres. En effet, il n'existe pas de distinction entre les nombres entiers, les nombres réels, les nombres complexes et la simple ou double précision. Cette caractéristique rend le mode de programmation très facile et très rapide. En Fortran par exemple, une subroutine est presque nécessaire pour chaque variable simple ou double précision, entière, réelle ou complexe. Dans MATLAB, aucune nécessité n'est demandée pour la séparation de ces variables.

La bibliothèque des fonctions mathématiques dans MATLAB donne des analyses mathématiques très simples. En effet, l'utilisateur peut exécuter dans le mode commande n'importe quelle fonction mathématique se trouvant dans la bibliothèque sans avoir à recourir à la programmation.

Pour l'interface graphique, des représentations scientifiques et même artistiques des objets peuvent être créées sur l'écran en utilisant les expressions mathématiques. Les graphiques sur MATLAB sont simples et attirent l'attention des utilisateurs, vu les possibilités importantes offertes par ce logiciel.

MATLAB peut-il s'en passer de la nécessité de Fortran ou du C ?

La réponse est non. En effet, le Fortran ou le C sont des langages importants pour les calculs de haute performance qui nécessitent une grande mémoire et un temps de calcul très long. Sans

compilateur, les calculs sur MATLAB sont relativement lents par rapport au Fortran ou au C si les programmes comportent des boucles. Il est donc conseillé d'éviter les boucles, surtout si celles-ci est grande.

2. Démarrage de MATLAB

Pour lancer l'exécution de MATLAB :

- sous Windows, il faut cliquer sur Démarrage, ensuite Programme, ensuite MATLAB,
- sous d'autres systèmes, se référer au manuel d'installation.

L'invite '>>' de MATLAB doit alors apparaître, à la suite duquel on entrera les commandes.

La fonction "***quit***" permet de quitter MATLAB :

>>quit

La commande "***help***" permet de donner l'aide sur un problème donné.

Exemple :

>> help cos

COS Cosine.

COS(X) is the cosine of the elements of X.

Autres commandes :

- what : liste les fichiers *.m et *.mat dans le directory utilisé
- who : liste les variables utilisées dans l'espace courant
- ans : réponse retournée après exécution d'une commande

Exemple :

>>x=[1:5,1]
x =
1 2 3 4 5 1

ou bien :

>>[1:5,1]
ans =
1 2 3 4 5 1

clock : affiche l'année, le mois, le jour, l'heure, les minutes et les secondes.

>>clock
ans =
*1.0e+003 **
1.9980 0.0100 0.0180 0.0170 0.0020 0.0098

>>date
ans =
18-Oct-1998

<u>Calcul en mode Commande dans MATLAB :</u>

Soit à calculer le volume suivant : $V = \frac{4}{3}\pi R^3$ où R=4cm

Pour calculer V, on exécute les commandes suivantes :

>>R=4
R =
4

*>>V=4/3*pi*R^3*
V =
268.0826

(Ici, pi=π).

<u>Calcul arithmétique</u> :

+ ⇒ plus

- ⇒ moins

/ ⇒ division

* ⇒ multiplication

<u>Exemple :</u>

x=2, $P(x) = \frac{4x^2 - 2x + 3}{x^3 + 1}$

>>x=2
x =
2

*>>P=(4*x^2-2*x+3)/(x^3+1)*
P =
1.6667

Test 'if' ⇒ Ce test s'emploie, souvent, dans la plupart des programmes. Un test '***if***' est toujours suivi par un '***end***'.

Exemple :

>>V=268.0826
V =
268.0826

*>>if V>150, surface=pi*R^2, end*
surface =
50.2655

L'opérateur 'NON' ⇒ Il est noté (ou symbolisé) par '**~=**'

Exemple :

>>R=4
R =
4

*>>if R~=2, V=4/3*pi*R^3;end*

L'opérateur 'égal' (==) dans 'if' ⇒ Il est noté (ou symbolisé) par '**==**'.

Exemple :

>>R=4
R =
4

*>>if R==4, V=4/3*pi*R^3;end*

L'opérateur 'ou' ⇒ Il est noté (ou symbolisé) par '|'

Exemple : Si R=4 ou m=1, alors $V = \frac{4}{3}\pi R^3$

*>>if R==4 | m==1, V=4/3*pi*R^3;end*

Autres opérateurs :

> ⇒ supérieur à

< ⇒ inférieur à

>= ⇒ supérieur ou égal

<= ⇒ inférieur ou égal

> ⇒ supérieur à	.* ⇒produit élément par élément de matrices
< ⇒ inférieur à	.^ ⇒puissance élément par élément de matrices
>=⇒supérieur ou égal	./ ⇒division élément par élément de matrices
<=⇒inférieur ou égal	xor ⇒OU exclusif (XOR)
Error ⇒affiche le message : 'error' ⇒	message spécifié, émet un 'bip' et interrompt l'exécution du programme

Exemples :

Si g>2 ou g<0, alors a=4

>> if g>2 |g<0, a=4, end

Si a>3 et C<0, alors b=15

>>if a>3 & c<0, b=15, end

Les opérateurs '&' et '|' peuvent être utilisés dans la même chaîne :

>>if ((a==2 | b==3)&(c<5),g=1,end

L'opérateur 'if…….else…….elseif…….end':

Exemples :

>>R=2, if R>3, b=1 ; elseif R==3, b=2, else b=0, end

L'instruction 'elseif' peut être répétée dans un programme autant de fois.

Variables et noms de variables :

Les variables et les *noms de variables n'ont pas à être déclarés*, car dans MATLAB, il n'y a *aucune distinction* entre variable '*entière*', variable '*réelle*' ou variable '*complexe*'.

<u>Variables complexes</u> :

Traditionnellement, en *Fortran* les variables *i, j, k, l, m* et *n* sont réservées aux variables entières.

Dans MATLAB, *i* et *j* sont réservées aux unités imaginaires ($i = \sqrt{-1}$ ou $j = \sqrt{-1}$). Mais, on peut également les utiliser commes d'autres variales (entières ou réelles) si on les précise.

Dans le tableau ci-dessous, on dresse les noms de variables et les nombres spéciaux utilisés par MATLAB :

Nom de la variable	***Signification***	***Valeur***
eps	*précision relative des nombres réels (distance entre 1.0 et le nombre réel flottant le plus proche)*	$2.2204.10^{-16}$
pi	π	*3.14159……..*
i et j	*unités imaginaires*	$\sqrt{-1}$
inf	*nombre infini (1/0=inf)*	∞
NAN	*ce n'est pas un nombre : 0/0=NAN*	
date	*date*	
nargin	*nombre d'arguments d'appel transmis à une fonction*	
flops	*compteur opérations en virgule flottante*	
nargout	*nombre d'arguments de retour demandés à une fonction*	

Opérateurs 'for/end' et 'while/end'

<u>Exemples</u> :

*>>for R=1 :5, V=4/3*pi*R^3; disp([R,V]), end*

Dans ce cas, *R* varie de 1 à 5, et la commande "*disp([R,V])*" retourne la matrice suivante :
[R=1 :5,V (V(1) :V(5)]

On peut définir également l'instruction 'length' qui représente la taille de la variable. En effet, dans ce cas, on a :
length(R)=5 ; (R=1 :5) et length(R)-1=4 (4 intervalles de pas 1).

*>>while R<5, R=R+1 ; V=4/3*pi*R^3; disp([R,V]), end*

while exécute l'instruction qui suit tant que le test logique est vrai.

Exemple de pas dans la boucle for :

*>>for R=5 :-1 :1, V=4/3*pi*R^3; disp([R,V]), end*

Ici, le pas utilisé est dégressif (=-1). On peut utiliser les imbrications de '*for*' autant de fois que l'on souhaite.

Exemple :

*>> for i=0 :10, for j=1 :5, V=4/3*pi*R^3;disp([R,V]);end,end*

**** format :***

>>format → affiche les résultats avec 4 chiffres après la virgule.
>>format long → affiche les résultats avec 16 chiffres après la vrgule.

**** break :***

Interrope l'exécution d'un '*for*' ou d'un '*while*'

**** goto :***

Dans MATLAB l'instruction '*goto*' est remplacée par '*break*'.

Exemple :

```
while R==1
.
.
.
if x>Xlimite, break, end
.
.
.
.
end
```

**** clear :***

Efface toutes les variables existantes en mémoire

**** clc :***

Efface l'écran (fenêtre) de MATLAB

**** input(' ') :***

Introduire une ou des variables par l'intermédiaire du clavier.

Exemple :

>>z=input('Longueur L=');

retourne :
Longueur L=

>>y=input('Introduire votre nom','s');

Le second argument indique que la variable à introduire peut être un (ou des) caractère(s).

**** fprintf(' ')***

format de sortie sur écran

Exemples :

1. *fprintf('La longueur L=%f\n',L)*

Dans cette chaîne, on a :

% → pour la sortie de la variable,
f → format de sortie
\n → retour à la ligne
L → variable.

2. *fprintf('La longueur L= %e%f\n',L)*

le résultat affiché est avec des puissances de 10 (*exemple* : 2.5e04)

3. *fprintf('Nom_fichier','Volume= %e%12.5f\n',V)*

écrit la valeur = variable *V* dans un fichier Nom_fichier.
la variable *V* est avec 5 chiffres après la virgule.

Pour l'ouverture et la fermeture des fichiers, on peut utiliser également '*fopen*' et '*fclose*'. Ainsi, on peut joindre à l'ouverture de fichier par exemple, la permission d'ouvrir uniquement, de lire uniquement, de lire et écrire, etc.

Exemple :

>>fich=fopen('Nom_fichier',' ?')

? → représente la permission :

'w' → écriture : créer si nécessaire,

'r' → lecture uniquement (pas de création),

'+r' → lecture et écriture.

Ecriture d'un fichier ou de variables dans un fichier :

Ouvrir tout d'abord le fichier :

>>fich=fopen('Nom_fichier',' r') ; y=fscanf(fich,'%f')

La variable *y* est maintenant égale à la (ou les) variable(s) de *fich*.

Produit factoriel :

La fonction '***gamma***' donne le produit factoriel d'un nombre *n*.
Exemple : pour *n=6*, on a : *6!=6*5*4*3*2*1=720*

```
>>factorielle=gamma(6+1)
factorielle =
720
```

La fonction '***gamma***' peut calculer la factorielle des nombres entiers et même des nombres réels.

Utilisation de nombres ou de variables complexes :

MATLAB, peut utiliser et gérer les variables ou les nombres complexes. La plupart des fonctions implicites définies pour les réels existent pour les complexes, y compris la puissance.

Exemple :

>>z=3.5-1.25i ;

>>log(z)
ans =
1.3128 - 0.3430i

>>cos(2-i)
ans =
-0.6421 + 1.0686i

L'imaginaire pur est noté par *i* ou *j*. Ainsi, i^2 ou j^2 donne :

>>i^2
ans =
-1.0000 + 0.0000i

Soit *z* un nombre complexe. Son conjugué est donné par la fonction '*conj(z)*'.

>>z=3.5-1.25i
z =
3.5000 - 1.2500i

>>conj(z)
ans =
3.5000 + 1.2500i

Opérations sur les nombres complexes :

1. Addition de nombres complexes :

>>z1
z1 =
3.5000 - 1.2500i

>>z2
z2 =
1.3140 - 0.0948i

>>z1+z2
ans =
4.8140 - 1.3448i

2. Soustraction de nombres complexes :

>>z1-z2
ans =
2.186 - 1.1552i

3. Multiplication de nombres complexes :

*>>z1*z2*
ans =
4.4805 - 1.9743i

4. Division de nombres complexes :

>>z1/z2
ans =
2.7181 - 0.7551i

5. Opération de puissance :

>>z1^z2
ans =
4.5587 - 2.9557i

5. Module et argument d'un nombre complexe :

Dans MATLAB, les fonctions '*abs*' et '*angle*' permettent l'obtention directe du module et de l'argument d'un nombre complexe.

Exemple :

>>r=abs(z1)
r =
3.7165

>>theta=angle(z1)
theta =
-0.3430

L'angle theta est en radians.

Il est possible d'utiliser la notation exponentielle.

Exemple :

>>Z=exp(z1)

Z =
10.4420 -31.4261i

Comme on peut définir des tableaux ou des matrices de réels, il est aussi possible de définir des tableaux ou de matrices complexes avec MATLAB.

Exemples :

*>>Z=[1+2i 0 z1;1 z2 z1+z2;z1*z2 z1-z2 z1/z2]*

Z =
1.0000 + 2.0000i 0 3.5000 - 1.2500i
1.0000 1.3140 - 0.0948i 4.8140 - 1.3448i
4.4805 - 1.9744i 2.1860 - 1.1552i 2.7181 - 0.7551i

On peut également faire des opérations arithmétiques sur les matrices :

>>A=Z^(-1)
A =
0.2279 - 0.1872i -0.2475 + 0.1665i 0.1564 - 0.0282i
-0.6237 + 0.3429i 0.4146 - 0.4741i 0.0537 + 0.3314i
0.1251 - 0.0321i 0.1174 + 0.1358i -0.0282 - 0.0914i

*>>A*Z*
ans =
1.0000 + 0.0000i -0.0000 + 0.0000i -0.0000 + 0.0000i
0.0000 1.0000 - 0.0000i 0.0000 - 0.0000i
-0.0000 - 0.0000i 0.0000 - 0.0000i 1.0000 + 0.0000i

>>A/Z
ans =
0.1328 - 0.2826i -0.0267 + 0.2886i -0.0451 - 0.1223i
-0.1566 + 0.6725i 0.0057 - 0.5356i 0.1202 + 0.1689i
-0.1037 - 0.0857i 0.0965 + 0.0148i -0.0276 + 0.0428i

Déclarations et affectations des variables :

fprintf('caractère : %c\n',c) ; fprintf('entier: %d\n',i) ; fprintf('flottant: %f\n',f) ; fprintf('flottant: %s\n',s) ; Il existe également d'autres types d'affectations. Ces dernières sont analogues à celles du langage '*C*'.	c='#' i=1, 2, 3, … , n → des entiers $f = i / \sqrt{i}$ par exemple (si $i \geq 0$) s : la valur peut être la plus petite que posssible. Exemple : s=1.35e-17.

Arrangement de variables :

Une matrice d'ordre 0 est un scalaire. Une matrice d'ordre 1 est un vecteur. Ce vecteur peut être un vecteur ligne ou un vecteur colonne.

Exemples :

>>x=[0,1,2,3] → retourne un vecteur ligne :
x =0 1 2 3

ou encore :

>>x=[0 1 2 3] → retourne aussi un vecteur ligne.

x(2)=1 par exemple.

>>x=[0 ;1 ;2 ;3] → retourne un vecteur colonne.
x =
0
1
2
3

>>x=[0 ;1 ;2 ;3]' → retourne un vecteur ligne.

>>x=[0 1 2 3]' → retourne un vecteur colonne. x(1)=0, x(2)=1, x(4)=3.

Remarque :

Dans MATLAB, les numéros de lignes ou de colonnes ***commencent toujours par 1*** *et non par zéro* comme c'est le cas dans les autres langages de programmation. Exemple : *x(0)* n'existe pas.

Matrice d'ordre supérieur à 1 :

Une matrice d'ordre supérieur à 1 est une matrices à deux dimensions. Exemple : x(i,j).

Exemples :

>>x=[0 :2 ;4 :6] → retourne :
x =
0 1 2
4 5 6

C'est une matrice à 2 lignes et 3 colonnes.

>>y=[0 :2 ;4 :6]' → retourne la matrice transposée :
y =
0 4

1 5
2 6

La taille de la matrice y est donnée par la fonction '*size(y)*' :
>>size(y)
ans =
3 2
La réponse est : 3 lignes et 2 colonnes.

La colonne *j* de la matrice x est donnée par : y(:,j) → pour j=2, on a :
y(:,2)=
4
5
6

La ligne *i* de la matrice x est donnée par : y(i,:) → pour i=2, on a :
y(2,:)=
5

Pour une matrice carrée A d'ordre n, on a sa matrice identité qui est donnée par la fonction 'eye'.
Exemple : pour n=3, on a :

>>A
A =
1 2 3
4 5 6
6 7 8

>>eye(size(A))
ans =
1 0 0
0 1 0
0 0 1

eye(size(A)) donne la matrice identité de même ordre que celle de A.

<u>Aspect des nombres dans MATLAB :</u>

Dans MATLAB, il n'existe aucune distinction entre les nombres entiers, les nombres réels ou le nombres complexes. Les nombres sont donc traités automatiquement. La précision des calculs est affectée par le nombre de variables enregistrées et traitées.
Les paramètres essentiels qui déterminent cette précision dans un langage de programmation sont :

- le plus petit nombre positif : x_min
- le plus grand nombre positif : x_max

- l'erreur de la machine (epsilon) : eps

Le tableau ci-dessous donne une comparaison des différents paramètres obtenus par le langage Fortran et MATLAB :

Précision du langage	*MATLAB (sur station)*	*Fortran sur station Simple (double)*	*Fortran (Cray)*
x_min	4.5e-324	2.9e-39 (même)	4.6e-2476
x_max	9.9e+307	1.7e+38 (même)	5.4e+2465
eps	2.2ee-16	1.2e-7 (2.8e-17)	1.3e-29

Dans MATLAB, l'infini est représenté par '*inf*' (=∞). Si on exécute la commande *x=1/inf*, la réponse sera '*NAN*' c'est à dire : '*Not A Number*'.

Pour chercher x_min, x_max et eps, on peut utiliser les programmes suivants :

Cherche de x_min :
% Recherche de de x_min
x=1 ; while x>0, x=x/2, end

Cherche de x_max :
% Recherche de de x_max
x=1 ; while x<inf, x= x*2, end

Cherche de eps :
% Recherche de de eps
x=1 ; while x>0, x=x/2 ; ex=0.98*x+1,ex=ex-1, if ex>0,ex ;end,end

Fonctions Mathématiques dans MATLAB :

Fonctions trigonométriques	***Remarques***
sin(x)	
cos(x)	
tan(x)	
asin(x)	
acos(x)	
atan(x)	$-\pi/2 \le a\tan(x) \le \pi/2$

atan2(x,y)	$a\tan(x/y)$; $-\pi/2 \le a\tan 2(x,y) \le \pi/2$
sinh(x)	
cosh(x)	
tanh(x)	
asinh(x)	
acosh(x)	
atanh(x)	

Autres fonctions mathématiques *(élémentaires)*	***Remarques***
abs(x)	Valeur absolue de x
angle(x)	Valeur complexe de l'angle de phase : Si x=réel → angle=0 Si $x=\sqrt{-1}$ → angle=π/2
sqrt(x)	Racine carrée de x
real(x)	Partie réelle de la valeur complexe de x
imag(x)	Partie imaginaire de la valeur complexe de x
conj(x)	Complexe conjugué de x
round(x)	Arrondi entier de x
fix(x)	Arrondi par défaut du réel x
floor(x)	Arrondi au voisinage de $-\infty$ du réel x
ceil(x)	Arrondi au voisinage de $+\infty$ du réel x
sign(x)	=+1 six>0 ; =-1 si x<0
rem(x,y)	Le reste de la division : = x-y*fix(x/y)
exp(x)	Exponentielle (de base e)
log(x)	Log (de base e)
log10(x)	log (de base 10)

<u>Autres fonctions</u> :

Soit x=[2 15 0] une matrice ligne (vecteur ligne)

sort(x) → donne une matrice ligne dont les éléments sont en ordre croissant :

```
>>sort(x)
ans =
0 2 15
```

sort(x') → donne une matrice colonne dont les éléments sont en ordre croissant :

```
>>sort(x')
ans =
0
2
15
```

sum(x) calcule la somme des éléments de la matrice x.

```
>>sum(x)
ans =
17
```

```
>>sum([4 2 1 0;8 9 3 12])
ans =
12 11 4 12
```

Pour trouver le maximum et le minimum du vecteur x, on utilise les fonctions max(x) et min(x) :

```
>>max(x)
ans =
15
```

```
>>min(x)
ans =
0
```

pour une matrice y quelconque, le max(y) et min(y) donnent :

```
>>y
y =
4 2 1 0
8 9 3 12
```

```
>>max(y)
ans =
8 9 3 12
```

>>min(y)
ans =
4 2 1 0

Pour trouver la valeur moyenne d'un vecteur de la matrice x par exemple, on utilise la fonction '*mean(x,i)*'où i est le numéro de la colonne :

<u>*Exemple*</u> : si $x = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$

Alors mean(x,1) est :

>>mean(x,1)
ans =
1.5000 2.5000 3.5000

et mean(x,2) est :

>>mean(x,2)
ans =
1 4

**** <u>Nombres aléatoires</u> :***

Les nombres aléatoires peuvent être générés par la fonction '*rand*'. Son expression est '*rand(n)*' où *n* est le rang de la matrice.
Si n=3 par exemple, rand(3) va générer 3 lignes et 3 colonnes de la matrice dont les coefficients sont aléatoires et compris entre 0 et 1.

<u>*Exemple*</u> :

>> rand(3)
ans =
0.4154 0.0150 0.9901
0.3050 0.7680 0.7889
0.8744 0.9708 0.4387

On peut donner à la variable z une initialisation dans un programme. Pour cela, on peut générer des valeurs aléatoires grâce à la fonction '*seed*' (c'est à dire ensemencer) → *rand('seed',z)*

**** <u>Écriture de fonctions personnelles</u>*** :

Dans MATLAB, les programmes qui sont sauvegardés comme des fichiers_.m sont équivalentes à des sous-programmes et des fonctions dans d'autres langages.

Fonction retournant une seule variable :

Exemple : $$f(x) = \frac{2x^3 + 7x^2 + 3x + 1}{x^2 - 3x + 5.e^{-x}}$$

Supposons que le programme MATLAB correspondant est sauvegardé sous le nom demof_.m. ce programme est le suivant :

```
function y=demof_(x)
y=(2*x.^3+7*x.^2+3*x-1)/(x.^2-3*x+5*exp(-x)) ;
```

Pour déterminer f(x=3) par exemple, il suffit d'exécuter la commande : "*y=demof_(3),*et la réponse sera :

```
>>y=
502.1384
```

**** Fonction retournant plusieurs variables :***

Une fonction peut retourner plus qu'une variable. Considérons la fonction qui permet d'évaluer la moyenne et l'écart type de données.
Pour retourner les 2 variables, un vecteur est utilisé du côté gauche de la fonction, comme l'illustre le cas suivant :

```
function [mean, stdv]=mean_st(x)
n=length (x) ;
mean=sum(x)/n ;
stdv=sqrt(sum(x.^2)/n-mean.^2) ;
```

Exemple :

```
>>x=[1 5 3 4 6 5 8 9 2 4];
>>[m,s]=mean_st(x)
```

La réponse est : *m=4.7000* et *s=2.3685*

**** Fonction pouvant utiliser une autre*** :

Exemple : $$f_{av} = \frac{f(a) + 2f(b) + f(c)}{4}$$

Le sous programme de la fonction f_{av} est donné ci-dessous :

function wa=f_av(f_nom,a,b,c)
*wa=(feval(f_nom,a)+2*feval(f_nom,b)+feval(f_nom,c))/4 ;*
où *f_nom* est le nom de la fonction *f(x).*

la fonction '*feval*' est une commande de MATLAB qui évalue la fonction de nom *f_nom* et d'argument *x* . Exemple : *y=feval('sin',x)* est équivalente à *y=sin(x).*

**** Écriture d'un programme MATLAB*** :

En MATLAB, les programmes se terminent par une extension '*.m*' dans le nom du fichier programme. Aucune compilation n'est à faire avant l'exécution du programme. Au cours de l'exécution, un message d'erreur apparaît et indique les lieux où se trouvent les erreurs.
Pour lancer l'exécution du programme, il faut se mettre toujours dans le même répertoire où se trouve ce programme.

Exemple : ce dernier se trouve dans c:\utilisateur ; il faut changer tout d'abord de répertoire après avoir lancé MATLAB en tapant :
"*cd c:\utilisateur"*

Les fichiers de données sont enregistrés avec une extension '.mat' et les variables sont enregistrées en double précision.

3. Génération de graphique avec MATLAB

MATLAB est un outil très puissant et très convivial pour la gestion des graphiques, que ce soit en une dimension, en deux dimensions ou en trois dimensions. Pour tracer une courbe *y=sin(x)* par exemple, où x=0 :50 ; il suffit de faire :

>>x=0:50;y=sin(x);plot(x,y)

Ci-dessous, un petit résumé très succinct est donné pour tracer, concernant le traçage des graphiques et la manipulation des axes et des échelles :

- xlabel('temps') → pour donner un titre à l'axe x,
- ylabel('vitesse') → pour donner un titre à l'axe y,
- title('évolution de la vitesse')→ pour donner un titre au graphique,
- text(2,4,'+++Température T1') → au point $M\begin{pmatrix}2\\4\end{pmatrix}$, écrire la légende de la courbe tracée avec "+++",
- loglog(x,y) → tracer la courbe en échelle logarithmique (log-log),
- semilogx(t,f(t)) → tracer la courbe seulement en échelle logarithmique suivant x,
- semilogy(t,f(t)) → tracer la courbe seulement en échelle logarithmique suivant y,
- grid on → afficher le quadrillage dans le graphique,
- grid off → masquer le quadrillage dans le graphique,
- clf → effacer le graphique,

- close figure(i) → fermer (ou quitter) la figure (i),
- close all → fermer tous les graphiques ouverts,
- plot(x,y,x,z,x,w) → tracer y, z et w en fonction de x sur le même graphe,
- polar(x,y) → tracer la courbe y en fonction de x en coordonnées polaires,
- plot(x,y,'+g') → tracer y en fonction de x avec des marques '+' en couleur verte,
- fplot('f_nom',[x-mini, x-maxi]) → tracer la fonction f_nom selon les axes données (x),
- axis('square') → tracer un graphe géométriquement carré,
- axis('off') → masque les axes x et y,
- axis('on') → affiche les axes x et y,
- axis([x-mini, x-maxi, y-mini,y-maxi]) → affiche le graphique selon les limites données des axes x et y,
- hold('on') → traçage des courbes sur le même graphique à chaque fois qu'on exécute la fonction plot,
- hold('off') → traçage de chaque courbe sur un nouveau graphique à chaque fois qu'on exécute la fonction plot,
- plot3(x,y,z) → tracer z en fonction de x et de y en 3D,
- la fonction 'meshgrid' est utilisée pour créer un maillage 2D (x,y) ; si on se donne xa=[… : …] et ya=[… : …], ceci est très important sont des valeurs complexes→ [x,y]=meshgrid(xa,ya),

- mesh(xa,ya,z) → tracer le graphique en 3D de z,
- quiver(x,y,u,v,s) → tracer le champ vectoriel $\vec{V}\begin{pmatrix} u \\ v \end{pmatrix}$ dans le plan (x,y) où '*s*' est la taille de base à donner aux vecteurs,
- contour(x,y,z,'niveau') → tracer les lignes iso-valeurs de z dans le plan (x,y) où le niveau représente le nombre de lignes iso-courbes.

Exemple :x=0 :50 ; y=sin(x) ; z=cos(y) +4*x ; contour(x,y,z,40) → on obtient 40 iso-courbes.

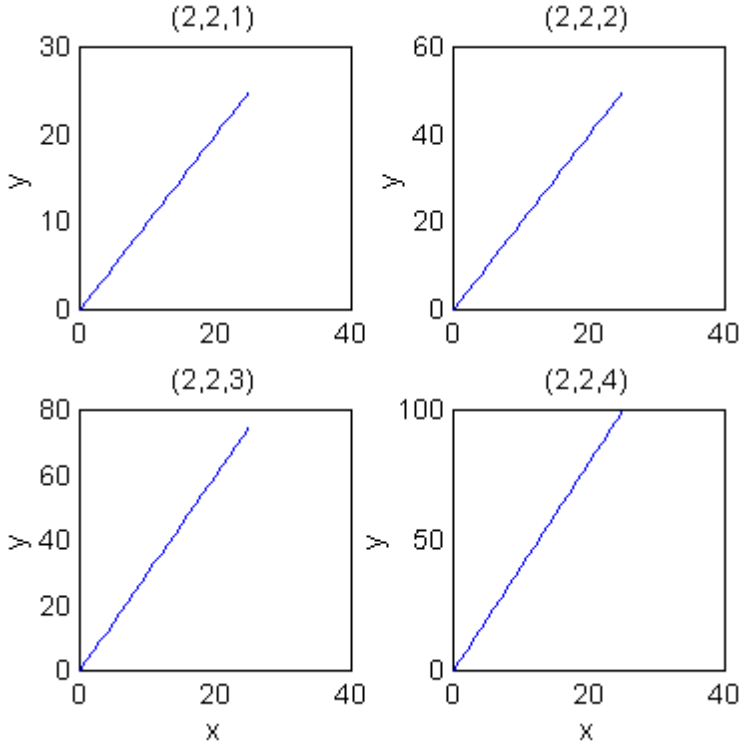
- text(2,0,'a-b','FontName','symbol') → écrire sur le graphique au point $M\begin{pmatrix} 2 \\ 0 \end{pmatrix}$

  α-β, (a-b en police symbol),

- subpot(m,n,k) → tracer *mxn* graphiques séparés dans la feuille, où *k* est le numéro de chaque graphiques. Exemple :

x=0 :25 ;y=x ;
subplot(2,2,1),plot(x,y) ;ylabel('y') ;title('(2,2,1)') ;
*subplot(2,2,2),plot(x,2*y) ;ylabel('y') ;title('(2,2,2)') ;*
*subplot(2,2,3),plot(x,3*y) ; ;xlabel('x') ;ylabel('y') ;title('(2,2,3)') ;*

*subplot(2,2,4),plot(x,4*y) ; ;xlabel('x') ;ylabel('y') ;title('(2,2,4)') ;*



Systèmes d'équations linéaires

1. Matrices et Vecteurs dans MATLAB

En mathématique, une matrice est constituée de *n* lignes et de *m* colonnes. Les coefficients de la matrice sont situés entre 2 parenthèses. Dans MATLAB, les parenthèses n'y figurent pas.

Une matrice carrée est une matrice dont le nombre de lignes est égal au nombre de colonnes (*n=m*). Une matrice ligne est une matrice d'ordre 1 appelée *vecteur ligne*. Une matrice colonne est une matrice d'ordre 1 appelée *vecteur colonne*. Une matrice d'ordre 0 est un scalaire.

Dans le cas général, une matrice de *n* lignes et de *m* colonnes a pour coefficients a_{ij}, où $0 \le i \le n$ et $0 \le j \le m$. Si cette matrice est appelée *A*, ses coefficients sont les suivant :

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1m} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2m} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3m} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nm} \end{pmatrix}$$

Si *B* est autre matrice à *p* lignes et *q* colonnes, alors le produit de *C=A*B* n'est possible que si *m=p*. Dans ce cas, le coefficient c_{11}, par exemple de cette matrice *C* s'écrit :

$c_{11} = a_{11}.b_{11} + a_{12}.b_{21} + \ldots + a_{1m}.b_{p1}$ (avec *m=p*).

Dans Matlab, la matrice produit *C* (= *A*B*) a pour coefficients c_{ij}. Ces coefficients sont calculés en fonction de a_{ij} et de b_{ij} :

$$c_{ij} = \sum a_{ik}.b_{kj}$$

Une matrice d'ordre zéro est une matrice à une ligne et une colonne : elle représente un scalaire. Une matrice *nulle* est une matrice dont les coefficients sont tous nuls. En attribuant le nom '*A*' à cette matrice, alors dans Matlab *A* s'écrit :

>>A=zeros(n,m)

où n et m sont respectivement le nombre de lignes et le nombre de colonnes.

La matrice *identité* est définie dans Matlab par la fonction : '*eye*'

Exemple :

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = IId$$

C'est une matrice carrée d'ordre 5. Dans Matlab, on peut l'écrire :

```
>>B=eye(5)
B =
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

Si B^t est la matrice transposée de *B*, cette dernière s'écrit dans Matlab :

>>C=B' → (l'appostrophe représente la transposée)

```
C =
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

Le déterminant d'une matrice carrée (*C* par exemple) est donné par :

```
>>det(C)
ans =
1
```

Soit A une matrice non nulle. La matrice inverse A^{-1} de *A* (si elle existe) est telle que :

$A*A^{-1}=IId$

Dans Matlab, cette matrice inverse est donnée par :

A^{-1}=A^(-1)=inv(A)=pinv(A)

Exemple :

>>A=[1 3 5;2 -1 0;5 4 3]

A =
1 3 5
2 -1 0
5 4 3

La matrice inverse de *A* est :

>>inv(A)
ans =
-0.0682 0.2500 0.1136
-0.1364 -0.5000 0.2273
0.2955 0.2500 -0.1591

ou encore :

>>pinv(A)
ans =
-0.0682 0.2500 0.1136
-0.1364 -0.5000 0.2273
0.2955 0.2500 -0.1591

2. Equations et systèmes linéaires dans MATLAB

Considérons le systèmes suivants de *n* équations à *m* inconnues :

$$\begin{cases} a_{11}.x_1 + a_{12}.x_2 + \ldots\ldots\ldots\ldots + a_{1m}.x_m = y_1 \\ a_{21}.x_1 + a_{22}.x_2 + \ldots\ldots\ldots\ldots + a_{2m}.x_m = y_2 \\ a_{31}.x_1 + a_{32}.x_2 + \ldots\ldots\ldots\ldots + a_{3m}.x_m = y_3 \\ a_{41}.x_1 + a_{42}.x_2 + \ldots\ldots\ldots\ldots + a_{4m}.x_m = y_4 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots = \ldots \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots = \ldots \\ a_{n1}.x_1 + a_{n2}.x_2 + \ldots\ldots\ldots\ldots + a_{nm}.x_m = y_m \end{cases} \Rightarrow A.x = y$$

Dans ce système, on a $A = \lfloor a_{ij} \rfloor$ connue, $y = y_i$ connues et $x = x_i$ inconnues.

- Si $n>m \Rightarrow$ système sur-déterminé,
- Si $n=m \Rightarrow$ système de Cramer → une solution unique (si $\det(A) \neq 0$),

- Si $n<m \Rightarrow$ système sous-déterminé

La solution numérique du système (pour $n=m$) donne les inconnues x_i :

x=A/y

Pour un vecteur colonne *y* donné ($y = \begin{pmatrix} 6 \\ 0 \\ -1 \end{pmatrix}$ par exemple), la solution par Matlab est :

```
>>x=A^(-1)*y
x =
-0.5227
-1.0455
1.9318
```

Notation :

Soit la matrice :

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & -1 & 0 \\ 5 & 4 & 3 \end{pmatrix}$$

Dans Matlab, la Matrice A peut être écrite sous les deux formes :

```
>>A=[1,3,5;2,-1,0;5,4,3];
```

ou bien :

```
>>A=[1 3 5;2 -1 0;5 4 3];
```

La séparation des lignes se fait par un point virgule et la séparation des colonnes se fait par un espace ou une virgule.

Exemple :

```
>>A=[1 3 5;2 -1 0;5 4 3]
A =
1 3 5
2 -1 0
5 4 3
```

Problèmes mal conditionnés :

Il existe un certain nombre d'équations linéaires solvables, mais leurs solutions sont incertaines à cause des erreurs d'arrondi lors des calculs. Les problèmes de ce type sont appelés *'problèmes mal conditionnés'*. L'arrondi durant les calculs ou les petits changements dans les coefficients calculés peuvent induire des erreurs significatives dans la résolution d'un problème (mal conditionné).

Cet effet d'arrondi, peut être illustré par le système des 2 équations suivantes, par exemple :

$$\begin{cases} 0{,}12065.x_1 + 0{,}98775.x_2 = 2{,}01045 \\ 0{,}12032.x_1 + 0{,}98755.x_2 = 2{,}00555 \end{cases}$$

La solution de ce système est :

$$\begin{cases} x_1 = 14{,}7403 \\ x_2 = 0{,}23942 \end{cases}$$

Pour monter l'erreur introduite sur les solutions en modifiant les coefficients (arrondi), on donne au coefficient *'2,01045'* de la première équation une légère augmentation de *'0,001'*.

Dans ce cas, le système devient :

$$\begin{cases} 0{,}12065.x_1 + 0{,}98775.x_2 = 2{,}01145 \\ 0{,}12032.x_1 + 0{,}98755.x_2 = 2{,}00555 \end{cases}$$

$$\begin{cases} x_1 = 17{,}9756 \\ x_2 = -0{,}15928 \end{cases}$$

Ceci met en évidence l'erreur engendrée par un arrondi.

La matrice *C* correspondant au système précédent est :

$$C = \begin{pmatrix} 0{,}12065 & 0{,}98775 \\ 0{,}12032 & 0{,}98755 \end{pmatrix}$$

La norme de *C* s'écrit :

$$\|C\| = \left(\sum_{ij} |c_{ij}|^2 \right)^{1/2} = \left(c_{11}^2 + c_{12}^2 + c_{21}^2 + c_{22}^2 \right)^{1/2} \Rightarrow \|C\| = 1{,}97993839$$

Dans Matlab on définit le *nombre-condition* de la matrice C [notation *cond(C)*] par :

$$cond(C) = \|C\|.\|C^{-1}\|$$

Ce nombre satisfait toujours la condition :

$$cond(C) \geq 1$$

Pour les matrices inconditionnées, ce *nombre-condition* attiend des grandes valeurs, mais ne donne pas une estimation directe de l'erreur sur la solution.

Dans Matlab, ce nombre se calcule par la fonction '*cond(C)*' où *C* est la matrice.

Exemple1 :

>>C
C =
0.1206 0.9878 0.9876

>>cond(C)
ans =
6.5598e+003

Exemple2 :

L'exemple de la matrice d'Hilbert représente un problème à matrice mal conditionnée. Cette matrice est définie par :

$A = \lfloor a_{ij} \rfloor$, où $a_{ij} = \frac{1}{i + j - 1}$

Programmer le nombre-condition (*cond(A)*) ainsi que $det(A).det(A^{-1})$ pour une matrice d'Hilbert de 5 par 5 à 14 par 14 :

Solution :

```
clear;
for n=5:14
for i=1:n
for j=1:n
a(i,j)=1/(i+j-1);
end
end
C=cond(a);
d=det(a)*det(a^(-1));
fprintf('n=%3.0f\t cond(a)=%e\t det*det=%e\n',n,C,d);
end
```

Après exécution du programme, on obtient les résultats suivants :

*n= 5 cond(a)=4.766073e+005 det*det=1.000000e+000*
*n= 6 cond(a)=1.495106e+007 det*det=1.000000e+000*
*n= 7 cond(a)=4.753674e+008 det*det=1.000000e+000*
*n= 8 cond(a)=1.525758e+010 det*det=9.999999e-001*
*n= 9 cond(a)=4.931532e+011 det*det=1.000001e+000*
*n= 10 cond(a)=1.602534e+013 det*det=9.999812e-001*
*n= 11 cond(a)=5.218389e+014 det*det=1.000119e+000*
*n= 12 cond(a)=1.768065e+016 det*det=1.015201e+000*
*n= 13 cond(a)=3.682278e+018 det*det=9.707419e-001*
*n= 14 cond(a)=1.557018e+018 det*det=-4.325761e+000*

Durant l'exécution du programme, un message s'affiche sur l'écran pour n=11 à n=14 : *Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 3.659249e-017.*

Ce qui signifie qu'à partir de *n=11* jusqu'à *n=14*, on a *det(a)*det(a^{-1})* différent de 1, donc les erreurs d'arrondi commencent leurs apparitions.

3. Méthode directe (Méthode du pivot)

Soit à résoudre le système suivant de 3 équations à 3 inconnues par la méthode du pivot (dans Matlab) :

$$\begin{pmatrix} -0,04 & 0,04 & 0,12 \\ 0,56 & -1,56 & 0,32 \\ -0,24 & 1,24 & -0,28 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix}$$

déterminer x_1, x_2, x_3.

Solution :

1. On définit tout d'abord la matrice argument dans Matlab par :

>>A=[-0.04 0.04 0.12 3;0.56 -1.56 0.32 1; -0.24 1.24 -0.28 0] ;

2. On choisit la ligne 1 comme ligne pivot : a(1,1)=-0.04,

on divise la ligne 1 par a(1,1) → b(1,:)=a(1,:)/a(1,1) → nouvelle ligne,

on annule le 1er terme de la ligne 2 et le 1er terme de la ligne 3 :

⋯ → *b(2,:)=a(2,:)-b(1,:)*a(2,1),*
⋯ → *b(3,:)=a(3,:)-b(1,:)*a(3,1).*

On obtient après calculs :

b =
1 -1 -3 -75
0 -1 2 43
0 1 -1 -18

3. On choisit la ligne 2 comme ligne pivot.

On divise cette ligne par *b(2,2)=-1*, on obtient :

⋯ → *c(2,:)=b(2,:)/b(2,2)* → nouvelle ligne = 0 1 -2 -43

on annule le terme *b(1,2)* de la ligne 1 et le terme b(3.2) de la ligne 3 :

⋯ → *c(1,:)=b(1,:)-c(2,:)*b(1,2)* → *1 0 -5 118,*
⋯ → *c(3,:)=b(3,:)-c(2,:)*b(3,2)* → *0 0 1 25.*

4. On choisit la ligne 3 comme ligne pivot c*(3, :).*

On divise cette ligne par *c(3,3)=1*, on obtient :

⋯ → *d(3,:)=c(3,:)* → nouvelle ligne = 0 0 1 25

on annule dans la ligne 1 c*(1,3)* et dans la ligne 2 *c(2,3)* :

⋯ → *d(1,:)=c(1,:)-d(3,:)*c(1,3)* → 1 0 0 7,
⋯ → d*(2,:)=c(2,:)-d(3,:)*c(2,3)* → 0 1 0 7.

D'où la matrice d s'écrit :

d =
1 0 0 7
0 1 0 7
0 0 1 25

et la solution est :

$$\begin{cases} x_1 = 7 \\ x_2 = 7 \\ x_3 = 25 \end{cases}$$

L'algorithme suivant (dans Matlab) permet de résoudre le système précédent :

```
clear %effacer toutes les variables en mémoire dans Matlab
a=[-0.04 0.04 0.12 3;0.56 -1.56 0.32 1;-0.24 1.24 -0.28 0];a
x=[0 0 0]'; %x est le vecteur colonne de composantes xi qui est
initialisé ici
% 1er point pivot ligne 1 -calculs sur les lignes 2 et 3
b(1,:)=a(1,:)/a(1,1);
b(2,:)=a(2,:)-b(1,:)*a(2,1);
b(3,:)=a(3,:)-b(1,:)*a(3,1);b
% 2ème pivot ligne 2 - calculs sur les lignes 1 et 3
c(2,:)=b(2,:)/b(2,2);
c(1,:)=b(1,:)-c(2,:)*b(1,2);
c(3,:)=b(3,:)-c(2,:)*b(3,2);c
% 3ème pivot ligne 3 - calculs sur les lignes 1 et 2
d(3,:)=c(3,:)/c(3,3);
d(1,:)=c(1,:)-d(3,:)*c(1,3);
d(2,:)=c(2,:)-d(3,:)*c(2,3);d
%Solutions recherchées
x(1)=d(1,4);
x(2)=d(2,4);
x(3)=d(3,4);x
```

après l'exécution de ce programme, on obtient :

a =
-0.0400 0.0400 0.1200 3.0000
0.5600 -1.5600 0.3200 1.0000
-0.2400 1.2400 -0.2800 0

b =
1.0000 -1.0000 -3.0000 -75.0000
0 -1.0000 2.0000 43.0000
0 1.0000 -1.0000 -18.0000

c =
1.0000 0 -5.0000 -118.0000
0 1.0000 -2.0000 -43.0000
0 0 1.0000 25.0000

d =
1.0000 0 0 7.0000
0 1.0000 0 7.0000
0 0 1.0000 25.0000

x =
7.0000
7.0000
25.0000

4. Méthodes itératives

4.1. Méthode de Jacobi

Résoudre le système suivant par la méthode itérative de Jacobi :

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ 2.x_1 - x_2 + 3.x_3 = 4 \\ 3.x_1 + 2.x_2 - 2.x_3 = -2 \end{cases}$$

Écrire un algorithme permettant de calculer x_1, x_2 et x_3 par itérations. Déterminer le nombre d'itérations nécessaires pour obtenir :

$$\varepsilon(i) = \left| x_{n+1}(i) - x_n(i) \right| = 10^{-10}$$.

Solution :

Appelons l'algorithme permettant de calculer la solution $\{x_i\}$ '*jacobi.m*'. Dans cet algorithme, nous avons utilisé un facteur de relaxation "λ" pour réaliser la convergence, car sinon le système diverge.

4.2. Méthode de Gauss-Seidel

Résoudre le même système précédent par la méthode de Gauss-Seidel. Écrire un algorithme (Matlab) permettant de résoudre ce système et déterminer le nombre d'itérations nécessaires pour obtenir une erreur $\varepsilon(i) = |x_{n+1}(i) - x_n(i)| = 10^{-10}$.

Comparer les résultats avec la méthode de Jacobi et conclure.

Représenter sur le même graphe l'évolution des solutions x_i en fonction du nombre d'itérations. Changer les valeurs initiales x_i^0, exécuter le programme puis conclure.

Solution :

* *Méthode itérative générale* :

Appelons l'algorithme permettant de calculer la solution $\{x_i\}$ '*G-seid.m*'
On se donne des valeurs arbitraires x_i^0 : par exemple x_i^0 =0. Le système à résoudre est : *A*x=y* ; où $A = \lfloor a_{ij} \rfloor$, $x = x_i$ et $y = y_i$

La matrice *A* et le vecteur colonne *y* sont connus, le vecteur colonne *x* reste inconnu. Ainsi, la méthode itérative générale consiste à écrire :

$$x(i) = \frac{y(i) - a(i,j).x(j)}{a(i,i)}$$ pour $i \neq j$ et $a(i,i) \neq 0$.

En particulier, pour un système de 3 équations à 3 inconnues, on a :

i=1 → j=2 et 3 $\Rightarrow$ $$x(1) = \frac{y(1) - a(1,2).x(2) - a(1,3).x(3)}{a(1,1)}$$

i=2 → j=1 et 3 $\Rightarrow$ $$x(2) = \frac{y(2) - a(2,1).x(1) - a(2,3).x(3)}{a(2,2)}$$

i=3 → j=1 et 2 $\Rightarrow x(3) = \frac{y(3) - a(3,1).x(1) - a(3,2).x(2)}{a(3,3)}$

et ceci pour chaque itération.

Programme *itmg.m* :

```
%*******************************
% Résolution d'un système linéaire      *
% par la méthode itérative générale    *
%*******************************

clear all;
clc;
fprintf('Méthode itérative générale\n');
n=30000;
a=[1 1 1;2 -1 3;3 2 -2];
y=[1 4 -2];
x=zeros(1,3);

w=0.2;  % facteur de relaxation : 0<w<1

epsilon=1e-10;
for k=1:n
   erreur=0;
   for i=1:3
      s=0;
      xb=x(i);
      for j=1:3
         if i~=j
            s=s+a(i,j)*x(j);
         end
      end
      x(i)=w*(y(i)-s)/a(i,i)+(1-w)*x(i);
      erreur=erreur+abs(x(i)-xb);
   end
      if (erreur/3<epsilon)
      fprintf('Itération no. : %d\t Erreur = %7.2e\n',k,erreur);
      break;
   end
end
x
```

Polynômes et interpolation polynomiale Résolution des équations non linéaires

L'objectif principal de l'interpolation est d'interpoler des données connues à partir des points discrets. Dans ce cas, la valeur de la fonction entre ces points peut être estimée. Cette méthode d'estimation peut être étendue et utilisée dans divers domaines ; à savoir la dérivation et l'intégration numérique des polynômes.

1. Opérations sur les polynômes dans MATLAB

Dans MATLAB, les polynômes sont représentés sous forme de vecteurs lignes dont les composantes sont données par ordre des *puissances décroissantes*. Un polynôme de degré *n* est représenté par un vecteur de taille (*n+1*).

Exemple :

Le polynôme : $f(x) = 8.x^5 + 2.x^3 - 3.x^2 + 4.x - 2$ est représenté par :

```
>>f=[8 0 2 -3 4 -2]
f =8 0 2 -3 4 -2
```

D'autres fonctions dans MATLAB telles que : '***conv***', '***deconv***', '***roots***', etc. peuvent être utilisées en plus des opérations propres aux vecteurs (*cf. chapitre précédent*).

1.1. Multiplication des polynômes

La fonction '***conv***' donne le produit de convolution de deux polynômes. L'exemple suivant montre l'utilisation de cette fonction.

Soient :

$$\begin{cases} f(x) = 3.x^3 + 2.x^2 - x + 4 \\ g(x) = 2.x^4 - 3.x^2 + 5.x - 1 \end{cases}$$

Le produit de convolution : $h(x) = f(x).g(x)$ est donné par :

>>f=[3 2 -1 4];

>>g=[2 0 -3 5 -1];

>>h=conv(f,g)
h =
6 4 -11 17 10 -19 21 -4

Ainsi, le polynôme $h(x)$ obtenu est :

$$h(x) = 6.x^7 + 4.x^6 - 11.x^5 + 17.x^4 + 10.x^3 - 19.x^2 + 21.x - 4$$

1.2. Division des polynômes

La fonction '***deconv***' donne le rapport de convolution de deux polynômes (déconvolution des coefficients du polynôme). L'exemple suivant montre l'utilisation de cette fonction.

Soient les mêmes fonctions précédentes $f(x)$ et $g(x)$:

$$\begin{cases} f(x) = 3.x^3 + 2.x^2 - x + 4 \\ g(x) = 2.x^4 - 3.x^2 + 5.x - 1 \end{cases}$$

La division de $g(x)$ par $f(x)$:

$h(x) = \dfrac{g(x)}{f(x)}$ est donnée par la fonction '*deconv*' :

>>f=[3 2 -1 4];

>>g=[2 0 -3 5 -1];

>>h=deconv(g,f)
h =
0.6667 -0.4444

et le polynôme $h(x)$ obtenu est :

$h(x) = 0,6667.x - 0,4444$

2. Manipulation de fonctions polynomiales dans MATLAB

Soit le polynôme suivant :

$$P(x) = a_n.x^n + a_{n-1}.x^{n-1} + \ldots + a_2.x^2 + a_1.x + a_0$$

où n est degré du polynôme et $a_i \ (i = 0,1,2,\ldots,n)$ sont les coefficients du polynôme.

Ce polynôme peut être écrit sous la forme :

$$P(x) = \left(\left(\ldots\left(\ldots\left(a_n.x + a_{n-1}\right).x + a_{n-2}\right).x \ldots + a_1\right).x + a_0\right)$$

Après factorisation, on a :

$$P(x) = a_n.(x - r_1)(x - r_2).(x - r_3)\ldots(x - r_n)$$

où $r_0, r_1, r_2, \ldots, r_n$ sont les racines du polynômes $P(x)$.

Exemple :

$$P(x) = x^4 + 2.x^3 - 7.x^2 - 8x + 12$$

Ce polynôme est équivalent à :

$$P(x) = ((((x + 2).x - 7).x - 8).x + 12)$$

ou encore :

$$P(x) = (x - 1)(x - 2)(x + 2)(x + 3)$$

Un polynôme d'ordre *n* possède *n* racines qui peuvent être réelles ou complexes.

Dans MATLAB, un polynôme est représenté par un vecteur contenant les coefficients dans un *ordre décroissant*.

Exemple :

Le polynôme : $P(x) = 2.x^3 + x^2 + 4x + 5$ qui est représenté dans MATLAB par :

```
>>P=[2 1 4 5];
```

a pour racines r_i. Pour trouver ces racines, on doit exécuter la fonction ***'roots'***.

D'où :

>>r=roots(P);

et le résultat donné est :

```
>> r
r =
0.2500 + 1.5612i
0.2500 - 1.5612i
-1.0000
```

Les trois racines de ce polynôme (dont 2 sont complexes) sont données sous forme d'un vecteur colonne. Quand les racines r_i sont connues, les coefficients peuvent être recalculés par la commande '*poly*'.

Exemple :

```
>>poly(r)
ans =
1.0000 0.5000 2.0000 2.5000
```

La fonction '***poly***' accepte aussi une matrice comme argument dont elle retourne le polynôme caractéristique.

Exemple :

>>A=[3 1;2 4];

```
>>p=poly(A)
p =
1 -7 10
```

Ainsi, le polynôme caractéristique de la matrice *A* est :

$$p(x) = x^2 - 7x + 10$$

Les racines de ce polynôme sont les valeurs propres de la matrice *A*. ces racines peuvent être obtenues par la fonction ***'eig'*** :

```
>>Val_prop=eig(A)
Val_prop =
2
```

2.1. Évaluation d'un polynôme

- Pour évaluer le polynôme $P(x)$ en un point donné, on doit utiliser la fonction '***polyval***'. On évalue ce polynôme pour *x=1*, par exemple :

```
>>polyval(P,1)
ans =
12
```

- *Exemple* :

On veut évaluer en *x=2.5* le polynôme suivant :

$$y = 3.x^4 - 7.x^3 + 2.x^2 + x + 1$$

```
>>C=[3 -7 2 1 1];

>>x=2.5;

>> y=polyval(C,x)
y =
23.8125
```

Si *x* est un vecteur contenant plusieurs valeurs, *y* sera aussi un vecteur qui contiendra le même nombre d'éléments que *x*.

2.2. Interpolation au sens des moindres carrés

Dans le domaine de l'analyse numérique des données, on a souvent besoin d'établir un modèle mathématique liant plusieurs séries de données expérimentales. L'interpolation polynomiale consiste à approcher la courbe liant les deux séries de mesures par un polynôme. Les coefficients optimaux de ce polynôme sont ceux qui minimisent la variance de l'erreur d'interpolation. Ce principe (voir cours) est connu sous le nom de la méthode des moindres carrés. La fonction ***'polyfit'*** retourne le polynôme *P* de degré *n* permettant d'approcher la courbe $y = f(x)$ au sens des moindres carrés.

Exemple :

>> x=[1.1 2.3 3.9 5.1];

>>y=[3.887 4.276 4.651 2.117];

>>a=polyfit(x,y,length(x)-1)
a =
-0.2015 1.4385 -2.7477 5.4370

Ainsi, le polynôme d'interpolation de *y* (d'ordre *length(x)-1=3*) est :

$$P(x) = -0{,}2015.x^3 + 1{,}4385.x^2 - 2{,}7477.x + 5{,}4370$$

Pour déduire l'erreur entre les valeurs expérimentales et le modèle obtenu par la fonction '***polyfit***', on dispose de la fonction '***polyval***' qui retourne la valeur du polynôme P pour toutes les composantes du vecteur (ou de la matrice) *x*.

Ainsi, cette fonction donne :

>>yi=polyval(a,x)
yi =
3.8870 4.2760 4.6510 2.1170

On remarque ici que les valeurs expérimentales de y sont bien restituées (*y=yi*). Dans ce cas, on a un coefficient de corrélation qui est égal à 1 (voir la définition de ce coefficient dans le cours).

Pour mieux comprendre ces fonctions prédéfinis dans MATLAB, on va simuler une courbe expérimentale par une sigmoïde à laquelle on superpose un bruit du type *Gaussien*. Cette courbe sera donnée par :

$$y = \frac{1}{1+e^{-x}} + 0{,}05.randn(1, length(x))$$

Le programme correspondant à l'approximation des ces données dans MATLAB (*regres.m*, par exemple) est le suivant :

```
%****************************************
% Génération de données expérimentales:   *
% y=1./(1+exp(-x))+0.05*randn(1,length(x))*
%****************************************

clc; % Effacer l'écran
clear all; % Effacer des variables de l'espace de travail
x=-5:0.1:5; % Intervalle de définition et de calcul de la sigmoïde

% Fonction sigmoïde bruitée
y=1./(1+exp(-x))+0.05*randn(1,length(x));
plot (x,y); % Tracé de la sigmoïde bruitée
title('Fonction sigmoïde bruitée - Polynôme d''interpolation');
xlabel('x');ylabel('y');

% Polynôme d'interpolation d'ordre 1
P=polyfit(x,y,1);

% Valeurs du polynôme d'interpolation
Vp=polyval(P,x);

% Tracé du polynôme d'interpolation
hold on;

plot(x,Vp,'--');
% Calcul de l'erreur d'interpolation
erreur=y-Vp;
% Tracé de la courbe de l'erreur
plot(x,erreur,':')
grid

gtext('Mesures')
gtext('Erreur')
gtext('Modèle')
hold off
% Affichage du polynôme d'interpolation
disp('Polynôme d''interpolation')
P
Var_erreur=num2str(std(erreur).^2);
disp(['La variance de l''erreur d''interpolation est : ',Var_erreur])
```

Après exécution du programme, on obtient à l'ordre 1 (droite affine : *fig. 1* ci-dessous) les résultats suivants :

>> regres
Polynôme d'interpolation
P =
0.1309 0.5008

La variance de l'erreur d'interpolation est : 0.011277

On remarque ici que le polynôme d'interpolation d'ordre 1 n'est pas une bonne approximation pour ces données. En effet, un polynôme d'ordre 5 donne une meilleure approximation de la sigmoïde (fig. 2). Ainsi, dans le programme précédent (*sigreg.m*), on change dans le '*1*' par '*5*' dans la fonction '***polyfit***', et on obtient les résultats suivants :

>> regres
Polynôme d'interpolation
P =
0.0002 -0.0000 -0.0111 0.0008 0.2326 0.4844

La variance de l'erreur d'interpolation est : 0.002279

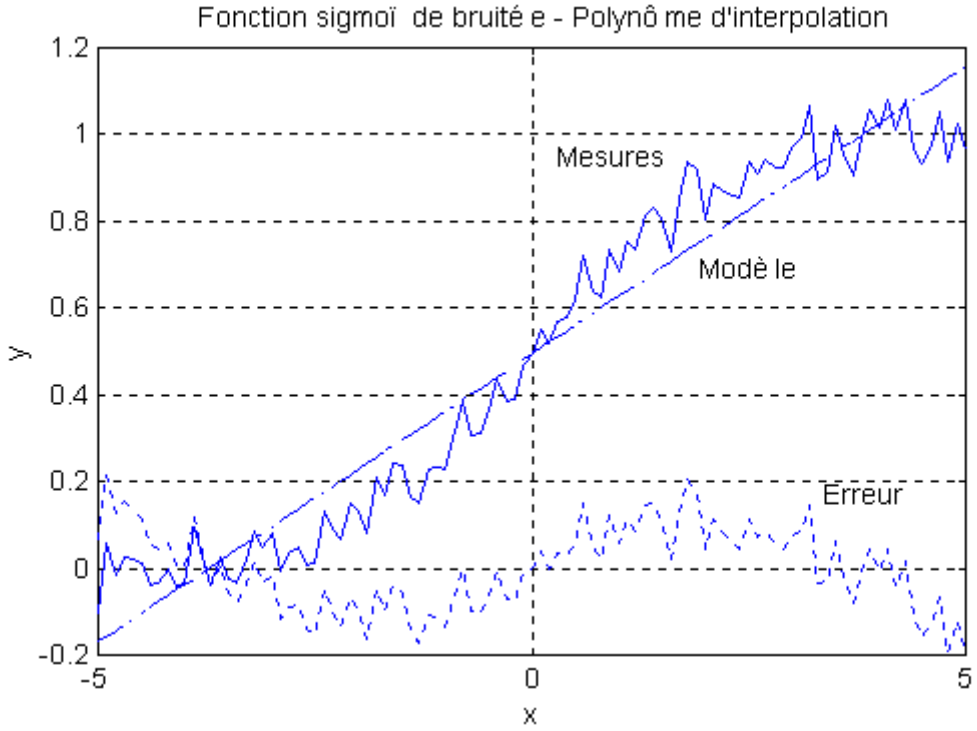


Fig.1 : Interpolation linéaire

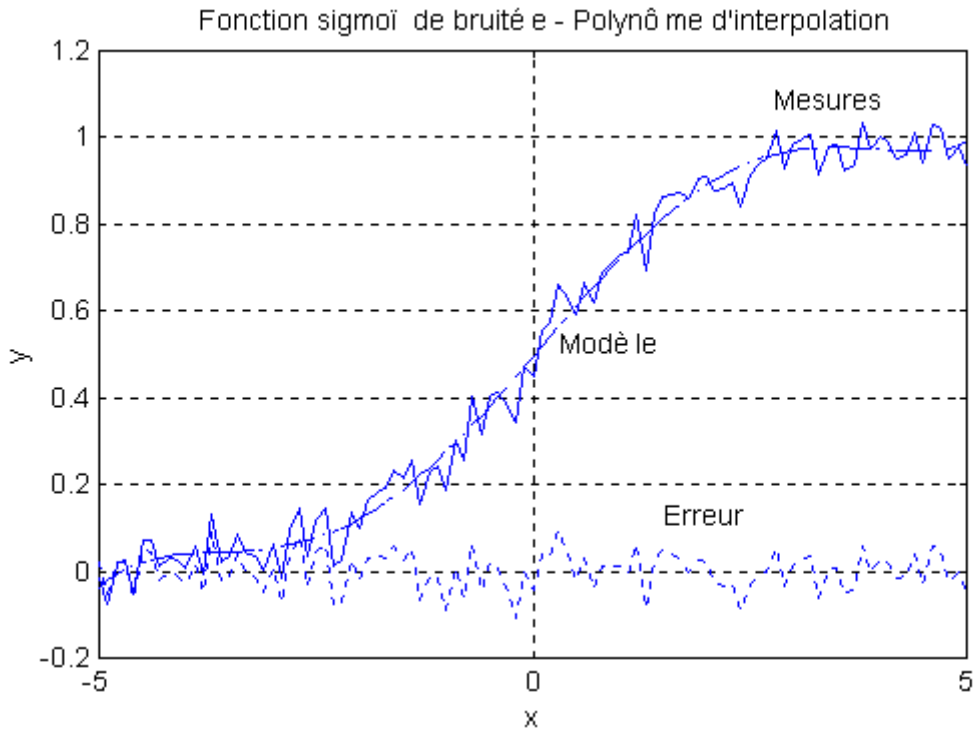


Fig.2 : Interpolation par un polynôme d'ordre 5

3. Interpolation linéaire et non linéaire

Une interpolation consiste à relier les points expérimentaux par une courbe sous forme de segments de droites ou de courbes polynomiales. Ceci peut être réalisé par la fonction '***interp1***'. La commande '***interp1(x,y,x_i,'type')***' retourne un vecteur de mêmes dimensions que x_i et dont les valeurs correspondent aux images des éléments de x_i déterminées par interpolation sur *x* et *y*. Si *f* est l'interpolation de *y*, la chaîne '***type***' spécifie alors le type d'interpolation qui doit être parmi les suivants :

- '*linear*' → interpolation linéaire
- '*spline*' → interpolation par splines cubiques,
- '*cubic*' → interpolation cubique.

Si on ne spécifie pas le type, l'interpolation linéaire est choisie par défaut.

Exemple 1 :

```
>>x = 0:10; y = sin(x); xi = 0:.25:10;
>>yi = interp1(x,y,xi,'cubic'); plot(x,y,'o',xi,yi)
```

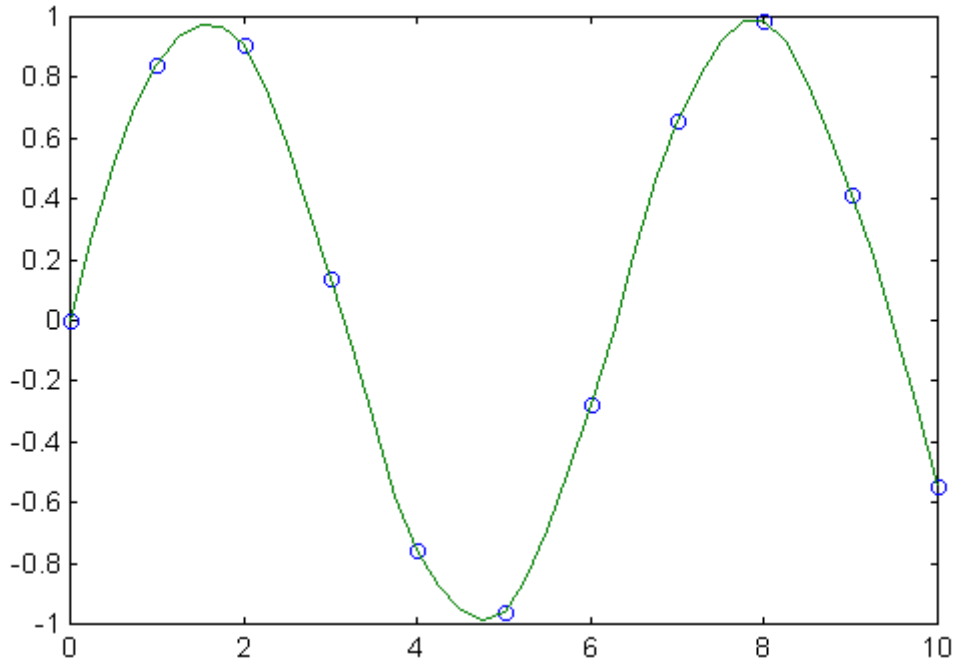


Fig. 3 : Interpolation cubique

Exemple 2 :

Dans le cas suivant, on étudiera ces différents types d'interpolation sur un même exemple de valeurs discrètes de la fonction '*cosinus*'. On appellera l'algorithme : '*interpol.m*'.

```
%************************************
% Utilisation de la fonction interp1 *
%************************************

clear all;
clc;
x=0:10;
y=cos(x); % Points à interpoler
z=0:0.25:10; % Le pas du vecteur z est inférieur à celui de x

% Interpolation linéaire
figure(1);
f=interp1(x,y,z);

% Tracé des valeurs réelles et de la courbe d'interpolation
plot(x,y,'*r',z,f);
grid on;
xlabel('Interpolation');

% Interpolation par splines cubiques
figure(2);
f=interp1(x,y,z,'spline');
plot(x,y,'*r',z,f);
grid on;
xlabel('Interpolation par splines cubiques');
```

En exécutant ce programme, on obtient les courbes suivantes :

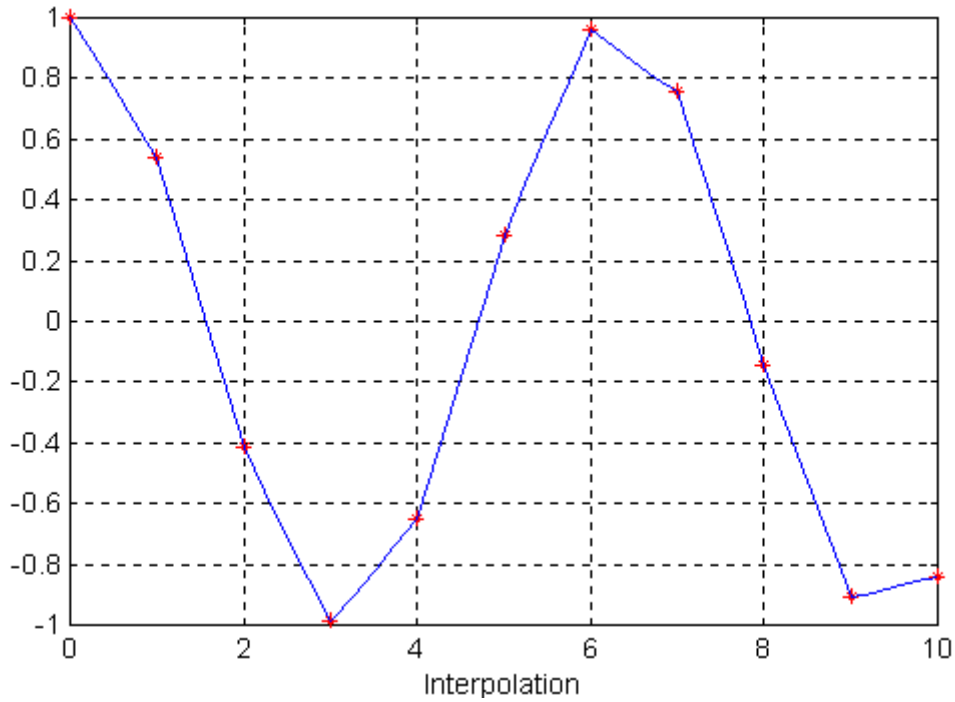


Fig. 4 : Interpolation linéaire

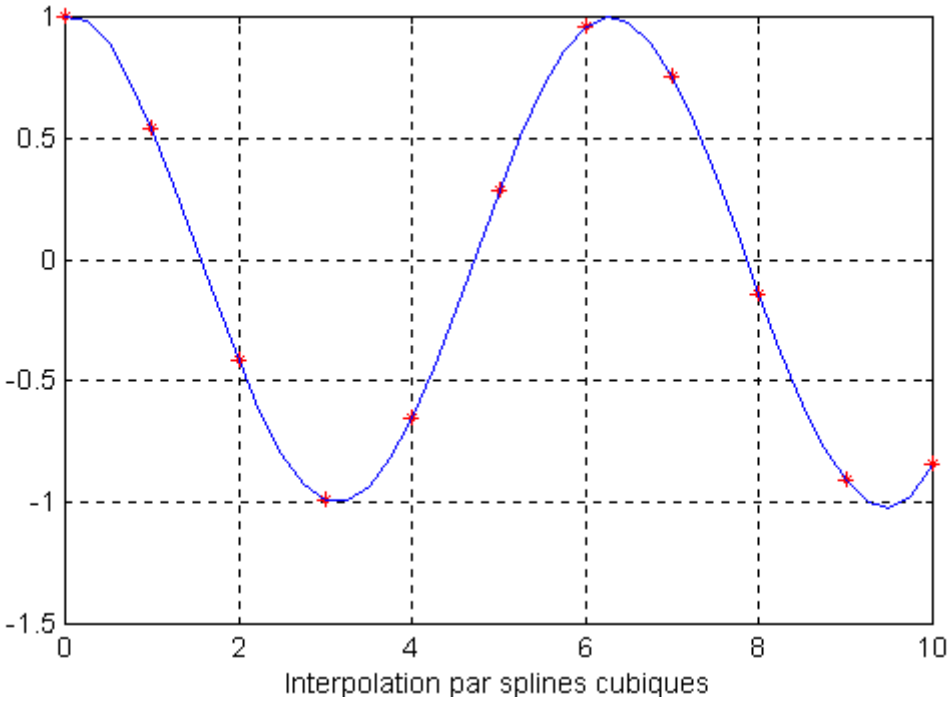


Fig. 5 : Interpolation par splines cubiques

La fonction '*interp2*' réalise l'interpolation dans l'espace trois dimensions (3D).

4. Interpolation de Lagrange

Exemple :

Les masses volumiques du matériau pour différentes températures sont données par le tableau ci-dessous :

i	*1*	*2*	*3*
Température T (en °C)	*94*	*205*	*371*
Masse volumique R(T) : (en kg/m^3)	*929*	*902*	*860*

- Écrire la formule d'interpolation de **Lagrange** qui permet d'interpoler les différents points de données précédentes.
- Trouver les masses volumiques du sodium pour *T=251 °C, T=305 °C* et *T=800 °C* en utilisant l'interpolation de Lagrange.

Solution :

a) Puisque le nombre de points est égal à 3, alors le polynôme de Lagrange sera de degré 2. Ce polynôme s'écrit :

$$R(T) = \sum_{i=1}^{3} R(T_i).\prod_{\substack{j=1 \\ j\neq i}}^{3} \frac{(T - T_j)}{(T_i - T_j)}$$

Soit :

$$R(T) = \frac{(T-205).(T-371)}{(94-205).(94-371)}.(929) + \frac{(T-94).(T-371)}{(205-94).(205-371)}.(902) + \frac{(T-94).(T-205)}{(371-94).(371-205)}.(860)$$

D'où pour $T=251\ °C$, on obtient $R(251) = 890.5\ kg/m^3$.

L'algorithme de calcul de *R(251)*, *R(305)* et *R(800)* est : '*polylag.m*' qui est listé ci-dessous :

```
%****************************************
% Interpolation polynomiale de Lagrange *
%****************************************
clc;
clear;
T=[94 205 371];
R=[929 902 860];
Ti=[251 305 800];
Ri=lag(T,R,Ti)
```

La fonction '*lag.m*' est un sous programme permettant de donner l'interpolation de Lagrange. La liste de ce sous programme est :

```
%***************************
% Sous-programme Lagran_.m *
%***************************
function Ri=lag(T,R,Ti)
Ri=zeros(size(Ti)); % Initialisation des Ti
n=length(R); % Nombre de points

for i=1:n
    y=ones(size(Ti));
    for j=1:n
        if i~=j
            y=y.*(Ti-T(j))/(T(i)-T(j));
        end
        Ri=Ri+y*R(i)
    end
end
return
```

Il suffit maintenant d'exécuter le programme '*polylag.m*' pour obtenir les résultats de *Ri*.

Ces résultats sont les suivants :

```
>>Ri
Ri =
890.5561 876.9316 742.45559
```

5. Résolution d'équations et de Systèmes d'équations non Linéaire

5.1. Résolution d'équations non Linéaires

Dans cette partie, on s'intéressera à la méthode de *Newton-Raphson* pour la résolution des équations non linéaires à une ou à plusieurs variables. On cherchera ensuite à trouver la valeur x^* qui annulera la fonction $f(x)$.

La méthode de *Newton-Raphson* permet d'approcher par itérations la valeur x^* au moyen de la relation suivante :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Dans toutes les méthodes itératives, il est nécessaire pour éviter une divergence de la solution, de bien choisir la valeur initiale x_0. Celle-ci peut être obtenue graphiquement.

On se propose d'appliquer cette méthode pour la recherche des racines de la fonction non linéaire suivante :

$$f(x) = e^x - 2.\cos(x)$$

Dans un premier temps, on se propose de tracer la courbe représentative de cette fonction en utilisant le programme ci-dessous 'p*num1.m*':

```
%************************
% Etude de la fonction : *
% f(x)=exp(x)-2.cos(x)   *
%************************

x=-1:0.1:1;
f=exp(x)-2*cos(x);
plot(x,f); grid on;
title('Fonction : f(x)=exp(x)-2.cos(x)');
```

Après exécution du programme, on obtient la courbe sur la figure 6. D 'après cette courbe, il est judicieux de choisir un $x_0 = 0,5$; car $f(0,5)$ est proche de zéro pour avoir une convergence rapide.

La fonction dérivée $f'(x)$ a pour expression : $f'(x) = e^x + 2.\sin(x)$.

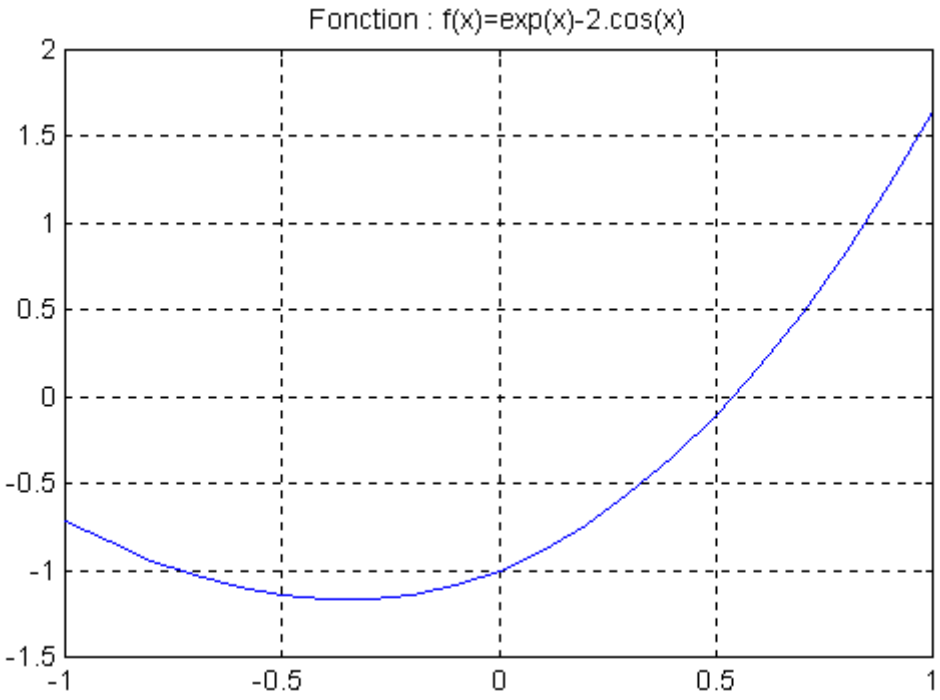


fig. 6 : Localisation de la valeur x_0 où $f(x_0) \approx 0$

Pour chercher la solution de $f(x)$, on peut rajouter au programme précédent 'p*num1.m*' quelques lignes :

```
%************************
% Etude de la fonction : *
% f(x)=exp(x)-2.cos(x) *
%************************
clf;
x=-1:0.1:1;
f=exp(x)-2*cos(x);
figure(1);
plot(x,f); grid on;
title('Fonction : f(x)=exp(x)-2.cos(x)');

clear all;
clc;
x(1)=input('Donner la valeur initiale x(1): \n');
e=1e-10;
n=5000;
for i=2:n
    f=exp(x(i-1))-2*cos(x(i-1));
    diff=exp(x(i-1))+2*sin(x(i-1));
    x(i)=x(i-1)-f/diff;
    if abs(x(i)-x(i-1))<=e
        xp=x(i);
        fprintf('xp=%f\n',x(i));
        break;
    end
end

j=1:i;
figure(2);
plot(j,x(j),'*r',j,x(j));
xlabel('Nombre d''itérations');
title('Convergence de la solution : Méth. de Newt.-Raph.');
disp('Les valeurs successives de x(i) sont :');
x'
```

Ainsi, après exécution de ce programme, on obtient alors toutes les valeurs successives de $x(i)$ pour chaque itération. Les résultats sont données ci-dessous. On remarque qu'il y a convergence après 5 itérations :

```
>>pnum1
Donner la valeur initiale x(1):
0.5
xp=0.539785
Les valeurs successives de x(i) sont :
ans =
0.5000
0.5408
0.5398
0.5398
0.5398
```

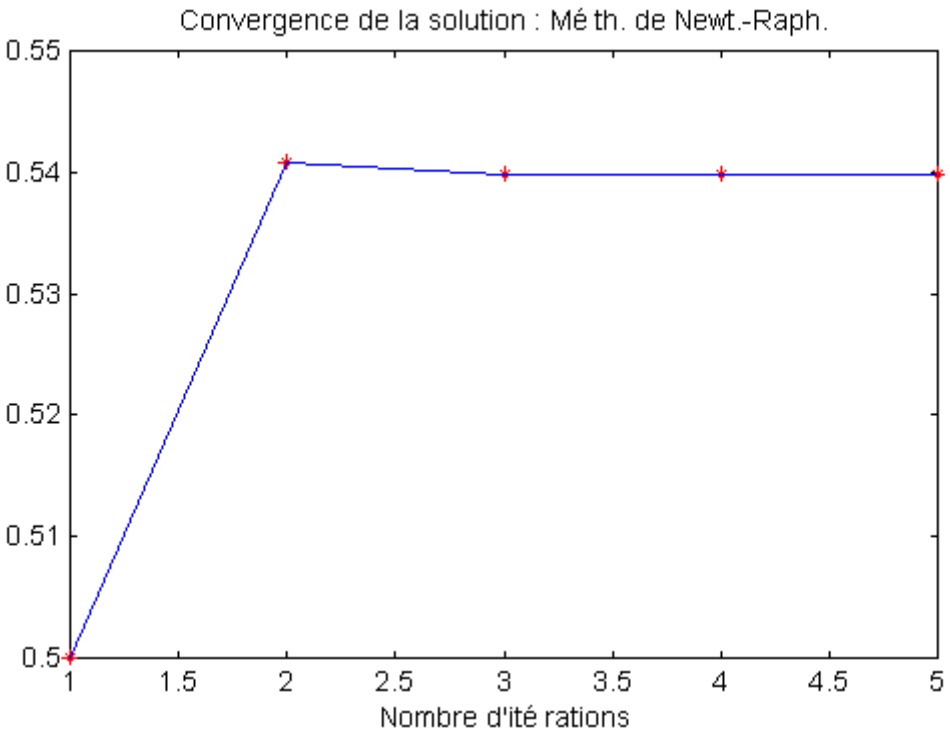


Fig. 7 : Évolution de la solution *x(i)* en fonction du nombre d'itérations

Dans MATLAB, on peut obtenir la même solution avec la fonction '*fzero*'. Pour cela, il faut créer le *fichier.m* MATLAB ('*f.m*', par exemple) dans lequel sera programmé $f(x)$.

```
%**********************************
% Fichier MATLAB représentant f(x) *
%**********************************
function f=f(x)
f=exp(x)-2*cos(x)
```

Pour obtenir la solution de $f(x) = 0$ au voisinage de $x = 0,5$, on exécute la commande '*fzéro('f',0.5)*'. Cette fonction affiche les valeurs obtenues de $f(x)$ à chaque itération pour donner à la fin la solution recherchée x^*.

```
>>d=fzero('f',0.5)
f =
-0.1064
-0.1430
-0.0692
-0.1579
-0.0536
-0.1788
-0.0313
-0.2080
```

5.8950e-004
-3.0774e-005
-4.1340e-009
2.2204e-016
-8.8818e-016

d =
0.5398

5.2. Résolution de systèmes d'équations non Linéaires

Nous allons appliquer cette méthode à un cas très simple : résolution d'un système de deux équations non linéaires à 2 inconnues, par exemple :

$$\begin{cases} f(x,y) = 0 \\ g(x,y) = 0 \end{cases}$$

l'algorithme de *Newton-Raphson* devient :

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} x_{k-1} \\ y_{k-1} \end{pmatrix} - \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \\ \frac{\partial g(x,y)}{\partial x} & \frac{\partial g(x,y)}{\partial y} \end{pmatrix}^{-1}_{(x_{k-1}, y_{k-1})} \cdot \begin{pmatrix} f(x_{k-1}, y_{k-1}) \\ g(x_{k-1}, y_{k-1}) \end{pmatrix}$$

Exemple :

$$\begin{cases} f(x,y) = x^2 - x + y^2 = 0 \\ g(x,y) = x^2 - y^2 - y = 0 \end{cases}$$

Les fonctions f et g admettent comme dérivées partielles :

$$\begin{pmatrix} \frac{\partial f(x,y)}{\partial x} = 2.x - 1 & \frac{\partial f(x,y)}{\partial y} = 2.y \\ \frac{\partial g(x,y)}{\partial x} = 2.x & \frac{\partial g(x,y)}{\partial y} = -2.y - 1 \end{pmatrix}$$

Afin de déterminer si le système admet des solutions sur $[-1,1]x[-1,1]$, on tracera les surfaces représentatives des fonctions f et g dans le but de montrer l'existence éventuelle d'un couple (x, y) pour lequel elles s'annulent. L'algorithme 'p*num2.m*' ci-dessous permet de tracer en 3 dimensions ces deux fonctions sur le même graphe.

```
%*****************************
% Fonctions à deux variables *
%*****************************

x=-1:0.1:1;
y=x;
[x,y]=meshgrid(x,y);
f=x.^2+y.^2-x;
g=x.^2-y.^2-y;
figure(1);
mesh(f);
grid on;
hold on;
mesh(g);
title('Courbes f(x,y) et g(x,y)');
xlabel('x'); ylabel('y');zlabel('f(x,y) et g(x,y)');
hold off;
```

On peut remarquer sur le graphe obtenu (en 3D) que les 2 fonctions f et g s'annulent simultanément en différents points. Grâce aux fonctionnalités de MATLAB, on peut rechercher les solutions éventuelles ou les conditions initiales pour l'algorithme de *Newton-Raphson* en faisant varier simultanément les variables x et y à l'aide de la fonction '*meshgrid*'. On obtient alors les courbes de niveaux des fonctions $f(x,y)$ et $g(x,y)$.

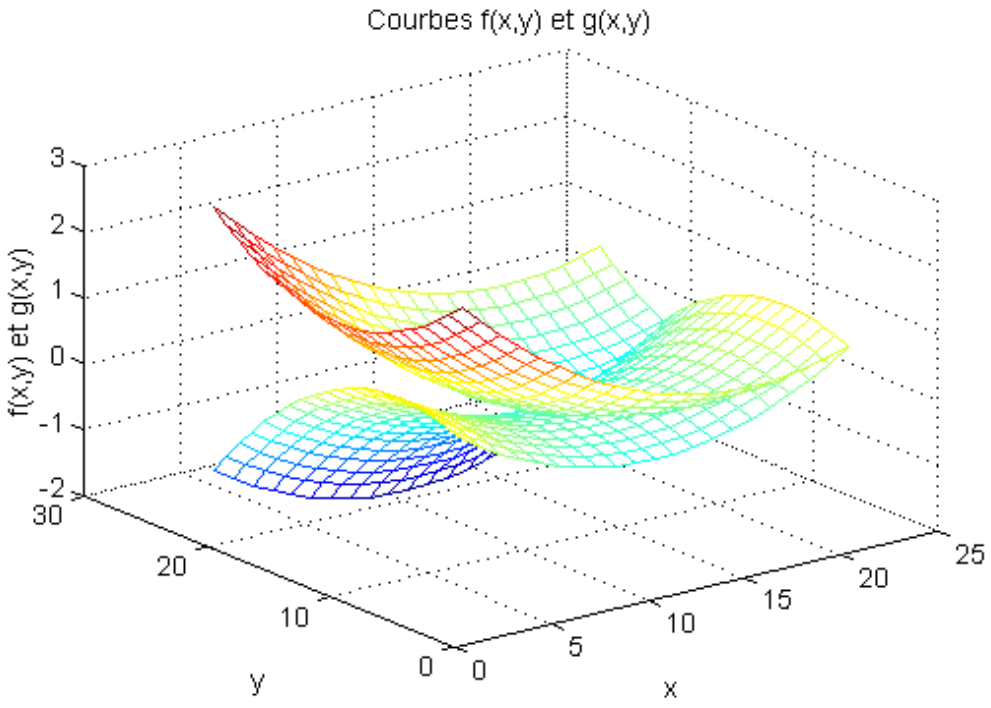


Fig. 8 : Fonctions $f(x,y)$ et $g(x,y)$ en 3D

On peut développer davantage l'algorithme 'p*num2.m*' pour chercher, par exemple, les conditions initiales et les solutions évidentes.

```
%*****************************
% Fonctions à deux variables *
%*****************************
x=-1:0.1:1;
y=x;
[x,y]=meshgrid(x,y);
f=x.^2+y.^2-x;
g=x.^2-y.^2-y;
figure(1);
mesh(f);
grid on;
hold on;
mesh(g);
title('Courbes f(x,y) et g(x,y)');
xlabel('x'); ylabel('y');zlabel('f(x,y) et g(x,y)');
hold off;
figure(2);
plot(f);
hold on;plot(g);grid on;
title('Intersection de f et g');
xlabel('x');ylabel('y');
axis([0 20 -0.5 0.5]);
gtext('f(x,y)');
gtext('g(x,y)');
```

Après exécution de '*pnum.m*', on obtient les courbes de $f(x,y)$ et $g(x,y)$ (fig. 9) en deux dimensions :

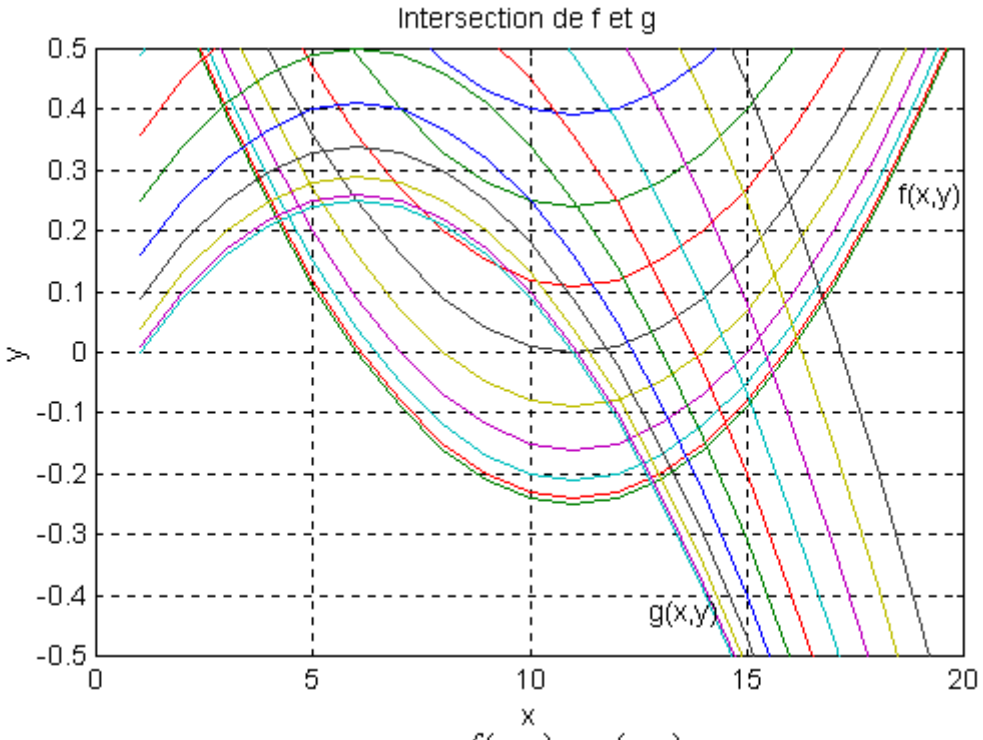


Fig. 9 : Fonctions $f(x,y)$ et $g(x,y)$ en 2D

Les solutions évidentes s'obtiennent par :

>> [i,j]=find(f==0 & g==0)
i =
11
j =
11

>>[x(11),y(11)]
ans =
-1 0

De cette façon, on peut retrouver plusieurs solutions évidentes si le pas d'échantillonnage est assez faible. Avec le pas de *0,1* choisi, on ne retrouve pas la solution évidente *(0,0)*.

Graphiquement, en faisant un zoom autour de l'abscisse '*x=11*' avec la fonction '*axis*', on peut cerner cette solution évidente.

>>axis([10.5 11.5 -0.02 0.02]);

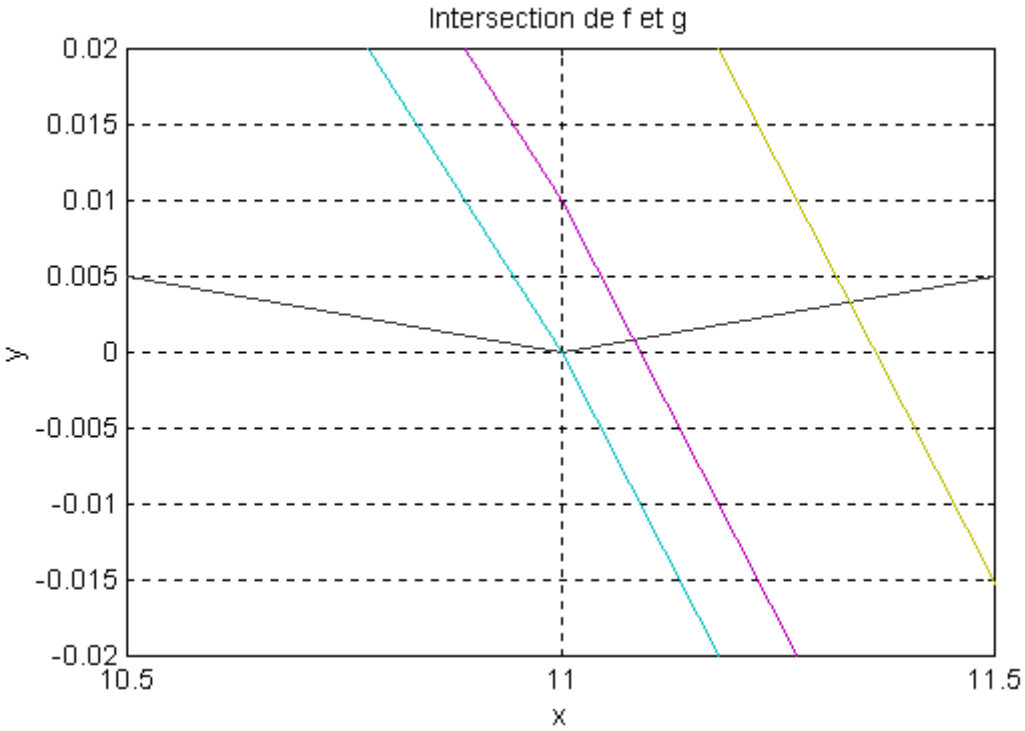


Fig. 10 : *zoom autour de l'abscisse* $x = 11$

Ainsi, l'algorithme de *Newton-Raphson* appliqué au système précédent, est représenté par la suite du programme MATLAB (*pnum2.m*) :

```
%*****************************
% Fonctions à deux variables *
%*****************************

clf;
clear all;
clc;
x=-1:0.1:1;
y=x;
[x,y]=meshgrid(x,y);
f=x.^2+y.^2-x;
g=x.^2-y.^2-y;
figure(1);
mesh(f);
grid on;
hold on;
mesh(g);
title('Courbes f(x,y) et g(x,y)');
xlabel('x'); ylabel('y');zlabel('f(x,y) et g(x,y)');
hold off;
figure(2);
plot(f);
hold on;plot(g);grid on;
```

```
title('Intersection de f et g');
xlabel('x');ylabel('y');
axis([0 20 -0.5 0.5]);
gtext('f(x,y)');
gtext('g(x,y)');
hold off;
clear x y;
% Conditions initiales
x(1)=0.85; y(1)=0.35;

% Algorithme de Newton-Raphson
for i=2:10
    diff=inv([2*x(i-1)-1 2*y(i-1);2*x(i-1) -2*y(i-1)-1]);
    f=x(i-1)^2-x(i-1)+y(i-1)^2;
    g=x(i-1)^2-y(i-1)^2-y(i-1);
    xy=[x(i-1) y(i-1)]'-diff*[f,g]';
    x(i)=xy(1);
    y(i)=xy(2);
end
```

Le choix des conditions initiales influe sur la convergence de l'algorithme. Dans ce qui suit (suite de '*pnum2.m*'), nous étudierons l'évolution des solutions par 2 jeux de conditions initiales.

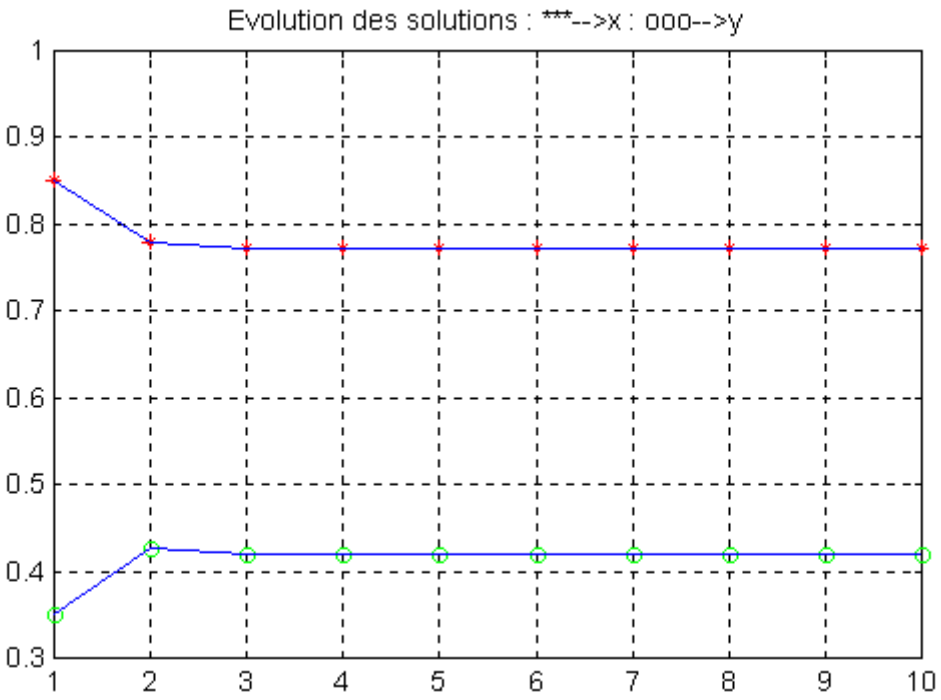
```
%*****************************
% Fonctions à deux variables *
%*****************************
clf;
clear all;
clc;
x=-1:0.1:1;
y=x;
[x,y]=meshgrid(x,y);
f=x.^2+y.^2-x;
g=x.^2-y.^2-y;
figure(1);
mesh(f);
grid on;
hold on;
mesh(g);
title('Courbes f(x,y) et g(x,y)');
xlabel('x'); ylabel('y');zlabel('f(x,y) et g(x,y)');
hold off;
figure(2);
plot(f);
hold on;plot(g);grid on;
title('Intersection de f et g');
xlabel('x');ylabel('y');
axis([0 20 -0.5 0.5]);
gtext('f(x,y)');
gtext('g(x,y)');
hold off;
clear x y;
% Conditions initiales
x(1)=0.85; y(1)=0.35;
% x(1)=0.2; y(1)=0.1;
% Algorithme de Newton-Raphson
```

```
for i=2:10
    diff=inv([2*x(i-1)-1 2*y(i-1);2*x(i-1) -2*y(i-1)-1]);
    f=x(i-1)^2-x(i-1)+y(i-1)^2;
    g=x(i-1)^2-y(i-1)^2-y(i-1);
    xy=[x(i-1) y(i-1)]'-diff*[f,g]';
    x(i)=xy(1);
    y(i)=xy(2);
end

% Tracé des courbes d'évolution des solutions
figure(3);
plot(1:10,x,'*r',1:10,x);
hold on;
plot(1:10,y,'og',1:10,y);
title('Evolution des solutions : ***-->x et ooo-->y');
grid on;
hold off;
```

L'évolution des solutions *x* et *y* (pour les conditions initiales : $x_1 = 0,85$ et $y_1 = 0,35$) est donnée par la figure 11 ci-après. Ces solutions sont les suivantes :

```
>>x
x =
0.8500 0.7800 0.7719 0.7718 0.7718 0.7718 0.7718 0.7718 0.7718 0.7718

>>y
y =
0.3500 0.4271 0.4197 0.4196 0.4196 0.4196 0.4196 0.4196 0.4196 0.4196
```

Les 2 solutions peuvent être obtenues au bout de 2 itérations grâce au bon choix des conditions initiales. Avec les conditions initiales x(1)=0,2 et y(1)=0.1, il y a convergence vers la solution (0,0) au bout de 3 itérations.



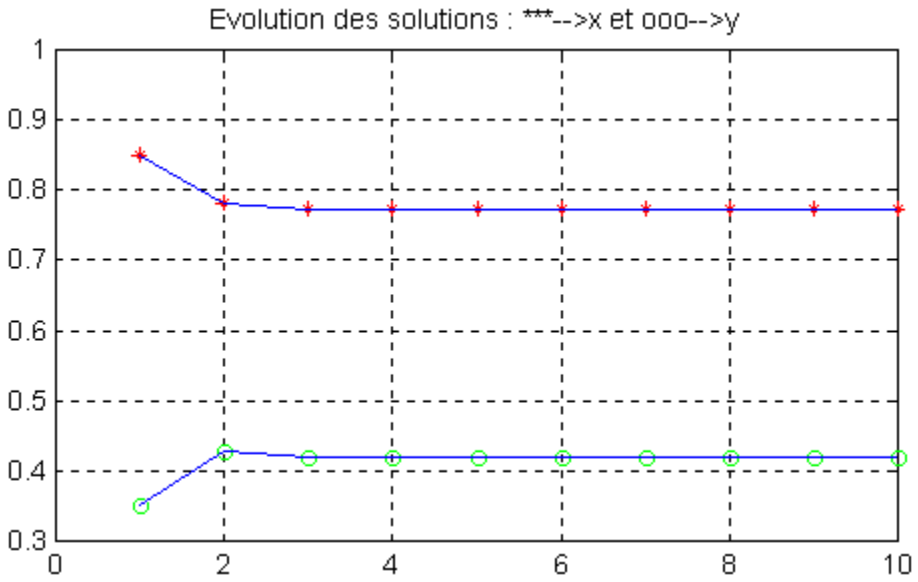


Fig. 11 : Convergence des solutions *(x,y)* : x(1)=0,85 et y(1)=0,35

Pour les conditions initiales *x(1)=0,2* et *y(1)=0,1*, l'évolution des solutions *x* et *y* est :

>>x
x =
0.2000 -0.1031 -0.0112 -0.0002 -0.0000 -0.0000 -0.0000 0 0 0

>>y
y =
0.1000 -0.0594 -0.0054 -0.0001 -0.0000 -0.0000 -0.0000 0 0 0

La boite à outil '*Optimisation Toolbox*' propose les fonctions '***fsolve***' et '***fsolve2***' qui permettent respectivement, la recherche des racines d'une fonction dans un intervalle donné et les solutions d'équations non linéaires et de systèmes d'équations non linéaires.

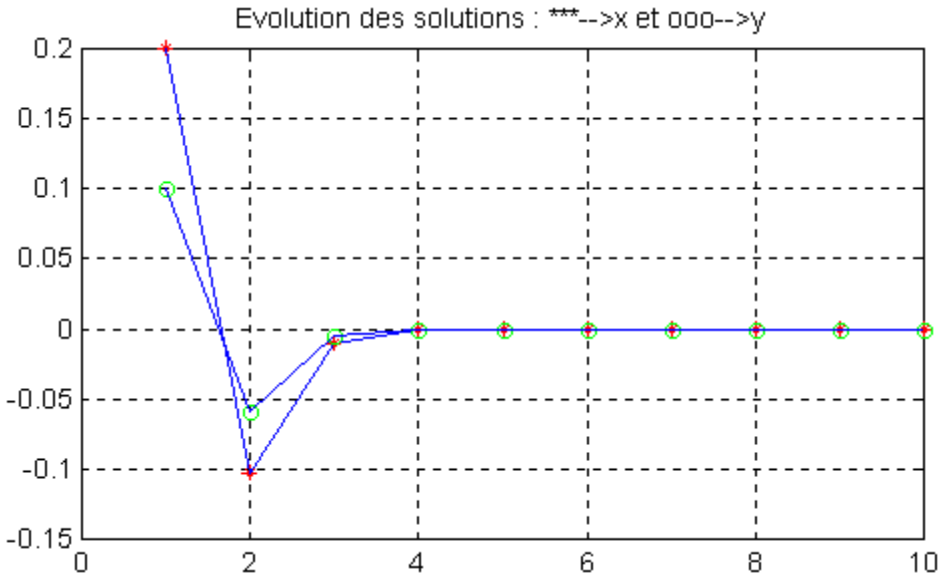


Fig. 12 : Convergence des solutions *(x,y)* : *x(1)=0,2* et *y(1)=0,1*

Intégration numérique des fonctions

1. Introduction

Dans la plupart des cas, les fonctions analytiques, du fait de leurs complexités, ne sont pas intégrables analytiquement. Dans d'autres cas, on a des fonctions qui sont évaluées numériquement en différents points de l'intervalle où ces dernières sont données, et l'intégrale de ces types de fonctions ne peut être obtenue que par des approches numériques. Dans ce chapitre, on s'intéresse aux méthodes utilisées fréquemment ; à savoir la méthode des trapèzes, la méthode de Simpson, les formules de Newton-Cotes et la méthode de Gauss. Nous étudierons également les méthodes de calcul d'intégrales doubles et les intégrales des fonctions non définies sur leurs bornes.

2. Méthodes d'intégrations numériques

2.1. Méthode des trapèzes

Soit $f(x)$ la fonction à intégrer sur $[a,b]$. L'intégrale I de $f(x)$ s'écrit en utilisant la méthode des trapèzes :

$$I = \int_a^b f(x).dx = \frac{h}{2}.\left(f_1 + 2.f_2 + 2.f_3 + \ldots + 2.f_i + \ldots + 2.f_n + 2.f_{n+1}\right) + E$$

$$= \frac{h}{2}.\left(f(x_1) + f(x_{n+1}) + 2.\sum_{i=2}^{n} f(x_i)\right) + E$$

où $h = \frac{b-a}{n}$; $x_i = a + (i-1).h$; $f_i = f(x_i)$ et $i = 1, 2, 3, \ldots, n, n+1$.

Le terme représentant l'erreur est :

$$E \approx -\frac{(b-a)}{12}.h^2.\overline{f''} \approx -\frac{(b-a)}{12.n^2}.\overline{f''}$$

$\overline{f''}$ est la moyenne de $f''(x)$ sur l'intervalle $[a,b]$. L'erreur E est inversement proportionnelle à la valeur de n^2.

Si la fonction f est donnée sur les intervalles réguliers ($x_i - x_{i-1} = h$), la méthode des trapèzes peut être écrite dans *Matlab* sous la forme :

$$I = h*\left(sum(f) - 0.5*\left(f(1) + f\left(length(f)\right)\right)\right)$$

avec :

$$sum(f) = f_1 + f_2 + f_3 + ... + f_i + ... + f_n + f_{n+1}$$

On peut également faire un programme pour calculer l'intégrale I.
Celui-ci appelé *'trapez_v.m'* par exemple, est listé ci-dessous :

```
function I=trapez_v(g,h)
I=(sum(f)-(f(1)+f(length(f)))/2)*h;
```

Considérons par exemple la fonction à intégrer : $f(x) = x^2 + 2x - 1$ sur un intervalle $[-10,8]$ où le pas est "*h*" égal à *1*. En mode interactif dans MATLAB (mode commande), avant de lancer le programme *'trapez_v.m'*, on donne la fonction f ainsi que ses bornes :

```
>>x=-10:1:8;
>>f=x.^2+2*x-1;
>>h=1;
```

On exécute ensuite le programme *'trapez_v.m'*, et on obtient :

```
>>I=trapez_v(f,h)
I =
2265
```

En utilisant les programmes *'trapez_n.m'* et *'trapez_g.m'* listés ci-après, on peut calculer aussi l'intégrale *I*.

```
>>I=trapez_n('fonction_f',a,b,n)
```
ou
```
>>I=trapez_g('fonction_f',a,b,n)
```

Liste du programme *'trapez_n.m'* :

```
function I=trapez_n(fonction_f,a,b,n)
h=(b-a)/n;
x=a+(0:n)*h;
f=feval(fonction_f,x);
I=trapez_v(f,h)
```

Liste du programme *'trapez_g.m'* :

```
function I=trapez_g(fonction_f,a,b,n);
n=n;
hold off;
h=(b-a)/n;
x=a+(0:n)*h;
```

```
f=feval('fonction_f',x);
I=h/2*(f(1)+f(n+1));
if n>1
    I=I+sum(f(2:n))*h;
end

h2=(b-a)/100;
xc=a+(0:100)*h2;
fc=feval('fonction_f',xc);
plot(xc,fc,'r');
hold on;
title('Méthode des trapèzes');
xlabel('x');
ylabel('y');
grid on;
plot(x,f,'m');
plot(x,zeros(size(x)),'c')

for i=1:n;
    plot([x(i),x(i)],[0,f(i)],'g');
end
```

La fonction *'fonction_f'* est donnée par le programme *'fonction_f.m'*.

Remarque :

La fonction *y=feval('sin',x)* est équivalente à *y=sin(x)*.

Exemples :

1°) On donne la fonction 'fonction_f' par le sous-programme *'fonction_f.m'* listé ci-dessous :

```
%****************
% fonction f(x) *
%****************
function f=f(x);
f=x+2*log(0.0000025235*x);
```

En exécutant le programme *trapez_g('fonction_f',1,50,60)*, où les bornes a et b sont égales respectivement à 1 et 50 et le nombre d'intervalles n est égal 60, on obtient l'intégrale de $f(x)$ sur $[1,50]$ pour 60 pas :

```
>>trapez_g('fonction_f',1,50,60)
ans =
279.3889
```

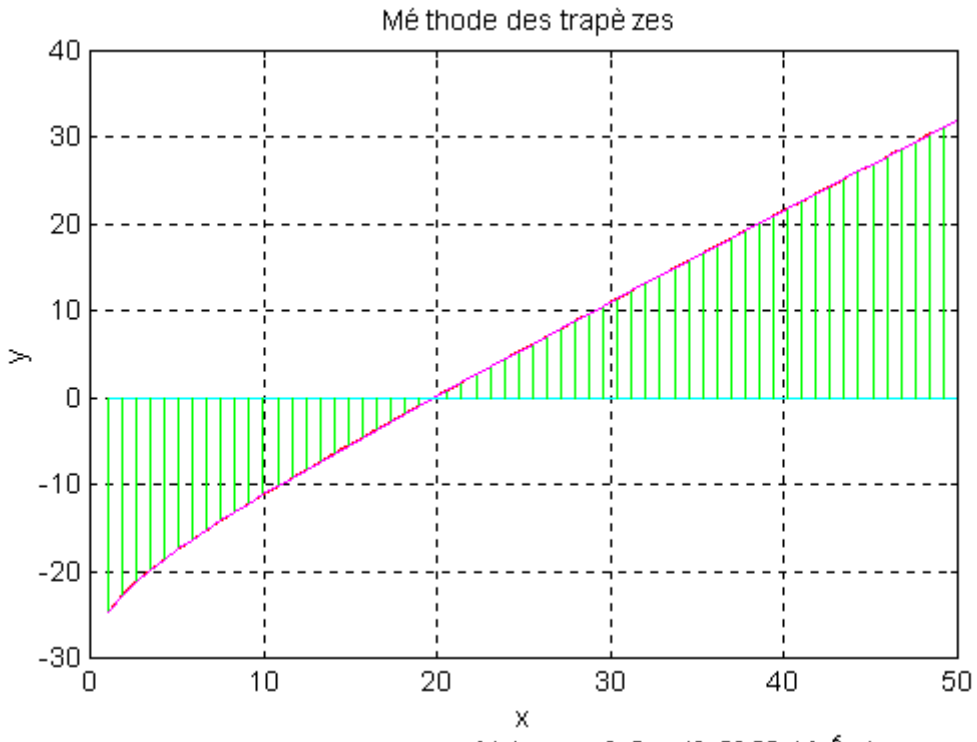


Fig. 1 : Intégrale de la fonction $f(x) = x + 2.Log(2.5253.10^{-6}.x)$

2°) Un véhicule de masse *m=2000 kg* se déplace à la vitesse de *V=30 m/s*. Soudainement, le moteur est débrayé à $t = 0$. L'équation du mouvement après l'instant $t = 0$ est donnée par :

$$m.V.\frac{dV}{dx} = -8,1.V^2 - 1200$$

(a)

où x est la distance linéaire mesurée à partir de $t = 0$.

Dans cette équation (a), le terme de gauche représente la force d'accélération, le 1er terme de droite représente la résistance aérodynamique exercée par le vent sur le véhicule et le second terme est le coefficient de frottement. Calculer la distance au delà de laquelle la vitesse de la voiture se réduit à *15 m/s*.

Solution :

L'équation (a) peut être écrite sous la forme :

$$dx = -\frac{mV.dV}{8,1.V^2 + 1200}$$

L'intégration de cette équation donne :

$$x = -\int_{15}^{30} \frac{m.V.dV}{8,1.V^2 + 1200}$$

Cette intégrale peut être évaluée par la méthode des trapèzes.

Si on se donne 60 intervalles (ou 61 points), on peut écrire :

$V_i = 15 + (i-1)\Delta V$ où $i = 1,2,3,\ldots,61$ et $\Delta V = \frac{45-15}{60} = 0,5$.

En définissant :

$$x_i = -\frac{m.V_i}{8,1.V_i^2 + 1200}$$

et en appliquant la méthode des trapèzes pour l'intégration, on obtient :

$$x \approx \Delta V.\left(\sum_{i=1}^{61} f_i - 0,5.(f_1 + f_{61})\right)$$

(b)

Le programme suivant (*trapeze.m*)permet de calculer la valeur x donnée par la formulation précédente (b) :

```
>>trapeze
x =
127.5066
```

En comparant cette solution à la solution exacte (*=127,51 m*), on remarque que l'erreur relative induite par cette méthode est de l'ordre de $2,7.10^{-3}$ %.

La liste du programme *trapeze.m* est la suivante :

```
clear;
clc;
nb_points=61;
i=1:nb_points;
h=(30-15)/(nb_points-1);
V=15+(i-1)*h;
f=2000*V./(8.1*V.^2+1200);
x=trapez_v(f,h)
```

3°) Connaissant l'intégrale exacte de la fonction $f(x)=\sqrt{1+e^x}$ sur l'intervalle $[0,2]$, on a $I=4,006994$. Étudier l'effet du nombre d'intervalles n sur l'erreur de calcul en adoptant la méthode trapézoïdale :

$$I=\int_0^2 \sqrt{1+e^x}\,dx$$

Solution :

Le programme suivant *'trz.m'* permet de calculer l'intégrale I :

```
clc;
clear;
Iexact=4.006994;
a=0;b=2;
fprintf('\n Méthode Trapézoïdale étendue \n');
fprintf('\n n\t \t I\t Pourc. erreur relat.\n');
fprintf('-----------------------------------------------\n');
n=1;
for k=1:10
    n=2*n;
    h=(b-a)/n;
    i=1:n+1;
    x=a+(i-1)*h;
    f=sqrt(1+exp(x));
    I=trapez_v(f,h);
    erreur=abs(Iexact-I)/Iexact;
    fprintf('%d\t %10.5f\t %10.8f\n',n,I,erreur);
end
```

Les résultats obtenus sont les suivants :

Méthode Trapézoïdale étendue

n I Pourc. erreur relat.

---

2 4.08358 0.01911429
4 4.02619 0.00478998
8 4.01180 0.00119825
16 4.00819 0.00029965
32 4.00729 0.00007496
64 4.00707 0.00001878
128 4.00701 0.00000474
256 4.00700 0.00000123
512 4.00700 0.00000035
1024 4.00699 0.00000013

D'après ces résultats, on montre que quand n est doublé, l'erreur relative décroît par un facteur 4.

2.2. Méthode de Simpson

Soit I l'intégrale de $f(x)$ sur l'intervalle $[a,b]$. Par la méthode de *Simpson*, I s'écrit :

$$I = \int_a^b f(x).dx = \frac{\Delta x}{3}.\left(f(1) + f(n+1) + 4.\sum_{\substack{i=2 \\ i \to pair}}^{n-1} f(i) + 2.\sum_{\substack{i=3 \\ i \to impair}}^{n} f(i) \right) + E$$

où $E = \Theta(\Delta x)^4$; $f(i) = f\left(a + \frac{i-1}{h}\right)$ et $h = \frac{b-a}{n}$.

Le terme E est donné par :

$$E \approx -(b-a)\frac{h^4}{180}.\overline{f''''}$$

et $\overline{f''''}$ est la moyenne de $f''''(x)$ sur l'intervalle ouvert $]a,b[$.

Exemple :

Évaluer l'intégrale de $f(x) = \sqrt{1+e^x}$ sur l'intervalle $[0,2]$ avec la méthode de Simpson pour $n = 2, n = 4, n = 8,$ et $n = 16$.

Solution :

La liste du programme *'msimp.m'* est la suivante :

```
clc;
clear;
Iexact=4.006994;
a=0;b=2;
fprintf('\n Méthode de Simpson \n');
fprintf('\n n\t \t I\t Pourc. erreur relat.\n');
fprintf('-------------------------------------------\n');
n=1;
for k=1:4
    n=2*n;
    h=(b-a)/n;
    i=1:n+1;
    x=a+(i-1)*h;
    f=sqrt(1+exp(x));
    I=h/3*(f(1)+4*sum(f(2:2:n))+f(n+1));
    if n>2
        I=I+h/3*2*sum(f(3:2:n));
    end
```

```
    erreur=abs(Iexact-I)/Iexact;
    fprintf('%d\t %10.5f\t %10.8f\n',n,I,erreur);
end
```

Les résultats de cet algorithme sont donnés ci-dessous :

Méthode de Simpson

n *I* *Pourc. erreur relat.*

---

2 *4.00791* *0.00022935*
4 *4.00705* *0.00001521*
8 *4.00700* *0.00000101*
16 *4.00699* *0.00000012*

On montre qu'à partir de $n = 8$, l'erreur relative devient presque nulle (de l'ordre de $10^{-4}\%$).

Les programmes *'simp_v'* et *'simp_n'* peuvent être utilisés pour intégrer la fonction $f(x)$. Dans l'algorithme *'simp_v(f,h)'*, f est un vecteur contenant les ordonnées de l'intégrante et h représente le pas de l'intervalle d'intégration.

D'où :

>>I=simp_n('fonction_f',a,b,n)

permet d'évaluer l'intégrale I dans laquelle '*fonction_f*' est le nom du sous-programme contenant la fonction $f(x)$, a et b sont les limites de l'intervalle d'intégration et $n = \frac{b-a}{h}$.

Exemple :

Soit $f(x) = \frac{R^2 - x^2}{b^2 . \sqrt{g.(x+R)}}$ où $n = 8, b = 0{,}1$ et $g = 9{,}81$.

➔ $I = \int_{-0.9.R}^{R} f(x).dx = \int_{-0.9.R}^{R} \frac{(R^2 - x^2)dx}{b^2 . \sqrt{g.(x+R)}}$

Ecrire un algorithme utilisant le sous-programme *simp_v(f,h)* (à écrire) et qui permet d'évaluer I sur 100 intervalles.

Solution :

Appelons ce programme *simp.m* par exemple, et donnons sa liste :

```
clear;
R=5;g=9.81;b=0.1;
x1=-0.90*R;x2=R;
h=(x2-x1)/100;
x=x1:h:x2;
f=(R^2-x.^2)./(b^2*sqrt(2*g*(x+R)));
I=simp_v(f,h)
```

Le sous-programme *simp_v(f,h)* est :

```
function I=simp_v(f,h);
n=length(f)-1; % Nombre d'intervalles

if n==1
    fprintf('Données à un seul intervalle'\n');
    return;
end

if n==2
    I=h/3*(f(1)+4*f(2)+f(3));
    return; % Revenir et continuer le calcul
end

if n==3
    I=(3/8)*h*(f(1)+3*f(2)+3*f(3)+f(4));
    return;
end

I=0;
if 2*floor(n/2)~=n ; %Le nombre d'intervalle doit être pair
    I=3*h/8*(f(n-2)+3*f(n-1)+3*f(n)+f(n+1));
    m=n-3;
else
    m=n;
end

I=I+h/3*(f(1)+4*sum(f(2:2:m))+f(m+1));

if m>2
    I=I+(h/3)*2*sum(f(3:2:m))
end
```

En exécutant le programme *simp.m*, on trouve l'intégrale I par la méthode de Simpson.:

>>simp
I =
1.8522e+003

3. Fonctions MATLAB utilisées pour l'intégration numérique

Dans Matlab (Toolbox), il existe 2 fonctions appelées '*quad'* et *'quad8'* pour l'intégration numérique. La fonction *'quad'* utilise la méthode de Simpson et la fonction *'quad8'* utilise les formules de *Newton-Cotes* à l'ordre 8. Ces deux fonctions sont utilisées de la façon suivante :

- *quad('fonction_f',a,b)*
- *quad('fonction_f',a,b,tol)*
- *quad('fonction_f',a,b,tol,trace)*

Dans la première forme, la tolérance *'tol'* qui correspond à l'erreur relative E, est considérée égale à *0,001* par défaut. Ainsi, le calcul quadratique de l'intégrale est réitéré jusqu'à ce que la tolérance soit satisfaite. Si la 3ème forme est utilisée avec une valeur non nulle de *'trace'*, un graphique d'évolution des itérations sera affiché sur l'écran. Quand à la fonction *'quad8'*, elle est utilisée de la même manière que la fonction *'quad'* dans les 3 formes d'expressions précédentes.

Exemple :

Soit à calculer l'intégrale suivante :

$$I = \int_0^{\infty} (x-1).e^{-x.(x-2)}.dx$$

avec la transformation suivante :

$$y = x.(x-2) = (x-1)^2 - 1$$

Dans ce cas, on a :

$$dx = \frac{dy}{2.\sqrt{y+1}}$$

On obtient donc :

$$I = \frac{1}{2}.\int_0^{\infty} e^{-y}.dy$$

Or, on connaît la fonction $\Gamma(x)$ (fonction Gamma) eulérienne du deuxième espèce qui est définie par :

$\Gamma(x) = \int_0^{\infty} t^{(x-1)} . e^{-t} . dt$ où $f(x)$ est un réel.

Pour un entier n strictement positif, $\Gamma(n)$ a la propriété suivante :

$\Gamma(n) = (n-1)!$

On pourra donc s'en servir pour calculer la factorielle d'un entier naturel n :

➔. $n! = \Gamma(n+1)$

Dans Matlab, cette fonction est notée *'gamma'*.

Exemple :

La factorielle de 10 est donnée par *gamma(10+1)=gamma(11)* :

```
>> fact=gamma(11)
fact =
3628800
```

Cette propriété est à l'origine du nom de la *'fonction factorielle'*, attribuée souvent à la fonction *'Gamma'*. La fonction *'gamma'* est donc prédéfinie dans Matlab. Cette fonction permet, de part sa définition pour tous les arguments réels positifs, d'introduire *'la factorielle'* des arguments non entiers.

Exemple :

$\Gamma(0,5) = 0,5! = \sqrt{\pi}$

```
>> format long

>>gamma(0.5)
ans =
1.77245385090552

>>racine_de_pi=sqrt(pi)
racine_de_pi =
1.77245385090552
```

>>gamma(0)
Warning: Divide by zero.
ans =
Inf

>>gamma(-2)
Warning: Divide by zero.
ans =
-Inf

<u>Tracé de la fonction Gamma</u>

Tracer la fonction Gamma pour $x \in [-3,3]$.

>>x=-3:0.1:3;

>>gx=gamma(x);

>>plot(x,gx);grid;title('Allure de la fonction Gamma');

>>xlabel('x');ylabel('Gamma(x)');

>>hold on;plot(x,zeros(size(x)))

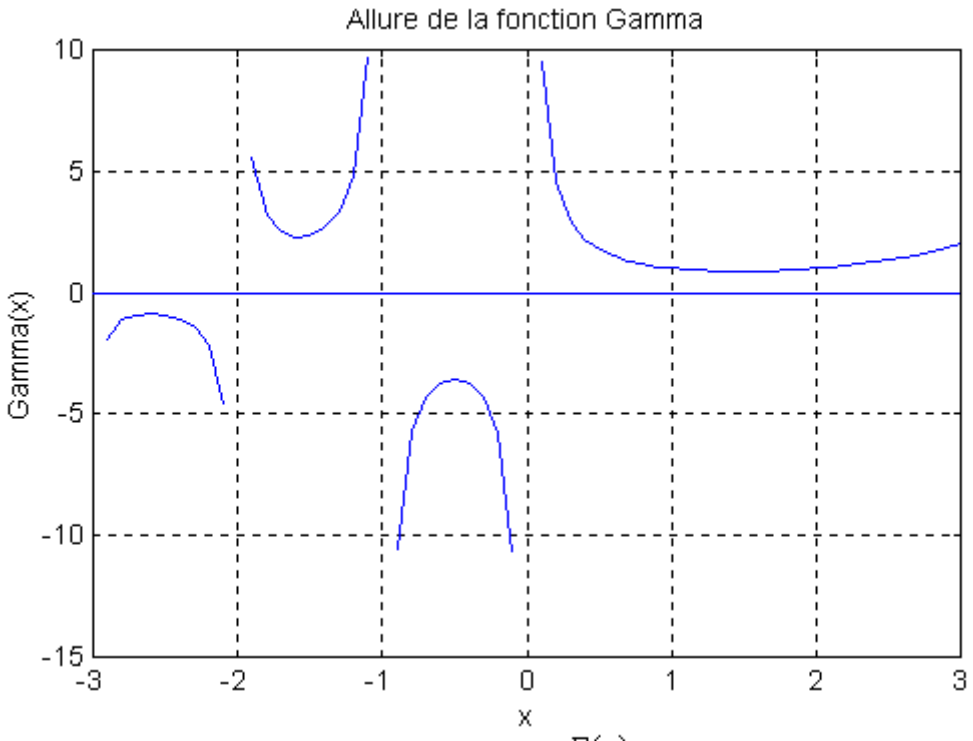


Fig. 2 : Fonction $\Gamma(x)$

Revenons maintenant à l'intégration de la fonction $f(x)$:

$$I = \int_0^{\infty} (x-1).e^{-x.(x-2)}.dx$$

D'après le changement de variable effectué, cette intégrale s'écrit :

$$I = \frac{1}{2}.\int_0^{\infty} e^{-y}.dy = \frac{1}{2}.\Gamma(1)$$

Calculons cette intégrale par les fonctions Matlab *'quad8'* et *'gamma'*. La fonction *'quad8'* nécessite l'écriture de la fonction à intégrer dans un fichier contenant l'extension *'.m'* (programme Matlab).

Liste du programme *fct.m* :

```
function f=fct(x);
% fonction à intégrer
f=(x-1).*exp(-x.*(x-2));
```

Tracé de la fonction à intégrer :

```
>>x=0:0.01:5;

>>ft=fct(x);

>>plot(x,ft);

>>grid; title('fonction à intégrer');

>>xlabel('x');ylabel('f(x)');
```

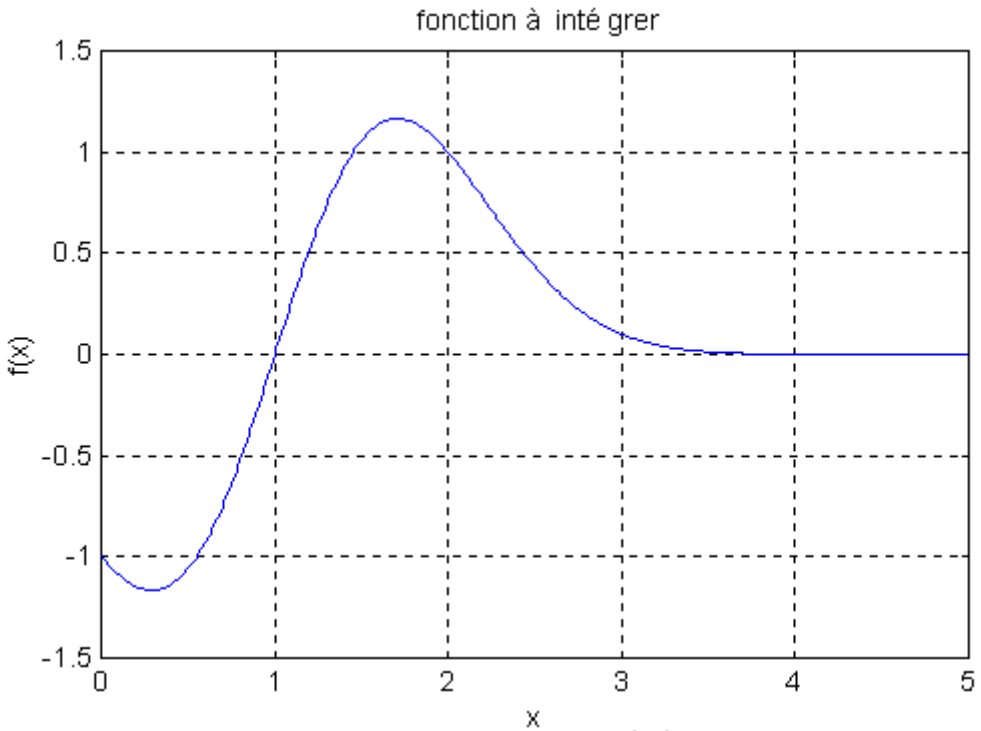


Fig. 3 : $f(x) = (x-1).e^{-x(x-1)}$

Dans le programme donné ci-dessous (*quadgamm.m*), on calcule l'intégrale précédente par les fonctions *'quad8'* et *'gamma'*, pour lesquelles on compare les performances en terme de temps de calcul et du nombre d'opérations en virgule flottante.

```
%**************************************
% Calcul de l'intégrale de f(x) par la *
% la fonction 'quad8' sur un K6-233 *
%**************************************
clear all; clc;
flops(0); % Mise à zéro du compteur d'opérations
tic; % Déclenchement du compteur temps de calcul
Iquad=quad8('fct',0,5)
Nbre_op1=flops, %Arrêt du compteur d'opérations
tps_q=toc, %Arrêt du compteur de temps de calcul

%**************************************
% Calcul de l'intégrale de f(x) par la *
% la fonction 'gamma' *
%**************************************
flops(0);
tic;
Igamma=gamma(1)/2
Nbre_op2=flops
tps_g=toc
```

En exécutant le programme *quadgamm.m*, on obtient les résultats suivants :

```
>> format long

>> quadgamm
Iquad =
0.49999987773178
Nbre_op1 =
686
tps_q =
0.06000000000000
Igamma =
0.50000000000000
Nbre_op2 =
40
tps_g =
0.05000000000000
```

D'après ces résultats, on remarque bien que le nombre d'opérations en virgule flottante et le temps d'exécution dans le cas de l'utilisation de la fonction *'gamma'* sont nettement inférieurs à ceux que l'on obtient en utilisant la fonction *'quad8'*.

La fonction *'gamma'* prend des valeurs très grandes au voisinage de zéro et des entiers négatifs. Afin d'éviter un dépassement de capacité, MATLAB dispose de la fonction *'gammaln'* qui retourne le logarithme népérien de ces valeurs.

Exemple :

```
>>format

>>x=5e-200;

>>gamma(x)
ans =
2.0000e+199

>>gammaln(x)
ans =
458.9076
```

Résolution numérique des équations différentielles et des équations aux dérivées partielles

1. Introduction

Le comportement dynamique des systèmes est un sujet très important en physique. Un système mécanique par exemple, se traduit par des déplacements, des vitesses et des accélérations. Un système électrique ou électronique, se traduit par des tensions, des intensités et des dérivées temporelles sur ces quantités. Un système thermique se traduit par des températures, des gradients de température, de coefficients d'échanges thermiques, etc. En général, les équations utilisées pour décrire de tels comportements dynamiques, incluent des quantités inconnues représentant les fonctions recherchées et leurs dérivées.

Une équation qui comporte une ou plusieurs dérivées de la fonction inconnue est appelée '*équation différentielle*', qui est représentée dans MATLAB par l'abréviation '***ODE***'. L'ordre de cette équation est déterminé par l'ordre du degré le plus élevé de la dérivation.

Les équations différentielles peuvent être classée en deux catégories : les équations différentielles avec des *conditions initiales* et les équations différentielles avec des *conditions aux limites*.

2. Équations différentielles du 1er ordre

Les équations du 1er ordre à conditions initiales peuvent être écrites sous la forme :

$$\begin{cases} y'(t) = f(y,t) \\ y(0) = y_0 \end{cases}$$

où $f(y,t)$ est une fonction de y et de t, et la seconde équation représente la condition initiale sans laquelle la solution de l'équation différentielle ne peut être évaluée.

Exemple 1 :

Soit à résoudre l'équation différentielle suivante :

$$M.\frac{dV}{dt} = -C.V^2 + M.g$$

où :

$$\begin{cases} M = 70kg \\ g = 9{,}81N/kg \\ C = 0{,}27kg/m \end{cases}$$

Déterminer numériquement $V(t)$. On choisit un pas de temps $h = 0{,}1s$, et on donne comme condition initiale $V(t=0) = 0$. Tracer la solution de cette équation différentielle pour $0 \le t \le 20s$.

Solution :

L'équation $M.\frac{dV}{dt} = -C.V^2 + M.g$ est équivalente à $V'(t) = f(y,t)$ où $V(0) = 0$ et $f(y,t) = -\frac{C}{M}.V^2 + g$.

La solution de cette équation peut être effectuée par la méthode d'*Euler*, qui consiste à écrire :

$$V_{n+1} \approx V_n + h.f(y,t)$$

Le programme MATLAB suivant (*eul1.m*) permet la résolution de cette équation :

```
%*****************************************
% Résolution d'une équation différentielle *
% par la méthode d'Euler à l'ordre 1       *
%*****************************************
clear ; clc ;
t=0 ;n=0 ;V=0 ;
C=0.27 ;M=70 ;g=9.81 ;h=0.1 ;
t_R(1)=t ;V_R(1)=V ;

while t<=20
    n=n+1 ;
    V=V+h*(-C/M*V.^2+g);
    t=t+h;
    t_R(n+1)=t;
    V_R(n+1)=V;
end

plot(t_R,V_R);
xlabel('Temps (en s)');
ylabel('Vitesse (en m/s)');
grid on;
```

On obtient ainsi un graphique (fig. 1) qui donne l'évolution de la solution $V(t)$ en fonction du temps.

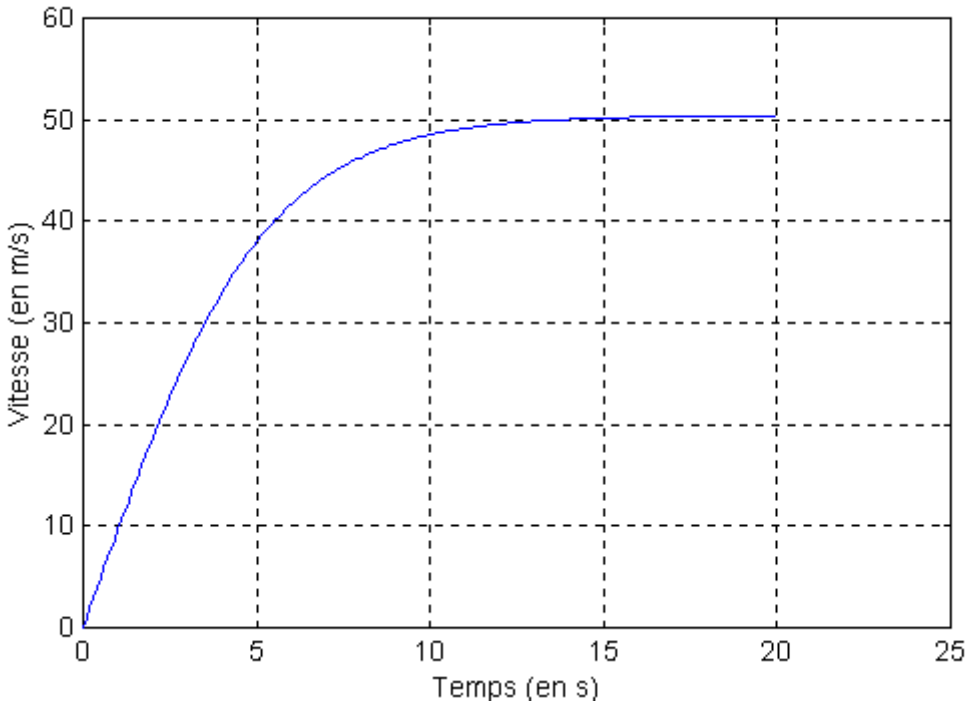


Fig. 1 : évolution de la vitesse en fonction du temps

Exemple 2 :

On veut résoudre l'équation différentielle du 1er ordre suivante :

$$\begin{cases} y' + y^{1,5} = 1 \\ y(0) = 10 \end{cases}$$

en utilisant les *formules d'Adams ouvertes* à l'ordre 1 et les *formules d'Adams* (prédicteur-correcteur) pour un pas $h = 0,1$. On donne la limite de convergence $\varepsilon = 10^{-5}$.

- Calculer y par les deux méthodes pour $0 \le t \le 1,5$
- Ecrire un programme dans Matlab permettant de calculer et de tracer les résultats de ces deux méthodes sur le même graphe,
- Conclusion.

Solution :
* Les formules *d'Adams* (prédicteur-correcteur) consiste à écrire la solution sous la forme : $y_{n+1} = y_n + \frac{h}{2}.\left(f(y_{n+1}, t_{n+1}) + f(y_n, t_n)\right)$

Dans ce cas, on a :

$$y' = f(y,t) = 1 - y^{1,5}$$

or y_{n+1} est inconnue ; donc $f(y_{n+1}, t_{n+1})$ est aussi inconnue.

Dans ce cas, on utilise la méthode itérative suivante :

$$y_{n+1}^{(k)} \approx y_n + \frac{h}{2}.\left(f(y_{n+1}^{(k-1)}, t_{n+1}) + f(y_n, t_n)\right)$$

où $y_{n+1}^{(k)}$ est la $k^{ième}$ valeur itérée de y_{n+1}, et $y_{n+1}^{(0)}$ est une valeur arbitraire initiale de y_{n+1}. La résolution doit se faire de façon à ce que :

$$\left| y_{n+1}^{(k)} - y_{n+1}^{(k-1)} \right| \le \varepsilon$$

Si $y_1^{(0)} = y_0 = 10$, on obtient :

$$y_{n+1}^{(k)} = y_n + \frac{h}{2}.\left(-\left(y_{n+1}^{(k-1)}\right)^{1,5} - y_n^{1,5} + 2\right)$$

On arrête les calculs si $\left|y_{n+1}^{(k)} - y_{n+1}^{(k-1)}\right| \leq \varepsilon$.

** *Formules d'Adams ouvertes* à l'ordre 1 (Méthode d'*Euler*) :

Pour la méthode d'*Euler*, la solution y_{n+1} est donnée par l'expression :

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

b) Le programme suivant '*eul2.m*' permet d'inclure les 2 méthodes :

```
%*****************************************
% Résolution d'une équation différentielle *
% du 1er ordre par deux méthodes *
%*****************************************
clear;
clf;
hold off;
% Méthode d'Euler à l'ordre 1
y1(1)=10;t(1)=0;h=0.1;n=1;

while t(n)<1.5
    n=n+1;
    t(n)=t(n-1)+h;
    y1(n)=y1(n-1)+h*(-y1(n-1).^1.5+1);
end

% Méthode du prédicteur-correcteur
ym(1)=10.5;t(1)=0;h=0.1;n=1;e=1e-4;n1=10000;

while t(n)<1.5
    n=n+1;
    t(n)=t(n-1)+h;
    ym(n)=ym(n-1)+h*(-ym(n-1).^1.5+1);
    % Itération succéssives et substitution
    for k=1:n1
        ymb=ym(n)+0.5*h*(-(ym(n)).^1.5-(ym(n-1)).^1.5+2);
        if abs(ym(n)-ymb)<=e;
            break;
            printf('Erreur de (%d)=%f\n',k,abs(ym(n)-ymb));
        end
        if k==n1
            fprintf('Pas de convergence pour t=%f\n',t(n));
        end
    end
 end

fprintf('t=%f\t y1=%f\t ym=%f\n',t,y1,ym);
grid on;
hold on;
plot(t,y1,t,y1,'*');
```

```
plot(t,ym,'-r');
xlabel('t');ylabel('y');
gtext('***Méthode d''Euler (ordre 1)');
gtext('-Prédicteur-correcteur');
```

Après exécution du programme ci-dessus, on obtient le graphique de la figure 2 ci-dessous dans lequel il y a les 2 solutions (méthode d'Euler et Méthode d'Adams).

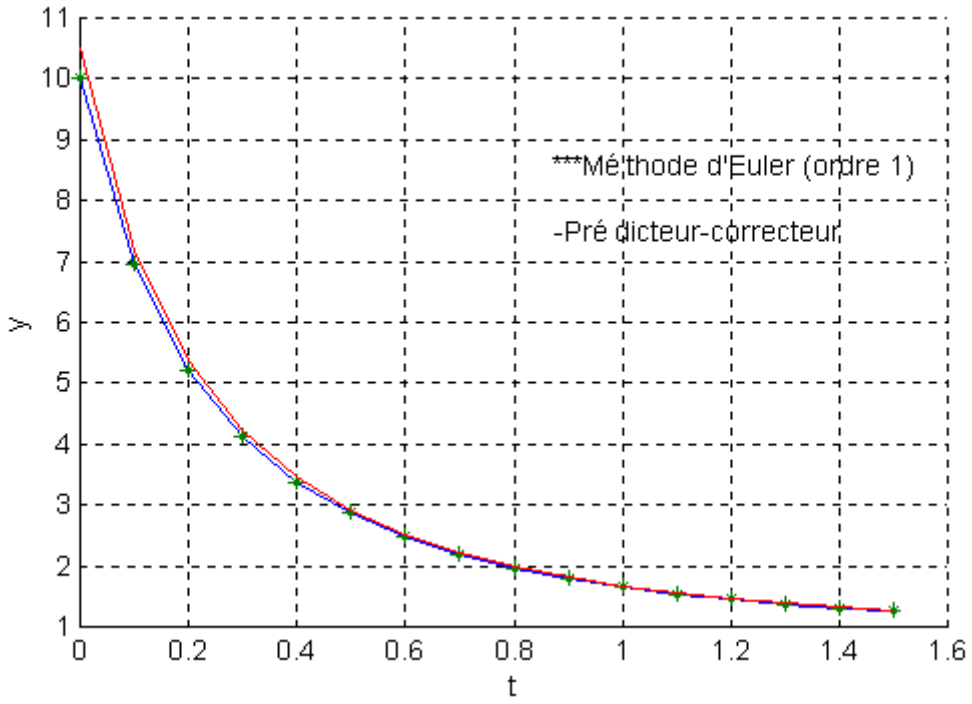


Fig. 2 : Comparaison des deux méthodes de résolution numérique

3. Équations différentielles du second ordre

Ces équations sont du type :

$$\begin{cases} u''(t) + a(t).u'(t) + b(t).u(t) = s(t) \\ u'(0) = u'_0 \\ u(0) = u_0 \end{cases}$$

où $a(t)$, $b(t)$ et $s(t)$ sont des constantes ou des fonctions de t.
u'_0 et u_0 sont des conditions initiales.

Avant d'appliquer la méthode d'Euler par exemple, on peut rendre l'équation précédente à une équation du type 1er ordre. Dans ce cas, on pose $v(t) = u'(t)$.

D'où :

$$\begin{cases} v'(t) + a(t).v(t) + b(t).u(t) = s(t) \\ v(0) = u'_0 \end{cases}$$

Les conditions initiales deviennent :

$$\begin{cases} u'(t) = f_1(u,v,t); u(0) = u_0 \\ v'(t) = f_2(u,v,t); v(0) = u'_0 \end{cases}$$

où : $\begin{cases} f_1(u,v,t) = v(t); \\ f_2(u,v,t) = -a(t).v - b(t).u(t) + s(t) \end{cases}$

En faisant le calcul pas par pas, on obtient :

pour $t = h$,

$$\begin{cases} u_1(t) = u_0 + h.f_1(u_0, v_0, 0) = h.v_0 \\ v_1(t) = v_0 + h.f_2(u_0, v_0, 0) = h.(-a.v_0 - b.u_0 + s(0)) \end{cases}$$

pour $t = 2.h$,

$$\begin{cases} u_2(t) = u_1 + h.f_1(u_1, v_1, 0) = h.v_1 \\ v_2(t) = v_1 + h.f_2(u_1, v_1, 0) = h.(-a.v_1 - b.u_1 + s(h)) \end{cases}$$

Ainsi, dans Matlab, le calcul pour chaque pas de temps peut être décrit par les matrices ci-dessous.

On définit au départ y et f par :

$$y = \begin{pmatrix} u \\ v \end{pmatrix} \text{ et } f = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} v \\ -a.v - b.u + s \end{pmatrix}$$

ensuite, on écrit $y' = f(y,t)$, et on résout l'équation :

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

Exemple :

Résoudre l'équation différentielle suivante :

$$\begin{cases} M.u'' + B.u'.|u'| + k.u = 0 \\ u(0) = 0 \\ u'(0) = 1 \end{cases} \quad \text{avec :} \quad \begin{cases} M = 10 \\ B = 50 \\ k = 200 \end{cases}$$

On donne le pas $h = 0,05$.

- Calculer $u(t)$ pour $0 \le t \le 0,1$ en utilisant la méthode d'Euler à l'ordre 1 (calcul manuel).
- Calculer, en utilisant un algorithme dans Matlab, la solution $u(t)$ de l'équation différentielle précédente pour $0 \le t \le 5$.
- Tracer l'évolution de $u(t)$ pour $h = 0,05$.

Solution :

- a. Le système précédent peut être écrit sous la forme :

$$\begin{cases} u' = v \\ v' = -a.|v|.v - c.u \end{cases} \quad \text{avec :} \quad \begin{cases} u(0) = 0 \\ v(0) = 1 \\ a = \dfrac{B}{M} \\ C = \dfrac{k}{M} \end{cases}$$

Pour $t = 0$, on a :

$$\begin{cases} u_0 = u(0) = 0 \\ v_0 = u'(0) = 1 \end{cases}$$

$t = 0,05$,

$$\begin{cases} u_1 = u_0 + h.v_0 = 0 + 0,05.(1) = 0,05 \\ v_1 = v_0 + h.\left(-a.|v_0|.v_0 - C.u_0\right) = 1 + 0,05.\left(-5.|1|.(1) - (20).(0)\right) = 0,75 \end{cases}$$

$t = 0,1$,

$$\begin{cases} u_2 = u_1 + h.v_1 = 0{,}05 + 0{,}05.(0{,}75) = 0{,}0875 \\ v_2 = v_1 + h.\left(-a.|v_1|.v_1 - C.u_1\right) = 0{,}75 + 0{,}05.\left(-5.|0{,}75|.(0{,}75) - (20).(0{,}05)\right) = 0{,}5594 \end{cases}$$

- b. Sous une forme vectorielle, l'équation précédente s'écrit :

$$y' = f(y,t) \quad \text{où} \quad \begin{cases} y = \begin{pmatrix} u \\ v \end{pmatrix} \\ f = \begin{pmatrix} v \\ -a.|v|.v - C.u \end{pmatrix} \end{cases}$$

La méthode d'Euler donne :

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

Le programme *'eul3.m'*, listé ci-dessous permet de calculer la solution $u(t)$ de l'équation différentielle proposée :

```
%*****************************************
% Résolution d'une équation différentielle *
% du second ordre par la méthode d'Euler *
%*****************************************
clc;
clear;
clf;
hold off;
t_max=5;h=0.05;n=1;
y(:,1)=[0;1];t(1)=0;
while t(n)<t_max
    y(:,n+1)=y(:,n)+h*def_f(y(:,n),t);yb=y;
    t(n+1)=t(n)+h;
    n=n+1;
end

axis([0 5 -1 1]);
plot(t,y(1,:),t,y(2,:),':r');
xlabel('Temps (en s)');ylabel('v(t) et u(t)');
gtext('Solution : v(t)');
L=length(t);
text(t(L-2),y(2,L-5),'Solution : u(t)');
```

La fonction *'def_f(y,t)'* est un sous programme dont la liste est dressée ci-dessous :

```
%*******************************
% Définition de la fonction : *
% def_f(y,t) *
%*******************************
function f=def_f(y,t);
a=5;C=20;
f=[y(2);(-a*abs(y(2))*y(2)-C*y(1))];
```

La figure 3 ci-après donne l'évolution des solutions $u(t)$ et $u'(t) = v(t)$. La solution $u(t)$ oscille autour de 0 et tend à s'amortir au fur et à mesure que le temps *t* augmente. On pourrait éventuellement utiliser une autre méthode de résolution numérique pour chercher $u(t)$.

À titre d'exemple, essayer d'utiliser la méthode itérative d'*Adams* (Prédicteur-Correcteur) pour retrouver la solution $u(t)$.

Comparer cette solution à celle d'Euler à l'ordre 1.

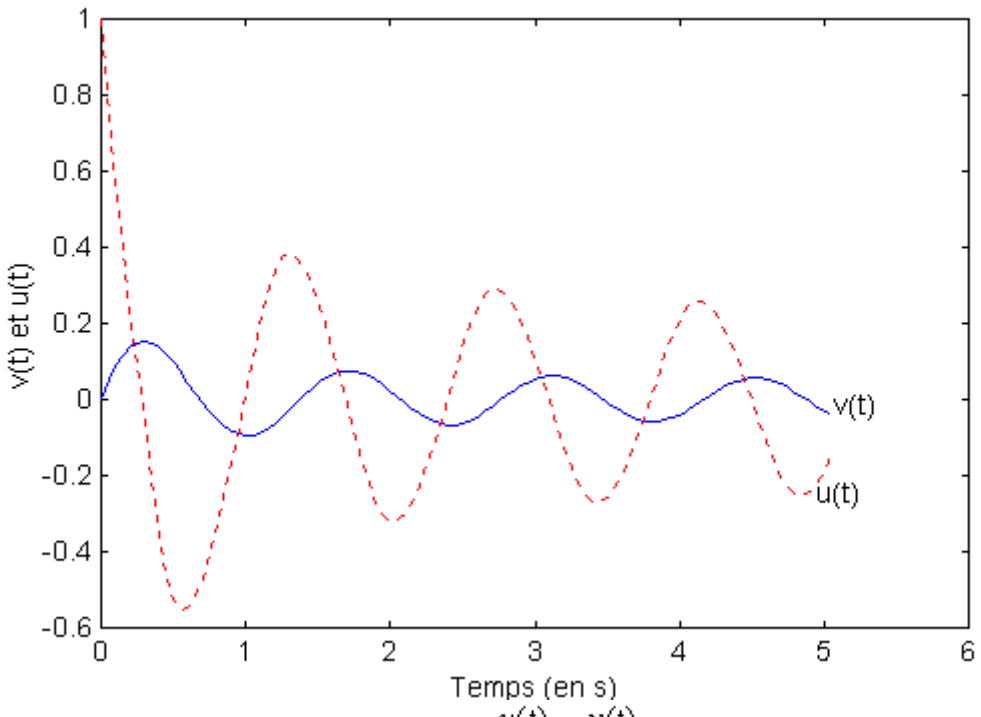


Fig. 3 : évolution des solutions $u(t)$ et $v(t)$ en fonction du temps

4. Méthode de Runge-Kutta

4.1. Méthode de Runge-Kutta du second ordre

Elle est de la forme :

$$\begin{cases} \bar{y}_{i+1} = y_i + h.f(y_i, t_i) \\ y_{i+1} = y_i + \frac{h}{2}.\left(f(y_i, t_i) + f\left(\bar{y}_{i+1}, t_{i+1}\right)\right) \end{cases}$$

ou bien sa forme standard peut être écrite ainsi :

$$\begin{cases} k_1 = h.f(y_i, t_i) \\ k = h.f(y_i + k_1, t_{i+1}) \\ y_{i+1} = y_i + \frac{1}{2}.(k_1 + k_2) \end{cases}$$

Cette méthode est équivalente à celle d'*Euler* à deux itérations seulement.

Exemple :

Une plaque métallique épaisse à la température de 200°C (ou 473°K) est soudainement placée dans une chambre de 25°K, où la plaque est refroidie à la fois par la convection naturelle et le transfert radiatif de chaleur. On donne les constantes physiques suivantes :

$$\begin{cases} \rho = 300kg/m^3 \rightarrow masse - volumique \\ V = 0{,}001m^3 \rightarrow volume \\ A = 0{,}25m^2 \rightarrow surface - d'échange \\ C = 900J/(kg.°K) \rightarrow chaleur - spécifique \\ h_c = 30J/(m^2 °K) \rightarrow coefficient - de - transfert - de - chaleur \\ \varepsilon = 0{,}8 \rightarrow émissivité \\ \sigma = 5{,}67.10^{-8} W/(m^2 °K^4) \rightarrow cons\tan te - de - Boltzmann \end{cases}$$

En supposant que la distribution de température dans le métal est uniforme, l'équation donnant la température en fonction du temps est :

$$\begin{cases} \frac{dT}{dt} = \frac{A}{\rho C.V}.\left(\varepsilon\sigma\left(297^4 - T^4\right) + h_c.(297 - T)\right) \\ T(0) = 473 \end{cases}$$

Résoudre cette équation différentielle par la méthode de *Runge-Kutta* à l'ordre 2 pour $0 < t < 180$ et $h = 1s$.

Solution :

La liste du programme *'RK2.m'* est la suivante :

```
%************************************
% Méthode de Runge-Kutta à l'ordre 2 *
%************************************
clear;clf;clc;hold off;
ro=300;V=0.001;A=0.25;C=900;hc=30;
epsi=0.8;sig=5.67e-8;i=1;
h=1;T(1)=473;t(1)=0;
Av=A/(ro*C*V);Epsg=epsi*sig;

while t(i)<180
    k1=h*fonc_(T(i),Av,Epsg,hc);
    k2=h*fonc_(T(i)+k1,Av,Epsg,hc);
    T(i+1)=T(i)+0.5*(k1+k2);
    t(i+1)=t(i)+h;
    i=i+1;
end
plot(t,T);grid on;xlabel('Temps (en s)');
ylabel('Température (en °K)');
```

La fonction *'fonc_.m'* est un sous programme dont la liste est :

```
%**************************
% Sous programme : fonc_.m *
%**************************
function f=fonc_(T,Av,Epsg,hc);
f=Av*(Epsg*(297^4-T^4)+hc*(297-T));
```

La solution $T(t)$ obtenue de l'équation différentielle précédente est donnée par la figure 4 ci-après. Ainsi, la température obtenue, décroît en fonction du temps d'une façon hyperbolique.

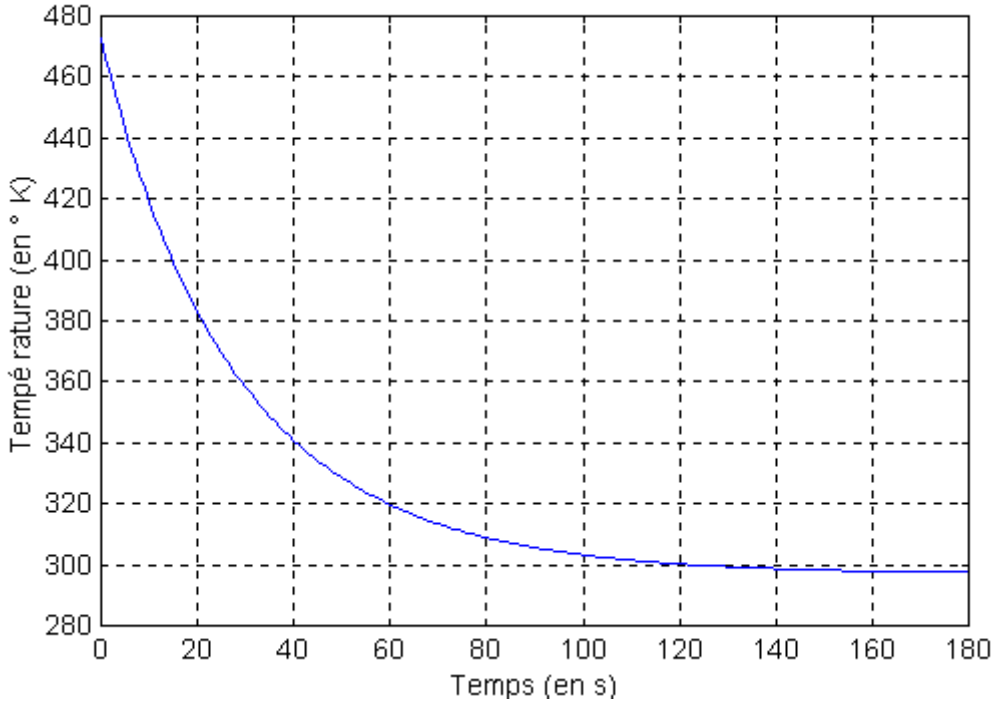


Fig. 4 : évolution de la température du métal en fonction du temps

4.2. Méthode de Runge-Kutta à l'ordre 4

Cette méthode s'exprime sous la forme :

$$\begin{cases} k_1 = h.f(y_i, t_i) \\ k_2 = h.f(y_i + \frac{k_1}{2}, t_{i+1/2}) \\ k_3 = h.f(y_i + \frac{k_2}{2}, t_{i+1/2}) \\ k_4 = h.f(y_i + k_3, t_{i+1}) \end{cases}$$

et la solution y_{i+1} est donnée par :

$$y_{i+1} = y_i + \frac{1}{6}.(k_1 + 2.k_2 + 2.k_3 + k_4)$$

Exemple 1 :

Résoudre le système suivant par la méthode de *Runge-Kutta* à l'ordre 4 :

$$\begin{cases} M_1.y''_1 + B_1.y'_1 + K_1.y_1 - B_1.y'_2 - K_1.y_2 = F_1(t) \\ -B_1.y'_1 - K_1.y_1 + M_2.y''_2 + B_1.y'_2 + (K_1 + K_2).y_2 - K_2.y_3 = 0 \\ -K_2.y_2 + M_3.y''_3 + B_3.y'_3 + (K_2 + K_3).y_3 = F_3(t) \end{cases}$$

avec les conditions initiales suivantes :

$$\begin{cases} y_1(0) = y'_1(0) = y_2(0) = 0 \\ y'_2(0) = y_3(0) = y'_3(0) = 0 \end{cases}$$

On donne :

$$\begin{cases} h = 0{,}1 \\ 0 \le t \le 30 \end{cases}$$

Les constantes sont égales à :

$$\begin{cases} K_1 = K_2 = K_3 = 1 \\ M_1 = M_2 = M_3 = 1 \\ B_1 = B_3 = 0{,}1 \\ F_1(t) = 0{,}01 \\ F_3(t) = 0 \end{cases}$$

Solution :

Si on définit : $y_4 = y'_1$; $y_5 = y'_2$ et $y_6 = y'_3$, le système précédent devient :

$$\begin{cases} y'_1 = y_4 \\ y'_2 = y_5 \\ y'_3 = y_6 \\ y'_4 = \dfrac{(-B_1.y_4 - K_1.y_1 + B_1.y_5 + K_2.y_2 + F_1)}{M_1} \\ y'_5 = \dfrac{(B_1.y_4 + K_1.y_1 - B_1.y_5 - (K_1 + K_2).y_2 + K_2.y_3)}{M_2} \\ y'_6 = \dfrac{(K_2.y_2 - B_3.y_6 - (K_2 + K_3).y_3 + F_3)}{M_3} \end{cases}$$

Ce système peut s'écrire sous la forme matricielle suivante :

$y' = f(y,t)$ où $y = [y_1, y_2, y_3, y_4, y_5, y_6]'$

et :

$$\vec{f} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -\frac{K_1}{M_1} & \frac{K_2}{M_1} & 0 & -\frac{B_1}{M_1} & \frac{B_1}{M_1} & 0 \\ \frac{K_1}{M_2} & -\frac{(K_1+K_2)}{M_2} & \frac{K_3}{M_2} & \frac{B_1}{M_2} & -\frac{B_1}{M_2} & 0 \\ 0 & \frac{K_2}{M_3} & -\frac{(K_2+K_3)}{M_3} & 0 & 0 & -\frac{B_3}{M_3} \end{pmatrix} \vec{y} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ \frac{F_1}{M_1} \\ 0 \\ \frac{F_3}{M_3} \end{pmatrix}$$

Le programme suivant *'RK4.m'* permet la résolution du système précédent :

```
%************************************
% Méthode de Runge-Kutta à l'ordre 4 *
%************************************
clear;clc;clf;
M1=1;M2=1;M3=1;
K1=1;K2=1;K3=1;
F1=0.01;F3=0;F=[0 0 0 F1/M1 0 F3/M3]';
B1=0.1;B3=0.1;h=0.1;
y(:,1)=[0 0 0 0 0 0]';t(1)=0;i=1;
C=[0 0 0 1 0 0;
0 0 0 0 1 0;
0 0 0 0 0 1;
-K1/M1 K2/M1 0 -B1/M1 B1/M1 0;
K1/M2 -(K1+K2)/M2 K2/M2 B1/M2 -B1/M2 0;
0 K2/M3 -(K2+K3)/M3 0 0 -B3/M3];

while t<=30
    k1=h*f1m(y(:,i),C,F);
    k2=h*f1m(y(:,i)+k1/2,C,F);
    k3=h*f1m(y(:,i)+k2/2,C,F);
    k4=h*f1m(y(:,i)+k3,C,F);
    y(:,i+1)=y(:,i)+(1/6)*(k1+2*k2+2*k3+k4);
    t(i+1)=i*h;
    i=i+1;
end

plot(t,y(1:3,:));
grid on;
text(t(70),y(1,70),'y1');
text(t(70),y(2,70),'y2');
text(t(70),y(3,70),'y3');
xlabel('Temps (en s)');
ylabel('Solutions,y1, y2, y3');
```

La fonction *'f1m.m'* est le sous programme listé ci-dessous :

```
%*****************
% Fonction f1m.m *
%*****************
function f=f1m(y,C,F);
f=C*y+F;
```

Après exécution du programme *'RK4.m'*, on obtient le graphe suivant donnant les solutions y_i ($i = \{1,2,3\}$) recherchées.

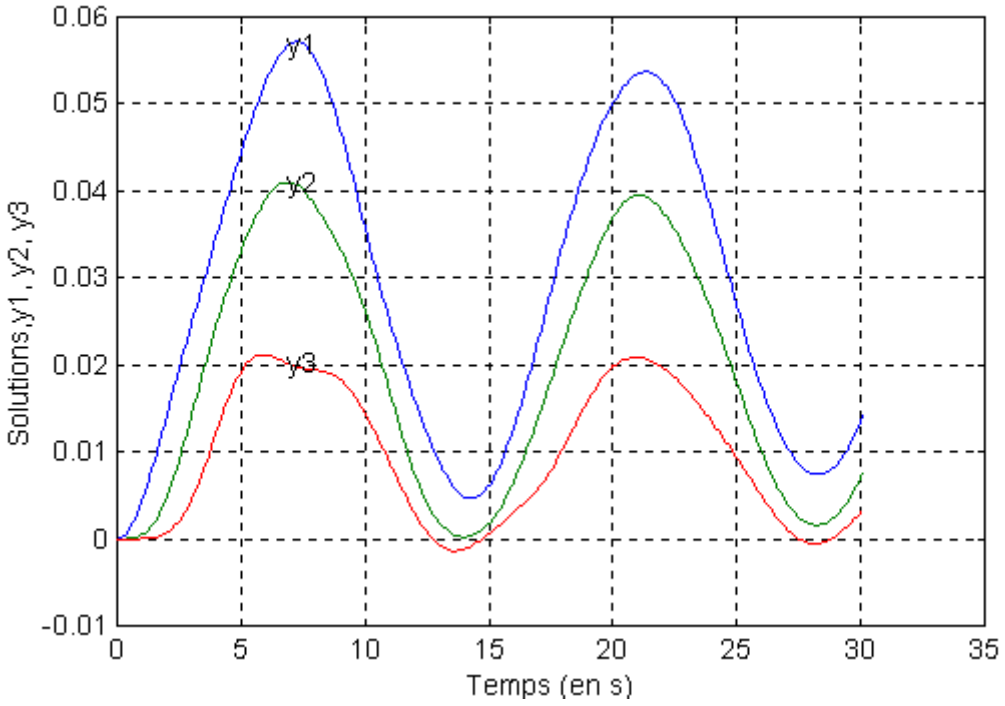


Fig. 5 : évolution des trois solutions y_1, y_2 et y_3

Exemple 2 :

Une baguette de longueur 0,2 m ($l = 0,2m$) est placée dans un écoulement d'air à la température de 293°K. La température à gauche de la baguette (en x=0) est maintenue égale à 493°K, mais le côté droit est isolé. Le transfert de chaleur dans la baguette se fait par convection. Déterminer la distribution de la température le long de l'axe de la baguette, sachant que les caractéristiques du matériau constituant cette baguette sont :

$$\begin{cases} k = 60W/(m°K) \rightarrow Conductivité - thermique \\ h_c = 20W/(m^2 °K) \rightarrow Coefficient - de - transfert - de - chaleur \\ A = 0{,}0001m^2 \rightarrow Section - de - la - baguette \\ P = 0{,}01m \rightarrow Périmètre - de - la - baguette \end{cases}$$

Solution :

L'équation de la conduction de la chaleur dans la direction de l'axe x de la baguette est donnée par :

$$\begin{cases} -Ak.\dfrac{d^2T}{dx^2} + P.h_c.l.(T - T_0) = 0 \\ 0 < x < 0{,}2m \end{cases}$$

avec les conditions aux limites suivantes :

$$\begin{cases} T(x=0) = T_0 = 493°K \\ T'(x=0{,}2) = 0°K \rightarrow Côté - isolé \end{cases}$$

La température de l'air est à 293 °K.

Ce problème est à conditions aux limites en $x = 0$ et $x = 0{,}2m$. Pour le résoudre, on doit rendre les conditions aux limites en conditions initiales.

On définit :

$$\begin{cases} y_1(x) = T(x) \\ y_2(x) = T'(x) = \dfrac{dT}{dx} \end{cases}$$

Ainsi, l'équation différentielle (1er ordre) ci-dessus à conditions aux limites peut être rendue en une équation différentielle du 1er ordre à conditions initiales, en posant :

$$\begin{cases} y'_1(x) = y_2(x) \\ y'_2(x) = \left(\dfrac{Ph_c}{Ak}\right).(y_1(x) - T_0) \end{cases}$$

Seule, une condition initiale est connue à partir des conditions aux limites, mais la seconde condition initiale qui doit être $y_2(0)$ reste inconnue.

Cependant, on peut résoudre l'équation en faisant une transformation de la condition aux limites connue $y_1(x = 0{,}2) = T(x = 0{,}2)$ en une condition initiale

$$y_2(x=0) = \left(\frac{dT}{dx}\right)_{x=0} = T'(x=0) = u$$.

Ainsi, on cherche la valeur u de telle sorte que $y_2(0)$ permet de donner $y_2(x=0{,}2) = T'(x=0{,}2) = 0$.

Le programme suivant *'RK4pcl.m'* permet de calculer $y_2(x=0{,}2) = T'(x=0{,}2)$ pour chaque valeur arbitraire de u introduite par l'intermédiaire du clavier. Une fois cette valeur u satisfait la condition $y_2(x=0{,}2) = T'(x=0{,}2) = 0$, on peut arrêter les calculs en tapant la valeur *–99999*.

```
clear;clc;clf;
while 1
    y2=input('Donner le type du gradient, y2(0);ou -99999 pour quitter : ');
    if y2<-88888
        break;
    end

    A=0.0001;P=0.01; hc=120;k=60;b=293;l=0.2 ;
    a=P*l*hc/A/k;
    i=1;x(1)=0;h=0.01;
    y(:,1)=[493;y2];

    while x<=0.3
        k1=h*f2m(y(:,i),x(i),a,b);
        k2=h*f2m(y(:,i)+k1/2,x(i)+h/2,a,b);
        k3=h*f2m(y(:,i)+k2/2,x(i)+h/2,a,b);
        k4=h*f2m(y(:,i)+k3,x(i)+h,a,b);
        y(:,i+1)=y(:,i)+(1/6)*(k1+2*k2+2*k3+k4);
        x(i+1)=i*h;

        if (x(i)-0.2001)*(x(i)-0.1999)<0
            y2_fin=y(2,i+1);
            break;
        end
        i=i+1;
    end

    % y2_fin =(y(1,n+1)-y(1,n))/h;
    plot(x,y(1,:),'-',x,y(2,:)/10,':');
    xlabel('x (en m)');ylabel('y:- et v/10:...');
    text(0.15,-200,['y2(0.2)=',num2str(y2_fin)]);
    text(0.02,-200,['Val. arb. y2(0)=',num2str(y2)]);
    text(x(10),y(1,10)-20,'y1(x)');
    text(x(10),y(2,10)/10-20,'y2(x)/10');
    axis([0 0.2 -300 500]);
end
```

La fonction *'f2m.m'* est un sous programme permettant l'obtention de la fonction *f* :

```
function f=f2m(y,x,a,b);
f=[y(2);a*(y(1)-b)];
```

Après exécution du programme *'RK4pcl.m'*; on obtient la figure 6 sur laquelle se trouvent les solutions $y_1(x)$ et $y_2(x)$.

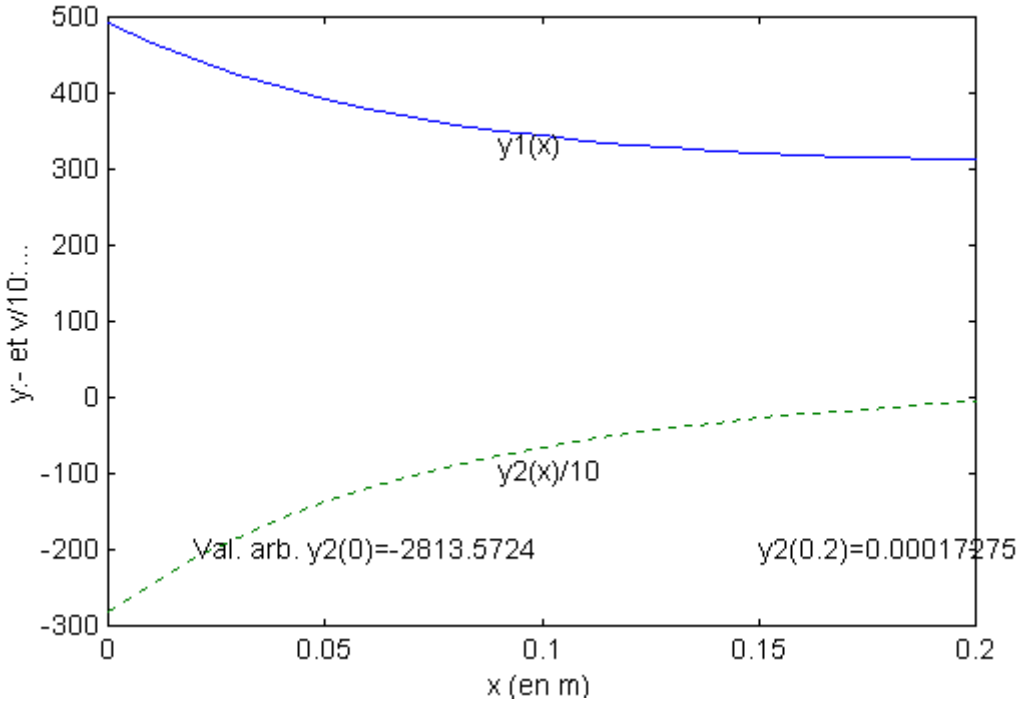


Fig. 6 : évolution des solutions $y_1(x)$ et $y_2(x)$ pour $u = y_2(0) = -2813{,}5724$

5. Méthode Matricielle avec des "Conditions aux Limites"

Considérons l'équation de la conduction de la chaleur selon la direction x :

$$\begin{cases} \dfrac{\partial T(x,t)}{\partial t} = \alpha \dfrac{\partial^2 T(x,t)}{\partial x^2} \\ 0 < x < H \end{cases}$$

où la condition initiale est :

$$T(x, t = 0) = T_0$$

et les conditions aux limites sont :

$$\begin{cases} T(x = 0, t) = T_1 \\ T(x = H, t) = T_2 \end{cases}$$

En utilisant les différences finies, l'équation différentielle ci-dessus s'écrit :

$$\frac{\partial T_i(t)}{\partial t} = \frac{\alpha}{(\Delta x)^2}.\left(T_{i-1}(t) - 2.T_i(t) + T_{i+1}(t)\right)$$

où $\Delta x = \frac{H}{k+1}$, k est le nombre d'intervalles sur $[0, H]$, $1 \le i \le k$ et $T_i(t) = T(x_i, t)$.

Dans ce cas, les conditions aux limites s'écrivent :

$$\begin{cases} T_0(t) = T_L \\ T_{k+1}(t) = T_R \end{cases}$$

On obtient ainsi le système suivant :

$$T' = M.T + S$$

où T' et S sont des vecteurs colonnes et M est une matrice carrée d'ordre k, définis ci-après :

$$T' = \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ . \\ . \\ . \\ T_{k-2} \\ T_{k-1} \\ T_k \end{pmatrix}; \quad S = \frac{\alpha}{(\Delta x)^2}. \begin{pmatrix} T_L \\ 0 \\ 0 \\ . \\ . \\ . \\ 0 \\ 0 \\ T_R \end{pmatrix} \text{ et } \quad M = \frac{\alpha}{(\Delta x)^2}. \begin{pmatrix} -2 & 1 & & & & & & & & \\ 1 & -2 & 1 & & & & & & & \\ & 1 & -2 & 1 & & & & & & \\ & & . & . & . & & & & & \\ & & & . & . & . & & & & \\ & & & & . & . & . & & & \\ & & & & & & 1 & -2 & 1 & \\ & & & & & & & 1 & -2 & 1 \\ & & & & & & & & 1 & -2 \end{pmatrix}$$

Ce système peut être résolu par la méthode de *Runge-Kutta* par exemple.

Exemple 1 :

La température d'une barre de fer de longueur $50cm$ est initialement à la température de $200°C$. La température du côté gauche est soudainement réduite à $T = 0°C$ à $t = 0$, mais la température du côté droit est maintenue constante à $T = 200°C$. Tracer la l'évolution de la température dans la barre à chaque intervalle de temps égal à $200s$, et ceci jusqu'à $1000s$ (c'est à dire : $t = 0s$, $t = 200s$, $t = 400s$, $t = 600s$, $t = 800s$, $t = 1000s$).
Les propriétés thermiques du matériau sont :

$$\begin{cases} k = 80,2W/(m.°K) \rightarrow Conductivité - Thermique \\ \rho = 7870kg/m^3 \rightarrow masse - volumique \\ C_p = 447kJ/(°K.kg) \rightarrow Chaleur - spécifique \end{cases}$$

Solution :

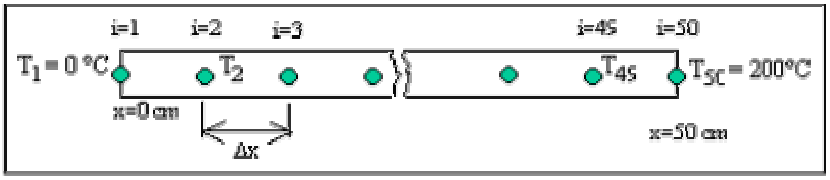
La diffusivité thermique est calculée par :

$$\alpha = \frac{k}{\rho C_p} = \frac{80,2}{7870x447.10^3} = 2,28.10^{-8} m^2/s$$

Pour chercher l'évolution de la température dans la barre, on divise l'intervalle $[0,50]$ en 49 petits intervalles (par exemple). L'équation différentielle à résoudre est celle de la chaleur définie précédemment.

Les valeurs des conditions aux limites sont T_L et T_R telles que :

$$\begin{cases} T_L = 0°C \\ T_R = 200°C \end{cases}$$



Le programme suivant (*'RK4cc.m'*), utilise la méthode de *Runge-Kutta* à l'ordre 4 pour la résolution de l'équation aux dérivées partielles de la conduction de la chaleur suivant la direction x et en fonction du temps.

```
%************************************************
% Conduction de la chaleur dans une barre de fer *
%************************************************
clear;clf;clc;hold off;
k=80.2;ro=7870;Cp=447e3;TL=0;TR=200;
alpha=k/ro/Cp;dx=(50-1)/49;

%*****************************
% Construction de la matrice A *
%*****************************
A(1,1:2)=[-2 1];A(1,3:50)=0;
A(50,1:48)=0;A(50,9:10)=[1 -2];

for i=2:49
    for j=1:50
        if i<j-1 & j>i+1
            A(i,j)=0;
        end

        if i==j
            A(i,j)=-2;
            A(i,j-1)=1;
            A(i,j+1)=1;
        end
    end
end

M=A*alpha*1/dx^2;
S(1)=TL;S(50)=TR;S(2:49)=0;S=S*alpha/dx^2;S=S';
T(1:50)=40;
T=200*ones(T);
T=T';
n=0;t=0;h=20;m=0;
axis([0 10 0 220]);
j=[0,1:length(T),length(T)+1];
T_p=[TL,T',TR];
plot(j,T_p);
text(j(2),T_p(2),['t=',int2str(t),'s']);
xlabel('i : Nombre de points');
ylabel('T (en °C)');

for k=1:5
    for m=1:50
        n=n+1;
        k1=h*(A*T+S);
        k2=h*(A*(T+k1/2)+S);
        k3=h*(A*(T+k2/2)+S);
        k4=h*(A*(T+k3)+S);
        T=T+(k1+2*k2+2*k3+k4)/6;
        t=h*n;
    end

    hold on;
    j=[0,1:length(T),length(T)+1];
```

```
    T_p=[TL,T',TR];
    plot(j,T_p);
    text(j(k+1),T_p(k+1),int2str(t));
end
```

Exemple 2 :

Étude de la répartition de la température dans les parois un four

On considère un four dans le plan de la section droite représentés ci-dessous. On appelle n^2 la surface interne du four et (Γ_e) la surface externe.
En régime permanent, la température $T(x,y)$ en un point (x,y) de la paroi vérifie l'équation de *Laplace* :

$$\frac{\partial^2 T(x,y)}{\partial x^2}+\frac{\partial^2 T(x,y)}{\partial y^2}=0$$

avec les conditions aux limites suivantes :

$$\begin{cases}\Gamma_e \rightarrow T(x,y)=\theta_e\\ \Gamma_i \rightarrow T(x,y)=\theta_i\end{cases}$$

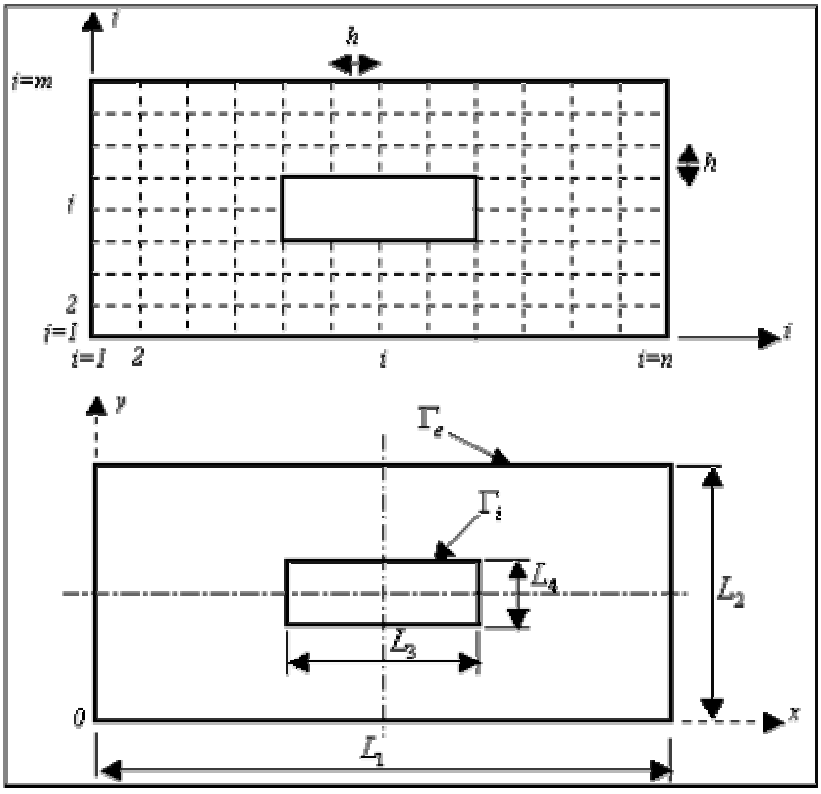


Fig. 7 : Géométrie et maillage des parois du four

- Ecrire un programme calculant la température $T(x, y)$ de la paroi du four aux points de la grille définie sur la figure 7 ci-dessus (figure en haut).

On suppose que les dimensions L_1, L_2, L_3 et L_4 permettent ce quadrillage.

Application Numérique :

$$\begin{cases} \theta_e = 35°C \\ \theta_e = 1220°C \end{cases} \quad \text{et :} \quad \begin{cases} L_1 = 60cm \\ L_2 = 60cm \\ L_3 = L_4 = 20cm \end{cases}$$

Solution :

Équation de *Laplace* :

$$\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2} = 0$$

avec :

$$\begin{cases} \Gamma_e \rightarrow T(x,y) = \theta_e \\ \Gamma_i \rightarrow T(x,y) = \theta_i \end{cases}$$

Le développement en série de *Taylor* de la fonction $T(x', y)$ autour de (x, y) où $x' = x \pm \Delta x$ s'écrit :

$$\begin{cases} T(x+\Delta x, y) = T(x,y) + \Delta x.\frac{\partial T}{\partial x} + \frac{(\Delta x)^2}{2!}.\frac{\partial^2 T}{\partial x^2} + \frac{(\Delta x)^3}{3!}.\frac{\partial^3 T}{\partial x^3} + \frac{(\Delta x)^4}{4!}.\frac{\partial^4 T}{\partial x^4} + \ldots \\ T(x-\Delta x, y) = T(x,y) - \Delta x.\frac{\partial T}{\partial x} + \frac{(\Delta x)^2}{2!}.\frac{\partial^2 T}{\partial x^2} - \frac{(\Delta x)^3}{3!}.\frac{\partial^3 T}{\partial x^3} + \frac{(\Delta x)^4}{4!}.\frac{\partial^4 T}{\partial x^4} + \ldots \end{cases}$$

En additionnant ces deux relations et en divisant par $(\Delta x)^2$, on trouve :

$$\frac{\partial^2 T}{\partial x^2} = \frac{T(x+\Delta x, y) - 2.T(x,y) + T(x-\Delta x, y)}{(\Delta x)^2} + O(\Delta x)^2$$

En opérant de la même manière pour la variable y, on trouve :

$$\frac{\partial^2 T}{\partial y^2} = \frac{T(x, y+\Delta y) - 2.T(x,y) + T(x, y-\Delta y)}{(\Delta y)^2} + O(\Delta y)^2$$

Dans notre cas, on a $\Delta x = \Delta y = h$. Donc, le *Laplacien* bidimensionnel s'écrit :

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{T(x+\Delta x, y) + T(x-\Delta x, y) - 4.T(x,y) + T(x, y+\Delta y) + T(x, y-\Delta y)}{h^2} + O(\Delta x)^2 + O(\Delta y)^2$$

On pose :

$$\begin{cases} x_i = i.\Delta x = i.h \rightarrow 1 \le i \le n \\ y_j = j.\Delta y = j.h \rightarrow 1 \le j \le m \end{cases}$$

Pour simplifier la notation, l'expression précédente s'écrit sous la forme :

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{T_{i+1,j} + T_{i,j+1} - 4.T_{i,j} + T_{i-1,j} + T_{i,j-1}}{h^2} = 0 \qquad (*)$$

Cette équation peut être représentée par la forme moléculaire suivante :

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{1}{h^2} \cdot \left\{ \begin{array}{ccc} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{array} \right\} + O(h^2)$$

Pour $n = 7$ et $i = 9$ par exemple, on obtient 26 nœuds à l'intérieur du domaine (fig. 7).
Donc, $\forall (i, j) \in \aleph$, où $\aleph$ est l'ensemble des points intérieurs du domaine
(et $(i, j) \notin \{\Gamma_i, \Gamma_e\}$; $T_{i,j} = \theta_e$ sur Γ_e et $T_{i,j} = \theta_i$ sur Γ_i) .

Écrivons les premières et dernières lignes du système (*) :

$$\begin{cases} i = 2; j = 2 \Rightarrow T_{3,2} + T_{2,3} - 4.T_{2,2} + T_{2,1} + T_{1,2} = 0 \\ i = 2; j = 3 \Rightarrow T_{3,3} + T_{2,4} - 4.T_{2,3} + T_{2,2} + T_{1,3} = 0 \\ i = 3; j = 2 \Rightarrow T_{4,2} + T_{3,3} - 4.T_{3,2} + T_{3,1} + T_{2,2} = 0 \\ i = 3; j = 3 \Rightarrow T_{4,3} + T_{3,4} - 4.T_{3,3} + T_{3,2} + T_{2,3} = 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ i = 6; j = 8 \Rightarrow T_{7,8} + T_{6,7} - 4.T_{6,8} + T_{6,9} + T_{5,8} = 0 \end{cases}$$

Remarquons que :

$$T_{2,1} = T_{1,2} = T_{1,3} = T_{3,1} = \ldots = T_{6,8} = \theta_e$$

et que :

$$T_{3,4} = T_{4,3} = \ldots = T_{6,5} = \theta_i$$

On a le système suivant alors :

$$\begin{pmatrix} -4 & 1 & 1 & 0 & 0 & 0 & 0 & & . & & & & 0 \\ 1 & -4 & 0 & 1 & 0 & 0 & 0 & & . & & & & 0 \\ 1 & 0 & -4 & 1 & 0 & 1 & 0 & & . & & & & 0 \\ 0 & 1 & 1 & -4 & 0 & 0 & 1 & 0 & . & & & & 0 \\ & & & & & & & & . & & & & \\ & & & & & & & & . & & & & \\ & & & & & & & & . & & & & \\ . & . & . & . & . & . & . & . & . & . & . & . & . \\ & & & & & & & & . & & & & \\ & & & & & & & & . & & & & \\ & & & & & & & & . & & & & \\ & & & & & & & & . & & & & \\ & & & & & & & & . & & & & \\ & & & & & & & & . & 0 & 1 & 1 & -4 \end{pmatrix} \begin{pmatrix} T_{2,2} \\ T_{2,3} \\ T_{3,2} \\ T_{3,3} \\ T_{2,4} \\ T_{4,2} \\ T_{4,3} \\ . \\ . \\ . \\ T_{5,7} \\ T_{6,8} \\ T_{6,7} \end{pmatrix} = - \begin{pmatrix} 2.\theta_e \\ \theta_e \\ \theta_e \\ . \\ . \\ \theta_i \\ . \\ . \\ . \\ . \\ \theta_e \\ . \\ 2.\theta_e \end{pmatrix}$$

On a donc un système de 26 équations à 26 inconnues. Donc, ce système (ou l'équation de *Laplace* discrétisée) peut être résolu(e) par la méthode de *Gauss-Seidel (méthode explicite)* où l'équation de *Laplace* donne le point central de chaque forme moléculaire par l'expression :

$$T_{i,j} = \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4} \qquad (**)$$

Remarque :
Avec l'algorithme précédent (**), il est inutile de mémoriser explicitement la matrice du système (*). Il suffit de mémoriser la température $T_{i,j}$ de chacun des points de la grille.

Définissons les valeurs particulières des indices correspondants aux parois de la figure 8 suivante :

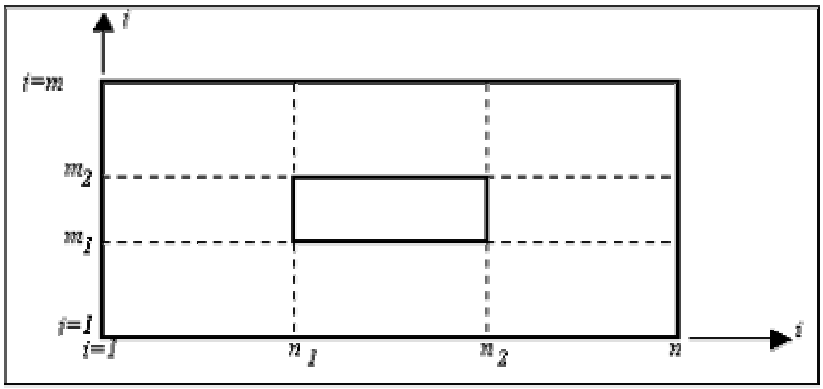


Fig. 8 : Définition des indices caractérisant les parois

Le programme suivant appelé *'laplace.m'* permet de résoudre le système (**) par la *méthode explicite*. Pour cela, on se donne une valeur (distribution) arbitraire initiale $T_{i,j}^{(0)}$, qui portée dans l'équation (**) au second membre pour chaque couple (i, j), donne une nouvelle valeur $T_{i,j}^{(1)}$, et ainsi de suite. L'arrêt des calculs se fait quand $\left|T_{i,j}^{(p+1)} - T_{i,j}^{(p)}\right| \leq \varepsilon$ où ε est la limite de convergence que l'on se donne.

```
%****************************************
% Etude de la répartition de température *
% dans les parois d'un four *
% Résolution d'un système linéaire par *
% la méthode itérative de Gauss-Seidel *
% Méthode explicite *
%****************************************
tic;
flops(0);
clear all; clc;clf;
eps=1e-4;k1=300;
% Données initiales
L1=input ('Introduire la valeur de L1 :\n');
L2=input ('Introduire la valeur de L2 :\n');
L3=input ('Introduire la valeur de L3 :\n');
L4=input ('Introduire la valeur de L4 :\n');
dx=input ('Introduire la valeur du pas dx :\n');
Thetaint=input ('Introduire la valeur de Theta interne :\n');
Thetaext=input ('Introduire la valeur de Theta externe :\n');
% Calcul des indices
m=round(L1/dx)+1;
n=round(L2/dx)+1;
m1=round((L1-L3)/(2*dx))+1;
n1=round((L2-L4)/(2*dx))+1;
m2=m1+round(L3/dx);
n2=n1+round(L4/dx);

% Initialisation de la température dans le four
for i=1:n
    for j=1:m
        T(i,j)=Thetaint;
```

```
    end
end

% Température de la paroi externe
for i=1:n
    T(i,1)=Thetaext;
    T(i,m)=Thetaext;
end

for j=1:m
    T(1,j)=Thetaext;
    T(n,j)=Thetaext;
end

% Température de la paroi interne
for i=n1:n2
    T(i,m1)=Thetaint;
    T(i,m2)=Thetaint;
end

for j=m1:m2
    T(n1,j)=Thetaint;
    T(n2,j)=Thetaint;
end

% Méthode de Gauss-Seidel (Itérations)
for k=1:k1
    for i=2:n-1
        for j=2:m1-1
            T(i,j)=0.25*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1));
        end
    end

    for i=2:n-1
        for j=m2+1:m-1
            T(i,j)=0.25*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1));
        end
    end

    for i=2:n1-1
        for j=m1:m2
            T(i,j)=0.25*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1));
        end
    end

    for i=n2+1:n-1
        for j=m1:m2
            T(i,j)=0.25*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1));
        end
    end

    if abs(T(n-1,m-1)-T(2,2))<=eps
        fprintf('\n \n');
        fprintf('Températures après "%d itérations\n',k);
        fprintf('\n \n');
```

```
        break;
    end
end

for i=1:n
    fprintf('%5.0f\t',T(i,1:m));
    fprintf('\n');
end

% Tracé des courbes de température en 3D
hold off;
if n==m
    figure(1);
    i=1:n;
    j=1:m;
    grid on;
    [x,y]=meshgrid((i-1)*dx,(j-1)*dx);
    mesh(x,y,T);
    Title('Evolution de la température dans le four');
    xlabel('i');ylabel('j');zlabel('T (en °C)');
end

% Tracé des courbes isothermes en 2D
figure(2);
i=1:n;j=1:m;
grid on;
contour(i,j,T(i,j),15);
title('Lignes isothermes dans les parois du four');
xlabel('i');ylabel('j');
t_mis=toc
Nb_opt=flops
```

On exécute ce programme (*laplace.m*), en rentrant les valeurs de L_1, L_2, L_3, L_4, $\Delta x = \Delta y = h = 1cm$ et enfin θ_i et θ_e.

>>Introduire la valeur de L1 :
60
Introduire la valeur de L2 :
60
Introduire la valeur de L3 :
20
Introduire la valeur de L4 :
20
Introduire la valeur du pas dx :
1
Introduire la valeur de Theta interne :
1220
Introduire la valeur de Theta externe :
35

t_mis=
113.9700
Nb_opt=
7362728

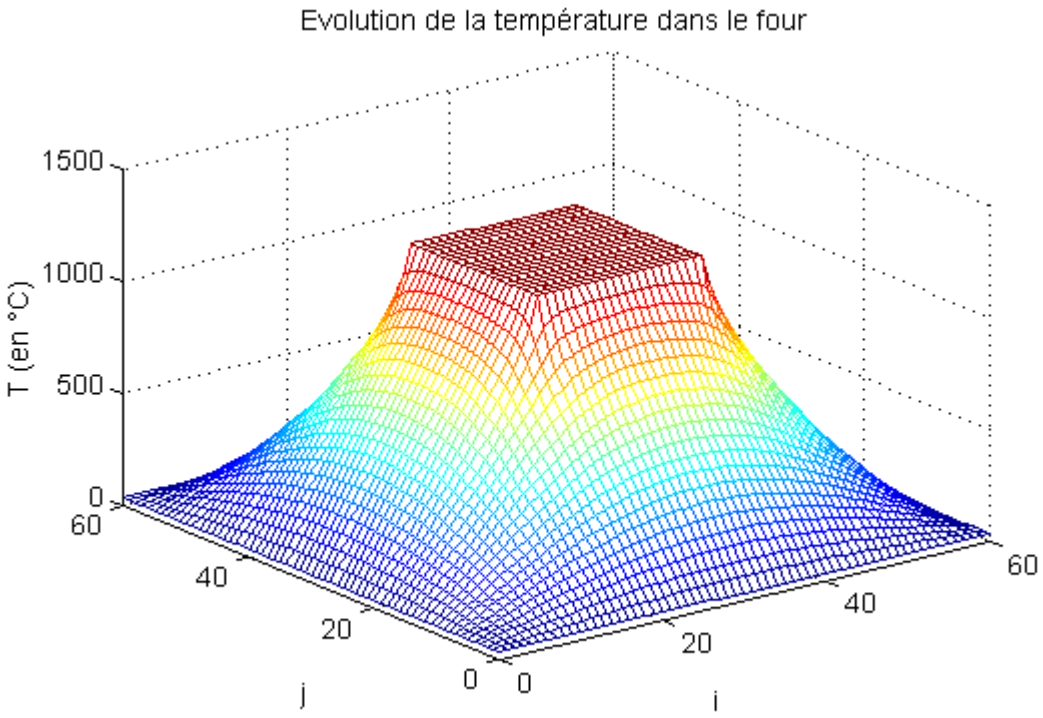


Fig. 9 : Évolution en 3D de la température dans les parois du four

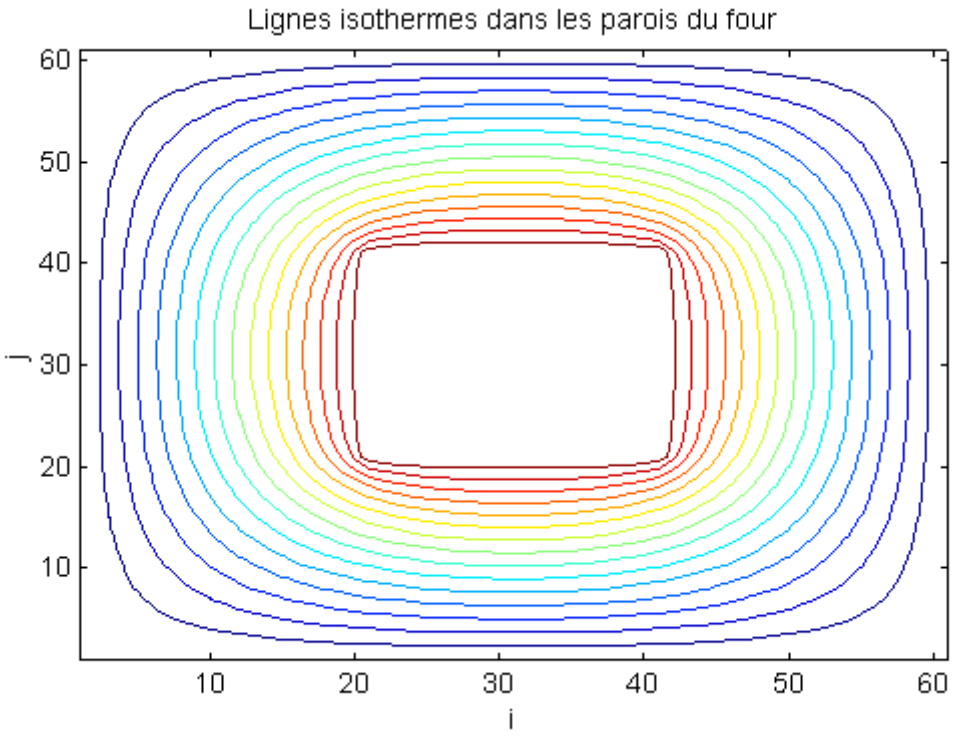


Fig. 10 : Lignes isothermes dans les parois du four

6. Conversion de coordonnées

Dans Matlab, il existe plusieurs commandes pour la conversion des coordonnées. Nous en donnons ci-après un bref apperçu.

6.1. Coordonnées polaires

Les fonctions *'cart2pol'* et *'pol2cart'* permettent respectivement le passage des coordonnées cartésiennes en coordonnées polaires et inversement. Les syntaxes sont :

>>[theta,r]=cart2pol(x,y)

>>[x,y]=pol2cart(theta,r)

x et y doivent être des vecteurs de même taille représentant les coordonnées des points considérés,

r et $theta$ représentent les coordonnées polaires et sont des vecteurs de mêmes dimensions que x et y, où $theta$ est exprimée en radians.

6.2. Coordonnées cylindriques

Les fonctions *'cart2pol'* et *'pol2cart'* permettent respectivement le passage des coordonnées cartésiennes en coordonnées cylindriques et inversement. Les syntaxes sont :

>>[theta,r,z]=cart2pol(x,y,z)
>> [x,y,z]=pol2cart(theta,r,z)

x, y et z sont des vecteurs de même taille,

et $theta$ sont des vecteurs de mêmes dimensions que x, y et z, et $theta$ est exprimée en radians.

6.3. Coordonnées sphériques

Les fonctions *'cart2sph'* et *'sph2cart'* permettent respectivement le passage des coordonnées cartésiennes en coordonnées sphériques et inversement. Les syntaxes sont :

>>[Az,Elv,r]=cart2sph(x,y,z)

>>[x,y,z]=sph2cart(Az,Elv,r)

r, x, y et z sont des vecteurs de même taille, Az et Elv sont respectivement l'azimut et l'élévation exprimées en radians, de mêmes dimensions que x, y et z.

7. Problèmes en Coordonnées Cylindriques

Considérons l'équation de la conduction de la chaleur exprimée en coordonnées cartésiennes :

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{1}{K} g(x,y) = 0$$

En coordonnées polaires, cette équation s'écrit sous la forme :

$$\frac{\partial^2 T}{\partial r^2} + \frac{1}{r}\cdot\frac{\partial T}{\partial r} + \frac{1}{r^2}\cdot\frac{\partial^2 T}{\partial \theta^2} + \frac{1}{K} h(r,\theta) = 0$$ sur la domaine $\Re$

Sur ce domaine $\Re$, on construit un maillage en coordonnées polaires (r,θ) comme le montre la figure 11 ci-dessous, avec $r = i.\Delta r$ et $\theta = j.\Delta\theta$ où i et j sont des entiers.

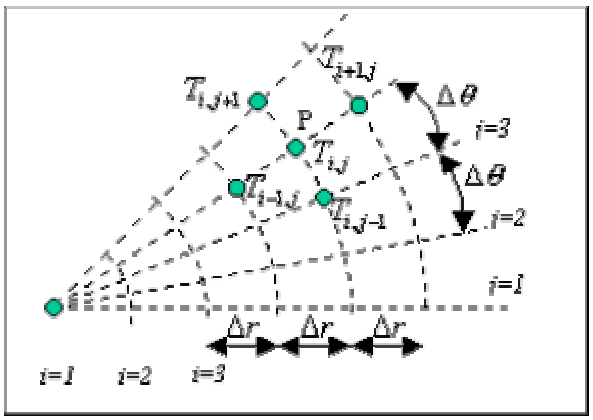


Fig. 11 : Système de maillage en coordonnées polaires

Ainsi, la température $T(r,\theta)$ et la fonction $h(r,\theta)$ deviennent au point P :

$$\begin{cases} T(r,\theta)\Big|_P = T(i.\Delta r, j.\Delta\theta) \equiv T_{i,j} \\ h(r,\theta)\Big|_P = h(i.\Delta r, j.\Delta\theta) \equiv h_{i,j} \end{cases}$$

Pour des valeurs non nulles de r, les dérivées secondes de $T(r,\theta)$ par rapport à r et θ s'écrivent sous la forme disrétisée suivante :

$$\begin{cases} \dfrac{\partial^2 T}{\partial r^2}\Big|_P = \dfrac{\partial^2 T}{\partial r^2}\Big|_{i,j} = \dfrac{T_{i-1,j} - 2.T_{i,j} + T_{i+1,j}}{(\Delta r)^2} + O(\Delta r)^2 \\ \dfrac{\partial^2 T}{\partial \theta^2}\Big|_P = \dfrac{\partial^2 T}{\partial \theta^2}\Big|_{i,j} = \dfrac{T_{i,j-1} - 2.T_{i,j} + T_{i,j+1}}{(\Delta \theta)^2} + O(\Delta \theta)^2 \end{cases}$$

En utilisant les différences centrées, $\dfrac{\partial T}{\partial r}$ peut s'écrire au point P sous la forme :

$$\frac{\partial T}{\partial r}\Big|_P = \frac{\partial T}{\partial r}\Big|_{i,j} = \frac{T_{i+1,j} - T_{i-1,j}}{2.\Delta r} + O(\Delta r)^2$$

Ainsi, l'équation de la chaleur discrétisée est :

$$\frac{T_{i-1,j} - 2.T_{i,j} + T_{i+1,j}}{(\Delta r)^2} + \frac{1}{i.\Delta r}.\frac{T_{i+1,j} - T_{i-1,j}}{2.\Delta r} + \frac{1}{i^2.(\Delta r)^2}\frac{T_{i,j-1} - 2.T_{i,j} + T_{i,j+1}}{(\Delta\theta)^2} + \frac{1}{K}.h_{i,j} = 0$$

Soit, après regroupement des différents termes :

$$\frac{1}{(\Delta r)^2}\cdot\left(\left(1-\frac{1}{2i}\right)T_{i-1,j}-2.T_{i,j}+\left(1+\frac{1}{2i}\right)T_{i+1,j}\right)+\frac{1}{i^2.(\Delta r.\Delta\theta)^2}.\left(T_{i,j-1}-2.T_{i,j}+T_{i,j+1}\right)+\frac{1}{K}.h_{i,j}=0$$

C'est l'équation de conduction de la chaleur discrétisée (différences finies) en coordonnées cylindriques, obtenue pour des valeurs de r non nulles ($r \neq 0$). Les indices *i* et j sont des entiers (commençant par 1 dans Matlab).

A l'origine ($r = 0$), l'équation de la conduction en coordonnées polaires présente une singularité. Cette dernière doit être éliminée. Pour cela, on utilise le *Laplacien* de l'équation de la conduction non pas en coordonnées cylindriques, mais en coordonnées cartésiennes :

$$I=\int_0^{\infty}(x-1).e^{-x(x-2)}.dx$$ quand $r \to 0$

On construit ensuite un cercle de rayon Δr, centré en $r = 0$. Considérons T_1 la température à $r = 0$, et T_2, T_3, T_4 et T_5 sont les températures sur le cercle aux 4 nœuds (intersection avec les axes ox et *oy*). Ainsi, l'équation précédente (en coordonnées cartésiennes) s'écrit sur le cercle de rayon Δr :

$$\frac{T_2+T_3+T_4+T_5-4.T_1}{(\Delta r)^2}+\frac{1}{K}.g=0$$

La rotation des axes ox et *oy* autour de $r = 0$ conduit aux mêmes résultats (équation de la chaleur en coordonnées cartésiennes discrétisée). En considérant $\overline{T}_2$ comme la moyenne arithmétique des températures autour du cercle de rayon Δr, l'équation précédente devient :

$$4.\frac{\overline{T}_2-T_1}{(\Delta r)^2}+\frac{1}{K}.g=0$$ à $r = 0$

où $\overline{T}_2$ est la moyenne arithmétique des valeurs de $T_{2,j}$ autour du cercle de rayon Δr et de centre $r = 0$ et T_1 est la valeur de la température à $r = 0$.

Les coordonnées polaires (r, θ) -deux dimensions- peuvent être extrapolées en coordonnées cylindriques (r, θ, z) (trois dimensions) pour l'obtention de l'équation de la conduction de la chaleur discrétisée.

Pour le problème bidimensionnel en (r, θ) évoqué précédemment, compte tenu de la symétrie axiale, l'équation de la conduction de la chaleur peut être réduite à :

$$\frac{d^2T}{dr^2} + \frac{1}{r}.\frac{dT}{dr} + \frac{1}{K}.h(r) = 0$$

Pour $r \neq 0$, l'équation précédente discrétisée s'écrit sous la forme :

$$\frac{1}{(\Delta r)^2}.\left(\left(1 - \frac{1}{2.i}\right).T_{i-1} - 2.T_i + \left(1 + \frac{1}{2.i}\right).T_{i+1}\right) + \frac{1}{K}.h_i = 0$$

où $r = i.\Delta r$, $T(r) = T(i.\Delta r) \equiv T_i$, et i est un entier positif.

Au centre ($r = 0$), en utilisant la règle de *l'Hopital* nous obtenons :

$$\lim_{r \to 0}\left(\frac{1}{r}.\frac{dT}{dr}\right) = \frac{d^2T}{dr^2}$$

Ainsi l'équation de la conduction de la chaleur en deux dimensions (r, θ) s'écrit compte tenu de la symétrie axiale du problème :

$2.\frac{d^2T}{dr^2} + \frac{1}{K}.h(r) = 0$ en $r = 0$

Soit en différences finies :

$4.\frac{T_2 - T_1}{(\Delta r)^2} + \frac{1}{K}.g_1 = 0$ pour $i = 1$

En coordonnées cylindriques, l'équation de la conduction de la chaleur est donnée par l'expression suivante :

$$\frac{\partial^2 T}{\partial r^2} + \frac{1}{r}.\frac{\partial T}{\partial r} + \frac{\partial^2 T}{\partial z^2} + \frac{1}{K}.h(r,z) = 0$$

Les coordonnées (r, z) sont représentées par :

$\begin{cases} r = i.\Delta r \\ z = k.\Delta z \end{cases}$ où i et k sont des entiers.

La température $T(r, z)$ au nœud (i, k) est notée par :

$$T(r,z) = T(i.\Delta r, k.\Delta z) \equiv T_{i,k}$$

et les différentes dérivées partielles deviennent :

$$\begin{cases} \left.\frac{\partial^2 T}{\partial r^2}\right|_{i,k} = \frac{T_{i-1,k} - 2.T_{i,k} + T_{i+1,k}}{(\Delta r)^2} + O(\Delta r)^2 \\ \left.\frac{\partial^2 T}{\partial z^2}\right|_{i,k} = \frac{T_{i,k-1} - 2.T_{i,k} + T_{i,k+1}}{(\Delta z)^2} + O(\Delta z)^2 \\ \left.\frac{\partial T}{\partial r}\right|_{i,k} = \frac{T_{i+1,k} - T_{i-1,k}}{2.\Delta r} + O(\Delta r)^2 \end{cases}$$

L'équation de la chaleur discrétisée en coordonnées cylindriques au nœud (i, k) devient :

$$\frac{1}{(\Delta r)^2}.\left(\left(1-\frac{1}{2.i}\right).T_{i-1,k} - 2.T_{i,k} + \left(1+\frac{1}{2.i}\right).T_{i+1,k}\right) + \frac{1}{(\Delta z)^2}.\left(T_{i,k-1} - 2.T_{i,k} + T_{i,k+1}\right) + \frac{1}{K}.h_{i,k} = 0$$

pour des valeurs de r non nulles.

Pour $r = 0$, on a :

$$\lim_{r\to 0}\left(\frac{1}{r}.\frac{\partial T}{\partial r}\right) = \frac{\partial^2 T}{\partial r^2}$$

et l'équation de conduction en $r = 0$ devient :

$$2.\frac{\partial^2 T}{\partial r^2} + \frac{\partial^2 T}{\partial z^2} + \frac{1}{K}.h(r,z) = 0 \quad \text{en } r = 0$$

Soit en utilisant les différences finies au nœud $(1, k)$:

$$4.\frac{T_{2,k} - T_{1,k}}{(\Delta r)^2} + \frac{T_{1,k-1} - 2.T_{1,k} + T_{1,k+1}}{(\Delta z)^2} + \frac{1}{K}.g_{1,k} = 0$$

8. Discrétisation de l'équation de la Conduction en régime instationnaire

Dans le système de coordonnées cartésiennes (trois dimensions), l'équation de conduction de la chaleur en régime instionnaire (temporel) s'écrit (si $K = Cste$) :

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} + \frac{1}{k}.h(x,y,z) = \frac{\rho.C_p}{K}.\left(\frac{\partial T}{\partial t}\right)$$

Pour résoudre cette équation aux dérivées partielles en utilisant la méthode des différences finies, on utilise un maillage cubique (mailles de côtés $(\Delta x, \Delta y, \Delta z))$ avec :

$$\begin{cases} x = i.\Delta x \\ y = j.\Delta y \\ z = k.\Delta z \end{cases}$$

où i, j et k sont entiers positifs, et le domaine temporel est divisé en petits intervalles de temps Δt de telle sorte que : $t = n.\Delta t$

Dans ce cas, la température $T(x, y, z, t)$ au point *P(x,y,z)* à un temps donné t est représentée par :

$$T(x, y, z, t) = T(i.\Delta x, j.\Delta y, k.\Delta z, n.\Delta t) = T^n_{i,j,k}$$

En utilisant la méthode des différences finies, les différents termes de l'équation ci-dessus au dérivées partielles s'écrivent au point *P(x,y,z)* :

$$\begin{cases} \left.\frac{\partial T}{\partial t}\right|_P = \left.\frac{\partial T}{\partial t}\right|_{i,j,k} = \frac{T^{n+1}_{i,j,k} - T^n_{i,j,k}}{\Delta t} + O(\Delta t)^2 \\ \left.\frac{\partial^2 T}{\partial x^2}\right|_P = \left.\frac{\partial^2 T}{\partial x^2}\right|_{i,j,k} = \frac{T^n_{i-1,j,k} - 2.T^n_{i,j,k} + T^n_{i+1,j,k}}{(\Delta x)^2} + O(\Delta x)^2 \\ \left.\frac{\partial^2 T}{\partial y^2}\right|_P = \left.\frac{\partial^2 T}{\partial y^2}\right|_{i,j,k} = \frac{T^n_{i,j-1,k} - 2.T^n_{i,j,k} + T^n_{i,j+1,k}}{(\Delta y)^2} + O(\Delta y)^2 \\ \left.\frac{\partial^2 T}{\partial z^2}\right|_P = \left.\frac{\partial^2 T}{\partial z^2}\right|_{i,j,k} = \frac{T^n_{i,j,k-1} - 2.T^n_{i,j,k} + T^n_{i,j,k+1}}{(\Delta z)^2} + O(\Delta z)^2 \end{cases}$$

et l'équation de la conduction discrétisée en trois dimensions devient :

$$\frac{\rho.C_p}{}.\left(\frac{T^{n+1}_{i,j,k} - T^n_{i,j,k}}{\Delta t}\right) = \frac{T^n_{i-1,j,k} - 2T^n_{i,j,k} + T^n_{i+1,j,k}}{(\Delta x)^2} + \frac{T^n_{i,j-1,k} - 2T^n_{i,j,k} + T^n_{i,j+1,k}}{(\Delta y)^2}$$

$$+ \frac{T^n_{i,j,k-1} - 2.T^n_{i,j,k} + T^n_{i,j,k+1}}{(\Delta z)^2} + \frac{1}{K}.h_{i,j,k}$$

En regroupant les différents termes de cette équation, on obtient :

$$T_{i,j,k}^{n+1} = \left(1 - \frac{2.K.\Delta t}{\rho.C_p}.\left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta x)^2} + \frac{1}{(\Delta x)^2}\right)\right).T_{i,j,k}^{n}$$
$$+ \frac{\Delta t.K}{\rho.C_p}.\left(\frac{T_{i-1,j,k}^{n} + T_{i+1,j,k}^{n}}{(\Delta x)^2} + \frac{T_{i,j-1,k}^{n} + T_{i,j+1,k}^{n}}{(\Delta y)^2} + \frac{T_{i,j,k-1}^{n} + T_{i,j,k+1}^{n}}{(\Delta z)^2}\right) + \frac{1}{K}.h_{i,j,k}$$

Cette équation ne peut être résolue que par une méthode itérative (méthode de *Gauss-Seidel* par exemple).

Si on pose :

$$M = 1 - \frac{2.K.\Delta t}{\rho.C_p}.\left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta x)^2} + \frac{1}{(\Delta x)^2}\right)$$

alors, la condition de convergence de la solution recherchée dépend essentiellement du signe de la quantité M. Cette quantité doit être strictement positive.

Ainsi, une condition doit relier le pas de temps Δt et les autres pas spatiaux (Δx, Δy et Δz) :

$$\Delta t < \frac{\rho.C_p}{2.K}.\left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta x)^2} + \frac{1}{(\Delta x)^2}\right)^{-1}$$

Pour amorcer le calcul itératif, on ensemence le domaine de frontière $\Re$ par des valeurs arbitraires ($T_{i,j,k}^{(0)} = T_0$ par exemple), et on arrête les calculs quand la condition suivante sera réalisée :

$$\left|T_{i,j,k}^{(r+1)} - T_{i,j,k}^{(r)}\right| \leq \varepsilon$$

où ε est la précision que l'on se fixera.