# Mécanique classique : Annexe numérique et informatique

A. Colin de Verdière et S. Pogossian

## 1. Méthodes numériques

La technique numérique est introduite au travers du système dynamique sur la droite :

$$\frac{dx}{dt} = f(x) \qquad \text{A1-1}$$

La fonction f(x) étant connue, il faut trouver x connaissant sa position initiale $x_0$ à $t = t_0$. La formulation analytique de cette solution est :

$$x(t) = x_0 + \int_{t_0}^{t} f[x(t')]\,dt' \qquad \text{A1-2}$$

L'idée de base d'une formulation numérique de A1-1 est de repartir de la définition de la dérivée :

$$\frac{dx}{dt} = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} \qquad \text{A1-3}$$

Mais plutôt que de prendre la limite lorsque $\Delta t$ tend vers 0, on choisit un petit intervalle de temps fini $\Delta t$. Si la position $x_0$ est connue à l'instant $t_0$, la nouvelle position à $t + \Delta t$ sera alors simplement :

$$x(t + \Delta t) = x_1 = x_0 + f(x_0)\Delta t$$

Connaissant $x_1$, on peut recommencer pour trouver successivement $x_2$, $x_3$,… $x_n$ aux temps $t_0 + 2\Delta t$, $t_0 + 3\Delta t$,… , $t_0 + n\Delta t$, l'itération générique pour passer d'une position à la suivante étant :

$$x_{n+1} = x_n + f(x_n)\Delta t \qquad \text{A1-4}$$

On appelle ceci un schéma Euler avant, la méthode la plus simple pour trouver la fonction $x(t)$ aux instants discrets $t_0 + n\Delta t$. L'étape suivante est de programmer le calcul des $x_n$ pour n grand. Avec les langages de type *matlab* ou les versions libres comme *octave* ou *scilab* les lignes de codes suivantes permettent de calculer tous les $x_n$ :

1/ On spécifie les conditions initiales à $t = t_0$ :

$$x_n = x_0$$

2/ On choisit le pas de temps $\Delta t$ et le nombre n de pas de temps :

$$\Delta t = 0{,}01$$

$$n = 100$$

3/ On utilise la boucle récursive suivante qui exécute les lignes d'instructions comprises entre le **for** et le **end** pour l'indice k allant de 1 à n :

**for** $k = 1 : n$

$$x_k \;=\; x_k \;+\; \Delta t\, f(x_k)$$

% la ligne ci-dessus stocke la nouvelle valeur $x_{k+1}$ dans la même case mémoire que $x_k$. Cette case mémoire $x_k$ voit donc défiler toutes les valeurs $x_1$, $x_2$,… $x_n$ et la ligne suivante stocke la solution pour pouvoir la visualiser ultérieurement (hors de la boucle).

% stockage de la solution

$x\_graph(k)=x_k$

**end**

Une simple boucle permet donc de calculer l'intégrale A1-2 mais il reste à choisir l'intervalle $\Delta t$. Si $\Delta t$ est choisi très petit, la précision doit être très bonne mais pour obtenir la solution au temps T fixé, le nombre n de pas de temps augmente sérieusement puisque $n = T/\Delta t$. Il faut donc s'intéresser à la précision du schéma numérique qui doit dépendre de $\Delta t$.

Pour en avoir une idée, intégrons A1-2 sur un pas de temps :

$$x(t + \Delta t) - x(t) = \int_t^{t+\Delta t} f[x(t')]\,dt' \qquad \text{A1-5}$$

La relation A1-5 est exacte. Traçons la fonction à intégrer entre ces deux pas de temps :

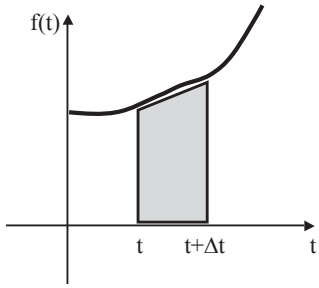


Figure A1-1

On voit que pour passer d'un pas de temps au suivant, A1-5 demande de calculer l'aire sous la courbe de la fonction f(t), ce que le schéma Euler avant fait en l'approximant par l'aire du rectangle $f(t)\Delta t$. La figure A1-1 montre tout de suite qu'il serait préférable que la fonction f soit centrée au milieu de l'intervalle. Pour juger de l'erreur du schéma numérique Euler avant on procède en comparant la vraie valeur $x(t + \Delta t)$ avec l'approximation $x_{n+1} = x_n + f(x_n)\Delta t$. Le développement limité de Taylor permet d'écrire la vraie valeur comme :

$$x(t+\Delta t) = x_n + \frac{\Delta t}{1!} f(t_n) + \frac{\Delta t^2}{2!} f'(t_n) + \ldots + \frac{\Delta t^n}{n!} f^{(n)}(t_n)$$

L'erreur est donc :

$$E = \left|x(t+\Delta t) - x_n\right| = \frac{\Delta t^2}{2!}\left|f'(t_n)\right| + O(\Delta t^3)$$

Quand $\Delta t$ est petit, l'erreur locale à l'instant $t + \Delta t$ est donc $O(\Delta t^2)$ si la dérivée $f'(t_n)$ n'est pas nulle. Maintenant l'erreur cumulée au bout de n pas de temps est $O(\Delta t)$. En effet la somme des erreurs après un temps $T = n\Delta t$ est $n \times O(\Delta t^2) = \frac{T}{\Delta t} O(\Delta t^2) = O(\Delta t)$. Si on divise le pas de temps par deux, l'erreur est divisée par deux mais l'effort de calcul pour atteindre un temps T donné double.

Pour autant le schéma Euler avant n'est pas le meilleur, loin s'en faut. De meilleurs schémas tentent de centrer f(t) lors de l'évaluation de l'intégrale A1-5. La difficulté est que l'on ne connaît pas $f(t + \Delta t)$ évaluée à la position $x(t + \Delta t)$. Une méthode est d'approcher l'aire exacte par celle du trapèze grisé sur la figure A1-1 en prenant comme limite à $t + \Delta t$ la prédiction faite par le schéma Euler avant. Sur cette base le schéma Euler amélioré consiste en deux étapes :

I. La prédiction $x_p$ avec le schéma Euler avant :

$$x_p = x_n + f(x_n)\Delta t$$

II. La correction faite en évaluant l'aire du trapèze sur la figure A1-1 par $\frac{1}{2}\left[f(x_n) + f(x_p)\right]\Delta t$.

L'ensemble du schéma est récrit ci-dessous :

$$\begin{aligned} x_p &= x_n + f(x_n)\Delta t \\ x_{n+1} &= x_n + \frac{1}{2}\left[f(x_n) + f(x_p)\right]\Delta t \end{aligned} \qquad \text{A1-6}$$

Une variante est le schéma Matsuno ou encore Euler arrière qui évalue simplement l'aire sur la figure A1-1 par $f(x_p)\Delta t$ dans l'étape de correction :

$$\begin{aligned} x_p &= x_n + f(x_n)\Delta t \\ x_{n+1} &= x_n + f(x_p)\Delta t \end{aligned} \qquad \text{A1-7}$$

Il y a deux étapes à chaque pas de temps, le prédicteur par le schéma Euler avant, et le correcteur qui vient corriger la première estimation en se servant de la valeur intermédiaire $x_p$ pour estimer $f(x)$. Les codes A1-6 ou A1-7 ne sont guère plus compliqués que A1-4 mais requièrent le double d'opérations par pas de temps. On peut montrer que l'erreur cumulée après n pas de temps est $O(\Delta t^2)$ pour le schéma Euler amélioré mais reste $O(\Delta t)$ pour le schéma Matsuno. Celui-ci possède néanmoins de meilleures propriétés de stabilité pour les régimes d'oscillations et c'est la raison pour laquelle il est utilisé dans les programmes sysdyn1.m, sysdyn2.m, sysdyn3.m.

Une méthode encore plus précise mais qui demande encore plus de calcul est la méthode de Runge-Kutta d'ordre 4 :

$$\begin{aligned} k_1 &= f(x_n)\Delta t \\ k_2 &= f(x_n + \frac{1}{2}k_1)\Delta t \\ k_3 &= f(x_n + \frac{1}{2}k_2)\Delta t \\ k_4 &= f(x_n + k_3)\Delta t \\ x_{n+1} &= x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t \end{aligned} \qquad \text{A1-8}$$

L'erreur cumulée au bout de n pas de temps de ce schéma est beaucoup plus petite $O(\Delta t^4)$. Comme les vitesses des calculs par ordinateur augmentent sans cesse, on peut se demander pourquoi il faudrait utiliser ces schémas précis sachant que l'on peut toujours contrôler l'erreur en diminuant le pas de temps de schémas moins précis. Bref comme le gain d'un schéma complexe comme A1-8 n'est parfois pas si évident, il ne faut les essayer que lorsque les schémas plus simples ne fonctionnent pas bien.

Quel que soit le schéma choisi, le choix du pas de temps $\Delta t$ reste affaire d'expérimentation. Pour être précis il faut qu'il soit petit mais alors le temps de calcul augmente d'autant. Si le pas de temps est trop grand, la précision chute, les schémas deviennent instables et la valeur de $x_n$ explose. Mais en général on connaît l'échelle de temps $\tau$ sur laquelle évolue le phénomène que l'on cherche à modéliser et on devra s'assurer que $\Delta t << \tau$. Avec un $\Delta t$ très petit, il faut cependant être conscient de l'accumulation des erreurs d'arrondis faites par un ordinateur. En simple précision un ordinateur effectue les calculs avec une erreur d'arrondi de $10^{-7}$ qui chute à $10^{-14}$ en double précision que l'on choisira donc quand un très grand nombre d'itérations s'avère nécessaire.

Le choix du pas de temps est illustré sur le système dynamique de l'oscillateur harmonique avec le schéma Matsuno (voir sysdyn2.m). Le système à résoudre est :

$$\begin{aligned} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -x \end{aligned} \qquad \text{A1-9}$$

avec x la position et y la vitesse. Le schéma numérique Matsuno correspondant est :

$$\begin{aligned} \text{I} &\begin{cases} x_p = x_n + \Delta t\, v_n \\ v_p = v_n - \Delta t\, x_n \end{cases} \\ \text{II} &\begin{cases} x_{n+1} = x_n + \Delta t\, v_p \\ v_{n+1} = v_n - \Delta t\, x_p \end{cases} \end{aligned} \qquad \text{A1-10}$$

Avec les conditions initiales x(0) = 1 et y(0) = 0, la solution exacte de A1-9 est x = cos(t). On compare ci-dessous la solution numérique $x_n$ avec la solution exacte sur une période (= 2π) obtenue via A1-10 pour trois valeurs de Δt.

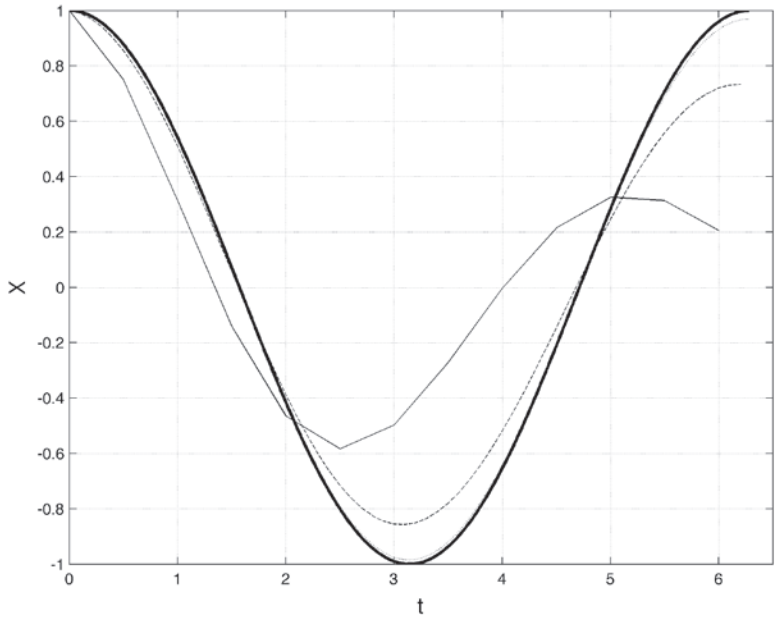


Figure A1-2 *La solution numérique x(t) (traits fins) est comparée avec la solution exacte (trait épais) pour trois valeurs de Δt (= 0,5, 0,1 et 0,01).*

Si les deux premières solutions obtenues pour les valeurs de Δt = 0,5 et 0,1 laissent clairement à désirer, on obtient une excellente précision pour Δt = 0,01, la solution numérique étant difficilement séparable de la solution exacte sur la figure A1-2.

# 2. Le codage informatique

On donne dans ces quelques pages le minimum vital pour démarrer avec le langage scientifique *matlab*. Les langages gratuits type *octave* ou *scilab* sont très proches de *matlab*.

Lorsqu'on lance *matlab*, deux fenêtres importantes s'ouvrent, une première fenêtre de *commandes (ou d'instructions)* qui fonctionne comme une calculatrice de bureau : on écrit une opération que l'on exécute en appuyant sur la touche return (ou enter) du clavier, le résultat s'affichant alors sous la forme ans = (ans pour *answer*). Une deuxième *fenêtre d'édition* permet d'éditer une suite de commandes comme le ferait un éditeur de texte. Cette suite de commandes constitue un programme (*script*) qui est stocké dans un fichier qui se termine par *.m*, soit *nomfichier.m* qui est placé par défaut (via un *save as*) dans le *directory Documents/Matlab*. Lorsque l'on tape *nomfichier* dans la première fenêtre de commande, les instructions du programme s'exécutent. Une instruction *pause* rencontrée dans le programme en suspend l'exécution ce qui permet de voir les premiers résultats. Le programme est relancé en appuyant sur la touche *return* du clavier. Pour connaître la signification et l'usage d'une commande *matlab*, il suffit de taper *help*, nom de la commande. Dans ce qui suit, les instructions *matlab* sont mises en *italique* et les explications en caractère droit. Pour travailler sur un autre programme sans fermer la session *matlab* en cours, il faut effacer toutes les variables et toutes les figures précédentes avec les deux commandes :

*clear all* : supprime toutes les variables

*close all* : ferme toutes les fenêtres graphiques

Pour ne pas l'oublier, on peut écrire ces deux instructions en tête du programme *nomfichier.m.*

## *1. Opérations*

Dans la fenêtre de commande vérifiez les opérations suivantes :

\+ – * (multiplication) / (division) ^ (exposant)

2*2

ans = 4 (pour answer)

La variable nommée ans contient maintenant la valeur 4.

ans^2

ans=16

Dans une suite d'opérations, la division est faite en premier, puis la multiplication, puis les additions soustractions :

1+1/2*4

ans=3

Mais il est préférable d'utiliser des parenthèses pour être sûr de ce que l'on fait, les parenthèses les plus intérieures étant exécutées en premier.

1 + (1/2)*4 donne le même résultat avec moins d'ambiguïté. Notez que le caractère espace (blanc) dans une expression numérique est permis pour donner plus de lisibilité.

Si on tape :

1/0

ans=Inf

Inf est la notation pour l'infini.

Si on tape :

0/0

ans=NaN

NaN signifie *Not a Number*, une opération indéterminée. On verra plus loin son usage très utile en traitement de données.

## 2. Nombre réel, nombre complexe et nombre texte

i) Un nombre réel est codé en simple précision sur 32 bits (4 bytes). Mais *matlab* le code par défaut en double précision, soit 64 bits (8 bytes). Quand on définit :

X = 10

L'instruction *whos* donnera l'information de la précision sur X qui est *'double'*.

L'instruction *eps('double')* montre que *matlab* distingue deux nombres séparés d'au moins $2.2 \times 10^{-16}$. La précision chute à $1.2 \times 10^{-7}$ en simple précision, tapez *eps('single')*.

L'exposant 10 de la notation scientifique s'écrit e en *matlab* :

Si on tape dans la fenêtre de commande :

3.7*10^(-7)

ans= 3.7000e–07

Il est toujours préférable d'utiliser la notation e dans son code d'instructions puisque c'est beaucoup plus simple.

Par défaut *matlab* écrit un nombre avec 5 chiffres significatifs (quatre après le point décimal), c'est le *format short* par défaut. S'il est nécessaire d'avoir plus de chiffres significatifs, taper *format long*. Les nombres seront imprimés avec 16 chiffres significatifs (15 après le point décimal).

ii) **Nombre complexe**

L'algèbre complexe est complètement intégré dans *matlab* de sorte que les 4 opérations précédentes sur les nombres réels restent les mêmes si les nombres sont complexes. Pour déclarer un nombre complexe, on écrit simplement :

2 + 3i

ans = 2.0000 + 3.0000i

Les commandes *real* et *imag* sont utilisées pour extraire la partie réelle ou imaginaire du nombre complexe précédent :

*real(ans)*

ans=2

On peut aussi déclarer un nombre complexe avec *complex* :

*complex(2 ,3)*

ans= 2.0000 + 3.0000i

et pour trouver le module :

*abs(ans)*

ans=3.6056

Autres commandes utiles *conj* pour le conjugué et *angle* pour la notation trigonométrique :

*z=2+3i*

*abs(z)*exp(i*angle(z))* redonne précisément z

iii) **Nombres 'texte'**

Quand on a besoin de mettre du texte sur une figure ou sur un résultat, on utilise la classe caractère *char*. Pour déclarer un texte en classe *char* on le met entre simples *quotes* comme *'texte'*.

*'bonjour'*

ans = 'bonjour'

Ce mot contient 7 caractères et il est codé comme un vecteur de 7 éléments. La commande *whos* permet de voir que chaque caractère est codé sur 2 bytes (1 byte = 8 bits) dans le format Unicode (ASCII).

Pour afficher un nombre dans un titre de figure, la commande *num2str* permet de transformer ce nombre en un nombre texte dans la classe *char :*

*A=2.37*

nombre = *num2str(A)*

nombre='2.37'

Pour convertir un nombre entier en *char*, utiliser la commande *int2str*

Depuis 2017, on peut aussi utiliser les doubles quotes *"texte"* à la différence près que le texte n'est plus codé comme un vecteur mais est codé comme un seul *scalaire* :

*"bonjour"*

ans = "bonjour"

La commande *whos* permet de voir que le mot est codé comme un scalaire de la classe *string* avec 132 bytes en tout.

La commande *string* permet de transformer un nombre dans la classe *string* :

*A=2.37*

*nombre = string(A)*

nombre = "2.37"

## 3. Caractères spéciaux

% démarre une ligne de commentaires dans un programme

: voir plus loin dans vecteurs

; supprime l'impression dans la fenêtre d'exécution du résultat de l'opération qui se trouve placé avant. Une ligne d'un programme (un fichier .m) se termine donc généralement par ;

; ce même caractère dans une matrice permet de passer d'une ligne à une autre.

, permet de mettre deux commandes à la suite sur une même ligne de programme.

… Si une commande *matlab* ne tient pas sur une ligne, on finit la ligne par un espace et trois … et on continue sur la ligne du dessous.

bla = blablabla …

suite de la ligne précédente

## 4. Variables

Une variable est un objet avec un *nom* placé à gauche du signe = et auquel on donne une *valeur* placée à droite du signe =. La *valeur* est contenue dans la case mémoire appelée *nom*. Le nom peut contenir des majuscules, des minuscules, des chiffres ou encore le caractère *underscore* _ qui permet de clarifier les noms. Il ne faut pas démarrer le nom par un chiffre. Les variables B et b sont deux variables différentes et il ne faut pas mettre d'espaces dans le nom. Si on écrit dans la fenêtre de commandes :

*a = 2*

*matlab* répond en écrivant a = 2. Pour éviter cet écho, mettre un point virgule à la fin :

*a=2 ;*

soit aussi

*b=3 ;*

On peut alors commencer à faire du calcul symbolique et écrire :

*c = b^2+a*b*

*matlab* répond :

*c=15*

Pour savoir quelles variables sont définies dans une session, taper :

*who*

et le nom des variables utilisées apparaît. Si on tape :

*whos*

on aura encore plus d'informations avec la taille (size) des variables et leur type (class).

Si on tape :

*clear a*

la variable a est effacée.

Si on tape :

*clear all*

toutes les variables sont effacées et tout se passe comme si on redémarrait à neuf une session *matlab*.

## 5. Vecteurs

*Matlab* est un langage puissant car il permet de travailler directement sur des vecteurs.

Un vecteur est défini en mécanique par ses composantes, deux dans le plan, trois dans l'espace. Mais on généralise ici la notion à toute suite de nombres consécutifs ou pas. Une suite de n nombres est un vecteur de dimension n.

Le caractère : joue un grand rôle car il permet de générer une suite de valeurs :

1 : 6 *exec*

ans = 1 2 3 4 5 6

donc ici ans est une suite de 6 entiers consécutifs. On peut aussi rentrer individuellement les composantes d'un vecteur avec les crochets [ ] et écrire :

a = [ 1  2.2  3.5  5] ;

La syntaxe x :d :y permet de générer une suite de nombres qui va à peu près de x à y avec l'incrément d. Taper 0 : 0.5 : pi pour voir ce qui se passe. Si d est négatif, la

suite sera décroissante. Pour générer une suite de 10 nombres régulièrement espacés de 0 à 2pi, utiliser *linspace* qui calcule l'incrément pour aller exactement de 0 à 2pi :

*t=linspace(0, 2*pi, 10)*

*matlab* répond avec les valeurs de t sur une ligne. On dit que est un vecteur ligne *(row)*.

Et maintenant pour calculer les 10 valeurs correspondantes du sinus de t, il suffit d'écrire :

*y=sin(t)*

*Matlab* calcule toutes les valeurs de y d'un seul coup, la vertu du langage vectoriel. Le vecteur y est composé des 10 valeurs de sin(t). De chaque côté du signe = on a donc toujours une entité avec le même nombre d'éléments.

Pour connaître le nombre d'éléments de t, taper :

*length(t)*

ans=10

Maintenant taper :

*t'*

*matlab* répond avec les valeurs de t disposées en colonne (*column*). Le symbole ' est le symbole pour calculer le *transposé* du vecteur. Pour savoir si on a affaire à un vecteur ligne ou à un vecteur colonne, taper :

*size(t)*

ans= 1  10  soit 1 ligne et 10 colonnes.

*size(t')*

ans = 10  1  soit 10 lignes et 1 colonne.

## *6. Indices*

Les indices permettent d'avoir accès à l'un ou à plusieurs éléments du vecteur t.

Taper :

*t(3)*

ans=1.3963

Mais souplesse exceptionnelle, l'indice peut être lui-même un vecteur :

*t(1 : 3)*

ans= 0  0.6981  1.3963

Une variable peut donc être un indice. Taper

*k = 1 : 3*

*t(k)*

redonne le même résultat.

Pour aller au dernier élément d'un vecteur dont on ne connaît pas la taille, on utilise *end*.

*t(end)*

ans = 6.2832

ou encore :

*t(end–2 :end)*

ans= 4.8869  5.5851  6.2832

## 7. Opérations sur les vecteurs

On additionne ou on soustrait des vecteurs de même dimension. Soit deux vecteurs de même dimension, x et y, deux vecteurs lignes de dimension n. La somme s'écrit simplement s = x + y et la différence d = x – y.

i) En traitement de données on cherche parfois juste à multiplier x et y élément par élément. Pour le faire, on écrit :

z = x.*y

qui donne le vecteur z de même dimension construit sur les produits des composantes

[x(1)*y(1) x(2)*y(2) … x(n)*y(n)]

Ne pas oublier le point . avant le *. De la même manière on peut diviser élément à élément en faisant x./y. On peut aussi élever chaque élément de x à une certaine puissance, ici au carré :

x.^2 = [x(1)^2  x(2)^2  …  x(n)^2]

À nouveau ne pas oublier le . avant le signe exposant ^. Et si on fait la somme de tout ça on obtient le module du vecteur x au carré avec *sum*(x.^2)

ii) Le produit scalaire est central pour les vecteurs et s'écrit :

a=x*y'

avec y', le transposé de y qui est donc un vecteur colonne. Il n'y a plus de point devant le signe *. Le produit scalaire s'effectue comme la multiplication de la matrice ligne de dimension (1, n) par la matrice colonne de dimension (n, 1). Ainsi le module b du vecteur x est : b = sqrt(x*x').

## 8. Instructions logiques

Ces opérateurs interviennent lorsqu'on a besoin de faire des tests conditionnels soit avec la commande *if end* soit avec la commande *find*

i) opérateurs de comparaison

= = égal à

~ = n'est pas égal à

< moins que

> plus que

<= moins ou égal à

>= plus ou égal à

ii) opérateurs logiques

| | |
|---|---|
| & | et |
| \| | ou |
| ~ | non |
| xor | ou exclusif |
| any | vrai si n'importe quel élément n'est pas nul |
| all | vrai si tous les éléments sont non nuls |

Le résultat d'une opération de comparaison entre deux nombres est 0 (false) ou 1 (true) codé sur un byte de la classe *logical*.

## 9. NaN

La notation *NaN* est très utile en traitement des données pour indiquer qu'une donnée est manquante dans une suite de valeurs. On note les températures sur la semaine du dimanche au lundi mais on a oublié le jeudi. On forme quand même le vecteur theta de 7 composantes ainsi :

theta=[10.5 9.2 8.3 12.0 NaN 10.1 9.6 ]

Sur un graphique, la valeur NaN du jeudi ne sera pas tracée. Dans une opération arithmétique la présence d'un NaN donnera un résultat qui sera aussi un NaN. C'est utile de déceler ces valeurs manquantes marquées par un NaN dans un traitement de données pour ne pas calculer ce qui serait de toutes façons au final un NaN. Pour savoir si un nombre est NaN, on utilise la commande logique *isnan*. Avec :

theta=[10.5 9.2 8.3 12.0 NaN 10.1 9.6 ] ;

*isnan(theta(1))*

ans=0 donc faux

*isnan(theta(5))*

ans=1 donc vrai

## *10. Matrices*

*Matlab* a été développé par l'équipe de Cleve Moler qui avait développé les routines Fortran d'algèbre linéaire LINPACK, EISPACK, etc. et ils ont créé *matlab* pour en simplifier l'utilisation. Pour entrer une matrice en *matlab*, on commence par [ et on finit par ] et on passe d'une ligne à une autre avec le caractère point virgule ;

*a = [ 1 2 3 ;4 5 6 ;7 8 9]*

a=

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

On peut générer d'autres matrices à partir de la sous-matrice a. Pour le voir, tapez :

b=[a 5*a] ou encore c=[a ; 5*a]

On peut générer une matrice de m lignes et n colonnes remplie de zéros par la commande :

*a=zeros(m,n)*

ou de un :

*a=ones(m,n)*

ou de NaN

*a=NaN(m,n)*

une dernière expression qui permet de prédéfinir la taille d'un tableau sans préjuger de ce que l'on mettra dedans.

*eye* est la matrice identité avec des 1 sur la diagonale :

*a=eye(3)* est la matrice identité (3, 3)

Les indices permettent d'accéder aux éléments individuels d'une matrice. Taper

*a(2 , 3) exec*

ans = 6

Pour extraire une ligne ou une colonne de a, utilisez le caractère :

a( : , 2) *matlab* extrait la deuxième colonne de a

a(3 , :) *matlab* extrait la troisième ligne de a

Pour supprimez une ligne ou une colonne, assigner-la à la matrice vide (*empty*) [ ]

a( : , 2) = [ ] *matlab* efface la deuxième colonne de a

L'utilisation de *end* est similaire pour les matrices et les vecteurs. Tapez

*a(end–2 , end)*

ans = 3

L'utilisation phare des matrices reste la résolution des systèmes linéaires. Quand A est une matrice carrée possédant un inverse, la solution du système linéaire :

$$A\,x=b$$

s'écrit simplement en *matlab* :

$$x = A \setminus b$$

avec le *backslash* orienté vers la gauche.

La multiplication de deux matrices A et B se calcule simplement comme :

$$C=A*B$$

le nombre de colonnes de A devant être égal au nombre de lignes de B, soit A(m,n) et B(n,p) résultant en la matrice C(m,p).

## *11. Instructions tests et récursives*

L'utilité d'un programme est de pouvoir répéter des opérations un très grand nombre de fois, ce qui permet de calculer des sommes de chiffres (donc des intégrales), de trouver les solutions de problèmes de mécanique et bien d'autres choses.

L'instruction majeure est le *for … end* qui est la boucle *DO … ENDDO* du Fortran, l'ancêtre des langages scientifiques encore utilisé dans les grosses applications de calcul pour ses capacités de *number crunching*.

L'instruction s'écrit :

*for index = début : incrément : fin*

instructions à exécuter

*end*

Les instructions sont à exécuter autant de fois que le demande l'index lorsqu'il va de la valeur *début* à la valeur *fin* avec l'*incrément* indiqué.

Exemple : calculer la somme des 100 premiers nombres impairs. Il existe plusieurs façons de faire ce calcul. En voici une qui est maintenant un programme ou script que l'on écrit dans la fenêtre d'édition. On appelle ce script *impair.m*

```
a=1 ;            % on donne la valeur du premier nombre impair
sum_a = a ;      % on initialise la variable somme que l'on cherche
    for i=2 :100     % on exécute les deux lignes suivantes pour i allant de 2 à 100.
        a = a+2 ;            % valeur du nombre impair suivant
        sum_a=sum_a+a ;      % nouvelle valeur de la somme
    end
```

*disp*('somme des 100 premiers nombres impairs') % la commande *disp* pour *display* écrit cette ligne.

[sum_a] % écrit la valeur calculée

Les deux lignes du programme entre le *for* et le *end* ne sont pas des égalités arithmétiques mais s'interprètent ainsi. Lorsque l'on définit la variable a, une case mémoire est réservée avec l'adresse a pour la retrouver :

a = a+2 veut dire que a + 2 est calculée à droite du signe = avec la valeur de a courante (dans la case mémoire qui s'appelle a). Ensuite on remet cette nouvelle valeur à gauche du signe = dans la même case mémoire dont le contenu vient d'être modifié. Cette nouvelle valeur de a est ensuite ajoutée dans sum_a de la même façon.

Maintenant pour exécuter *impair.m* dans la fenêtre de commande, on tape :

*impair*

sum_a =10 000 est la réponse demandée

*Matlab* permet de calculer cette somme plus simplement car deux instructions sont déjà pré-codées :

a=1 : 2 : 200 ; % génère le vecteur a qui contient les 100 premiers nombres impairs

sum(a) % une fonction matlab qui fait directement la somme des éléments de a.

La commande *if … else … elseif … end* exécute des instructions selon qu'un test logique est satisfait ou pas.

Exemple : on connaît une fonction y = f(x) dont on veut trouver le maximum en y et la position x de ce maximum.

Dans la fenêtre édition on crée le code *arthur.m*

x = 0 :0.1 :pi ; % on définit le vecteur x

y = sin(x) ; % la fonction y de même dimension que x

n = *length*(x) ; % n est la longueur du vecteur

x_max = x(1) ; % initialisation de la position du maximum

y_max = y(1) ; % initialisation du maximum de y

```
        for i=2 :n
                if y(i)  > y_max ; % exécute les deux instructions suivantes si l'inégalité est vraie
                        y_max = y(i) ;
                        x_max=x(i) ;
                end
        end
disp('maximum de y pour x =')
[y_max x_max]
```

Dans la fenêtre de commande, on tape :

*arthur*

ans=[0.9996  1.6000]

Ce qui donne la réponse à la question. Pour augmenter la précision, il faut augmenter la résolution spatiale de la fonction, par exemple prendre x = 0 : 0.001 : pi avec un incrément de $10^{-3}$.

Mais ici aussi *matlab* a aussi une solution directe :

```
[y_max k] =  max(y) ;
x_max = x(k)
```

*max* fournit le maximum y_max du vecteur y et l'indice k du vecteur où l'on trouve ce maximum.

Pour conclure, la boucle *for… end* et le test *if … end* sont les deux instructions centrales en programmation.

## 12. Graphiques

On ouvre une première figure par :

*figure(1)*

i) dessin à une dimension

Pour tracer la courbe y = sin(t) avec les valeurs de t précédemment définies :

*plot(t,y)* trace la courbe

*xlabel('time')* met le label time sur l'axe des x

*ylabel('Y')* met le label Y sur l'axe des y

*title(*'titre'*)* met un titre en haut du graphique

Supposons que l'on veuille mettre la valeur du maximum y_max dans le titre. On écrira :

*title*(['titre et maximum = ', *num2str*(y_max)])

Les crochets [ ] forment ici un vecteur de variables *char.*

Tapez *help plot* pour découvrir toutes les possibilités de tracé, points seuls, noir ou couleur, lignes seules noire ou couleur, continue ou pointillée.

L'instruction *hold on* permet d'ajouter une deuxième instruction *plot* sans détruire le premier tracé. Si on veut dessiner les points et les lignes, écrire les trois commandes :

*plot(*t,y,'kx'*)* points x en noir

*hold on*

*plot(*t,y,'r--'*)* courbe rouge pointillée

On pourra aussi rajouter une deuxième courbe sur le même dessin, etc.

Si on a beaucoup de dessins à faire c'est intéressant d'utiliser la fonction *subplot* qui découpe la fenêtre de la figure en m x n figures indépendantes :

*subplot*(m,n,p) crée m dessins dans la direction verticale et n dessins dans la direction horizontale et permet de travailler sur le dessin de numéro p.

ii) dessin à deux dimensions

Pour un graphique en deux dimensions, de nombreuses possibilités existent. Citons juste une des plus communes, la fonction *contour* :

*contour(x,y,z)* trace les contours de la surface z aux points de coordonnées (x,y). Si x a une dimension m et y une dimension n, il est plus commode de calculer z comme z(n,m). Les valeurs de x et y doivent être monotones pour que la surface z soit définie uniquement.

Quand on utilise *contourf*, les régions entre contours sont colorées. On peut choisir les couleurs avec *colormap* et pour connaître le lien entre les valeurs de z et les couleurs il faut inclure la commande *colorbar.* Tapez *help* pour voir toutes les options.

Pour supprimer le plot de la figure(1) faire *clf*

Pour être sûr de partir sur du neuf, on peut taper avant de commencer :

*figure(1),clf*

## 13. Quelques fonctions matlab utiles

| | |
|---|---|
| max | trouve le maximum d'un vecteur |
| min | trouve le minimum d'un vecteur |
| find | voir exemple |
| sum | somme des composantes du vecteur |
| cumsum | somme cumulée |
| mean | moyenne |
| std | écart type |
| sort | ordonne les composantes selon les valeurs croissantes |
| trapz | méthode des trapèzes pour le calcul d'une intégrale |
| polyfit | méthode des moindres carrés |
| abs | valeur absolue et module d'un nombre complexe |
| sqrt | racine carrée |
| exp | exponentielle |

La plupart de ces fonctions ne demandent pas de commentaires particuliers et il suffit de taper *help* avec le nom pour trouver toutes les options d'usage. La fonction *find* est particulièrement utile pour avoir accès très facilement à des informations sur une série de nombres (ou un tableau).

Exemple : on génère la suite de Fibonacci en partant de a(1)=0 et a(2)=1 sachant que le nombre suivant est juste la somme des deux précédents et on veut trouver les nombres compris entre 50 et 200. Le programme *fibo.m* écrit dans la fenêtre éditeur calcule les 20 premiers nombres de Fibonacci :

```
a(1)=0 ;          % initialisation
a(2)= 1 ;         % initialisation
n=20 ;            % on en calcule 20
for i=3 :n        % la boucle récursive
        a(i)=a(i–1)+a(i–2) % définition du nombre de Fibonacci
end
```

ind=*find*(50 <= a & a <= 200) % À l'aide de *find* on trouve les indices *ind* des nombres a compris entre 50 et 200.

Puis dans la fenêtre de commande on tape :

*fibo*

ind=11 12 13 sont les indices des nombres répondant à la question. Ces trois nombres sont donc :

```
a(ind)
ans= 55      89      144
```

Le lecteur pourra observer que le rapport de deux nombres consécutifs de Fibonacci converge rapidement vers le nombre d'or $(1+\sqrt{5})/2$.

### *14. Quelques équivalences d'instructions entre* matlab *et* Octave

| matlab | Octave |
|---|---|
| if … end | if … endif |
| for … end | for … endfor |
| function … end | function … endfunction |

Les points … représentent les lignes de programme entre les deux instructions.

## 3. Exemples de programmes

### *Chapitre 1*

Les programmes ci dessous sont écrits avec le langage OCTAVE.
% Entraînement graphique sur les vecteurs

```
a=[3,3]; % premier vecteur
b=[3,8]; % second vecteur
c=a+b; % somme : a+b
d=a-b; % difference : a-b
norme_de_a=norm(a) % norme (ou module) de a
norme_de_b=norm(b) % norme (ou module) de b
produit_scalaire=dot(a,b,2) % leur produit scalaire
produit_sc=a(1)*b(1)+a(2)*b(2) % comparer avec produit_scalaire
angle_forme_par_a_et_b=acos(produit_scalaire/(norme_de_a*norme_de_b))*
180/pi % en degré

pkg load geometry % package pour utiliser les flèches pour des vecteurs

figure(1);clf % trace la somme de deux vecteurs
plot(0,0);hold on;
plot(a(1),a(2),'r*');hold on;
plot(b(1),b(2),'b*');hold on;
plot(c(1),c(2),'m*');hold on;

m = drawArrow(0,0,a(1),a(2),2,0.1); % trace le vecteur a
set(m.body, 'color', 'g'); % en couleur verte
set(m.body, 'linewidth', 1); % avec une flèche pas trop épaisse
set(m.wing, 'color', 'g');
```

```
m = drawArrow(0,0,b(1),b(2),2,0.1);% trace le vecteur b
set(m.body, 'color', 'k'); % en couleur noire
set(m.body, 'linewidth', 1);
set(m.wing, 'color', 'k');

m = drawArrow(b(1),b(2),c(1),c(2),2,0.1); % trace le vecteur c à partir du point b
set(m.body, 'color', 'g');
set(m.body, 'linewidth', 1);
set(m.body, 'linestyle', '--');
set(m.wing, 'color', 'g');

m = drawArrow(a(1),a(2),c(1),c(2),2,0.1); % trace le vecteur c à partir du point a
set(m.body, 'color', 'k');
set(m.body, 'linewidth', 1);
set(m.body, 'linestyle', '--');
set(m.wing, 'color', 'k');

m = drawArrow(0,0,c(1),c(2),3,0.1); % trace le vecteur c  à partir de l'origine
set(m.body, 'color', 'r'); %  en rouge
set(m.body, 'linewidth', 2);
set(m.wing, 'color', 'r');

xlo=min([a(1),b(1),c(1)])-4;
xhi=max([a(1),b(1),c(1)])+1;
ylo=min([a(2),b(2),c(2)])-4;
yhi=max([a(2),b(2),c(2)])+1;

axis ([xlo, xhi, ylo, yhi], "square");
X = xlabel('X');
Y = ylabel('Y');

text(2.9,5.2,'c','color','r');
text(0.12,7.9,'c=a+b','color','r');
text(1.14,0.92,'a','color','g');
text(4.08,9.59,'a','color','g');
text(0.9,3.1,'b','color','k');
text(3.9,4.6,'b','color','k');

grid on;

figure(2) ; clf; % trace la différence : d=a-b
plot(0,0);hold on;
plot(a(1),a(2),'r*');hold on;
plot(b(1),b(2),'b*');hold on;

m = drawArrow(0,0,a(1),a(2),2,0.1); % trace le vecteur a
set(m.body, 'color', 'g'); % en couleur noir
set(m.body, 'linewidth', 1); % avec une flèche pas trop épaisse
set(m.wing, 'color', 'g');
```

```
m = drawArrow(0,0,b(1),b(2),2,0.1);% trace le vecteur b
set(m.body, 'color', 'k');
set(m.body, 'linewidth', 1);
set(m.wing, 'color', 'k');

m = drawArrow(b(1),b(2),b(1)+d(1),b(2)+d(2),2,0.1); % trace le vecteur d=a-b à
partir de b
set(m.body, 'color', 'm');
set(m.body, 'linewidth', 2);
set(m.wing, 'color', 'm');

grid on;
X = xlabel('X');
Y = ylabel('Y');
text(3.2,5.22,'d=a-b','color','m');
text(1.14,0.92,'a','color','g');
text(1.3,3.1,'b','color','k');
```

## *Chapitre 3*

*Programme Octave pour l'oscillateur harmonique*
Si on introduit une nouvelle variable alors on peut facilement utiliser le solveur d'Octave *lsode* pour résoudre le problème:

```
%       Ce programme intègre les équations d'un oscillateur harmonique
% avec des conditions initiales  x0 et v0. On donne aussi ome2=k/m.

global ome=1.1;     % déclare le ome comme une variable globale
tmax=25.0;            % l' intervalle du temps
x0=1.5;    % coordonnée initiale
v0= -1.2;  % vitesse initiale
Nmb= 2000 ;  % nombre d'itérations
t = linspace (0, tmax, Nmb); % définit le champ discret du temps

function ydot = osci(y,t)      % fonction qui permet d'intégrer directement
global ome;
ydot(1) = y(2);
ydot(2) = -y(1)*ome^2;
endfunction

yini1=[ome*x0;v0];          % initialisation vectorielle pour intégrer y'=f(y,t)
sol=lsode("osci",yini1,t');  % intégration par lsode
% x= sol( :,1)/ome ,          v=sol( :,2)
```

```
figure(1);clf;  % trace la dépendance horaire de x(t) et de v(t)
plot(t,sol(:,1)/ome);hold on;
plot(t,sol(:,2));hold on;
X = xlabel("t");
h = legend({"x","v"},4) ;
set(h,'color','w');
title('x et v d''un oscilateur');

figure(2);clf;        % trace v(x)
plot(sol(:,1)/ome,sol(:,2));
X = xlabel("X");
Y = ylabel("V");
title('Espace des phases');
```

## *Chapitre 8*

**% sous programme ode3dmotion avec coriolis.m ----**

```
tic(); % ???
%-------------------------------------------------------------------------------

function ydot=coriolis(y,t)
g=-9.82;
omga=2*pi/86400;
teta=59.935*pi/180;
omgx=0;
omgy=omga*cos(teta);
omgz=omga*sin(teta);

ydot(1)=y(4);   % dx/dt=vx
ydot(2)=y(5);   % dy/dt=vy
ydot(3)=y(6);   % dz/dt=vz
ydot(4)=y(5)*omgz-y(6)*omgy;    % dvx/dt=ax avec la force de coriolis
ydot(5)=y(6)*omgx-y(4)*omgz;    % dvy/dt=ay avec la force de coriolis
ydot(6)=g+y(4)*omgy-y(5)*omgx;  % dvz/dt=az avec la force de coriolis
endfunction
%-------------------------------------------------------------------------------
% Valeurs  initiales
vx0=0;
vy0=0;
vz0=140.144; % Valeur du problème d'Arnold sur Leningrad
ve=sqrt(vx0^2+vy0^2+vz0^2);

x0=0;
y0=0;
z0=0;
```

```
NMAX=2000;

tmax=2*ve/9.82;
tmax2=2*vz0/9.82;
tmax1=sqrt(2*z0/9.82);
tmax=max(tmax1,tmax2);

t0=0.0;
t=linspace(t0,tmax,NMAX);   % Initialisation du temps

yini=[x0;y0;z0;vx0;vy0;vz0];% position et vitesse initiale pour intégration

%       Intégration

sol1=lsode("coriolis",yini,t');

%-------------------------------------------------------

printf("c'est fini\n");

toc() % ???
%
figure(1)
plot(t,sol1(:,1));
title("x en fonction de t");

figure(2)
plot(t,sol1(:,2));
title("y en fonction de t");

figure(3)
plot(t,sol1(:,3));
title("z en fonction de t");

figure(4)
plot(sol1(:,1),sol1(:,2));
title("y en fonction de x");

max(sol1(:,3))
sol1(1000,3)
sol1(1000,1)  % valeur de déplacement x --montée
sol1(2000,1)  % valeur de déplacement x --montée et descente
```

**% ---- ode_2d motion foucault.m ----**

```
tic();
g=-9.82;
omega=7.27220521664304e-5
latitude=45*pi/180;
lef=2*omega*sin(latitude);
om_zero=0.7; %  correspond à un pendule avec l=20 m
Tez=2*pi/om_zero;

%-------------------------------------------------------------------------

function ydot=Foucault(y,t)
g=-9.82;
omega=7.27220521664304e-5;
latitude=45*pi/180;
lef=2*omega*sin(latitude);
om_zero=0.7; %  correspond ‡ un pendule avec l=20 m
lom=om_zero^2;

ydot(1)=y(3);   % dx/dt=vx
ydot(2)=y(4);   % dy/dt=vy
ydot(3)=lef*y(4)-lom*y(1);   % dx/dt=ax=-w^2x+2f*vy
ydot(4)=-lef*y(3)-lom*y(2);    % dvy/dt=ay=-w^2y-2f*vx
endfunction

% Valeurs  initiales

vx0=0; %
vy0=0;
x0=3.0;
y0=0;
ve=sqrt(vx0^2+vy0^2);

NMAX=60000;
tmax=6000*Tez;
t0=0.0;
t=linspace(t0,tmax,NMAX);   % Initialisation du temps

yini=[x0;y0;vx0;vy0];% position et vitesse initiale pour integration

%///        Intégration                ///////////////
```

```
sol1=lsode("Foucault",yini,t');

%----------------------------------------

printf("c'est fini\n");

toc()

figure(1)
plot(t,sol1(:,1),'r');
h = legend ({"x"});
title("x fonction de t");

figure(2)
plot(t,sol1(:,2),'g');
h = legend ({"y"});
title("y fonction de t");

figure(3)
plot(sol1(:,1),sol1(:,2),'b');
title("y fonction de x");
```

**% ---- ode_3d_motion_coriolis_prb5.m ----**

```
% Debut
tic();
g=9.82;
teta=59.9339*pi/180; % Latitude
NMAX=100000;
%Pour monter un km en hauteur il faut une vitesse initiale%
% https://planet-terre.ens-lyon.fr/article/pendule-pesanteur-latitude.xml
% https://fr.wikipedia.org/wiki/Pesanteur
% Formulaires et tables by Commissions romandes de mathematique, de physique
et de chimie
g0=9.780327*(1+5.3024e-3*sin(teta)^2-5.8e-6*sin(2*teta)^2); % g en fonction de
Latitude
%g=9.8191;
vz0=sqrt(2*1000*g);
% ******************************
  %-------------------------------------------------------------------------
```

```
function ydot=spp_corios(y,t)
teta=59.9339*pi/180; % Latitude
g0=9.780327*(1+5.3024e-3*sin(teta)^2-5.8e-6*sin(2*teta)^2); % g en fonction de
Latitude
g=g0-3.018209e-6*y(3); % dÈpendance de g en fonction de h
omga=7.292e-5; % la norme la vitesse angulaire de rotation
R=6.371e6;  % Le rayon de la Terre en m
teta=59.9339*pi/180; % St Peterbourg
omgx=0; % Composante x de la vitesse angulaire de rotation
omgy=omga*cos(teta); % Composante y de la vitesse angulaire de rotation
omgz=omga*sin(teta); % Composante z de la vitesse angulaire de rotation

ydot(1)=y(4);   % composante x : dx/dt=vx
ydot(2)=y(5);   % composante y : dy/dt=vy
ydot(3)=y(6);   % composante z : dz/dt=vz
ydot(4)=2*(y(5)*omgz-y(6)*omgy);  % composante x : dvx/dt= 2(vywz-vzwy)
ydot(5)=-omgz*(2*y(4)+R*omgy);%    % composante y : dvy/dt= 2(vzwx-vxwz)-
w^2Rsin(teta)cos(teta)
ydot(6)=-g+omgy*(2*y(4)+R*omgy);   % composante z : dvz/dt= 2(vxwy-vywx)+
w^2Rcos≤(teta)-g

endfunction


% Valeurs  initiales
vx0=0.0;
vy0=0.0;
vz0=sqrt(2*1000*g);

x0=0.0;
y0=0.0;
z0=0.0;
t0=0.0;      % instant initial
T=2.0*vz0/g; % la duree du mouvement
tmax=T+0.0306;%0.00584; % choix de la fenetre du temps
%tmax=T+0.00584;
t=linspace(t0,tmax,NMAX);   % Initialisation du temps
yini=[x0;y0;z0;vx0;vy0;vz0];% position et vitesse initiale pour integration

%///        Integration                ///////////////

sol1=lsode('spp_corios',yini,t');  % appel de l'integrateur (ODE)

%*****************************************************
```

```
figure(1);clf;
plot(t,sol1(:,1));hold on;
title('x en fonction de t');

figure(2);clf;
plot(t,sol1(:,2));hold on;
title('y en fonction de t');

figure(3);clf;
plot(t,sol1(:,3));hold on;
plot([0 tmax],[0 0],'linewidth', 1, 'linestyle', '-', 'color', [1 0 0]);hold on;
title('z en fonction de t');


figure(4);clf;
plot3(sol1(:,1),sol1(:,2),sol1(:,3));hold on; % Trajectoire en 3D
plot3([0 0],[0 0],[0 max(sol1(:,3))],'linewidth', 1, 'linestyle', '-', 'color', [1 0 0]);hold
on;
plot3(0,0,0,'bo');hold on; % Position  Initiale en bleu
plot3(sol1(end,1),sol1(end,2),sol1(end,3),'ro');hold on; % Position  finale en rouge
view(31.815,33.621); % pour visualizer l'angle courant [tta,alpha]=view

tx = linspace (min(sol1(:,1)), max(sol1(:,1)), 83)';
ty = linspace (min(sol1(:,2)), max(sol1(:,2)), 83)';
[xx, yy] = meshgrid (tx, ty);

zz = xx.*0;
mesh(xx,yy,zz,'EdgeColor','Y'); % trace le plan z=0

figure(5);clf;
plot(sol1(:,3),sol1(:,1));hold on;
plot(sol1(1,3),sol1(1,1),'bo');hold on;% Position  Initiale en bleu
plot(sol1(end,3),sol1(end,1),'ro');hold on; % Position  finale en rouge
title('x en fonction de z');


figure(6);clf;
plot(sol1(:,3),sol1(:,2));hold on;
plot(sol1(1,3),sol1(1,2),'bo');hold on;% Position  Initiale en bleu
plot(sol1(end,3),sol1(end,2),'ro');hold on; % Position  finale en rouge
title('y en fonction de z');
```

## *Chapitre 12*

Ces programmes sont en langage *matlab.* La méthode numérique est ici explicite.

**sysdyn1.m**

```
%
% intégration numérique d'un systeme dynamique de dimension 1 par schéma matsuno
% exemple: dx/dt=rx-x^3
%
clear all
close all
%
% Parametres
%
r=1;
%
% choix du pas de temps
%
delt=0.005
%
% nombre de pas de temps
%
N=200;
%
% stockage des donnees
%
x=ones(1,N)*NaN;
clock=ones(1,N)*NaN;
%
% choix de sockage tous les nout pas de temps
%
nout=5;
%
% conditions initiales
%
% ici on boucle sur les conditions initiales
% il y a p conditions initiales
%
p=11;
del_x0=1;
x00=-5;
%
for i=1:p
   % conditions initiales
   t=0;
```

```
   x0=x00+del_x0*(i-1);
   % stockage
   it=1;
   x(it)=x0;
   clock(it)=t;
%
% boucle temporelle d'un schema Matsuno: Runge-Kutta d'ordre 1
%
% initialisation de xn
      xn=x0;
      for j=2:N
         t=t+delt;
         % predicteur
         x_p=xn+delt*(r*xn-xn^3);
         % correcteur
         xn=xn+delt*(r*x_p-x_p^3);
         % stockage
         if mod(j,nout) == 0
            it=it+1;
            clock(it)=t;
            x(it)=xn;
         end
      end
%
% graphique
% les p trajectoires x(t)
   figure(1)
   plot(clock,x,'k-')
   hold on
   xlabel('t')
   ylabel('X')
%
end
```

**sysdyn2.m**

```
%
% intégration d'un système dynamique de dimension 2
% exemple sur l'oscillateur amorti:
% dx/dt=y; dy/dt=-mu*y-omega^2*x
%
clear all
close all
%
% paramètres
%
```

```
omega=pi;
mu=1
%
% choix du pas de temps
%
delt = 0.001
%
% nombre de pas de temps N
%
N=10000;
%
% vecteurs de stockage des données
%
x=ones(1,N)*NaN;
y=ones(1,N)*NaN;
clock=ones(1,N)*NaN;
%
% choix de stockage des données tous les nout pas de temps
%
nout=10;
%
% conditions initiales
%
t=0;
x0=-1.5;
y0=-1.;
%
   % stockage des CIs
   it=1;
   x(it)=x0;
   y(it)=y0;
   clock(it)=t;
%
% boucle temporelle d'un schema Matsuno: Runge-Kutta d'ordre 1
%
% initialisation de xn, yn
%
      xn=x0;
      yn=y0;
%
      for k=2:N
         t=t+delt;
         % predicteur
         x_p=xn+delt*yn;
```

```
        y_p=yn+delt*(-mu*yn-(omega^2)*xn);
        % correcteur
        xn=xn+delt*y_p;
        yn=yn+delt*(-mu*y_p-(omega^2)*x_p);
        % stockage
        if mod(k,nout) == 0
           it=it+1;
           clock(it)=t;
           x(it)=xn;
           y(it)=yn;
        end
     end
%
% fin de boucle temporelle
%
% graphique
% la trajectoire x(t), y(t)
   figure(1)
   plot(clock,x,'k-',clock,y,'k--')
   xlabel('t')
   ylabel('X et Y')
   %
   figure(2)
% CIs
   plot(x(1),y(1),'ko','MarkerEdgeColor','k','MarkerFaceColor','k')
   hold on
   plot(x,y,'k-','LineWidth',1.5)
   set(gca,'FontSize',14)
   xlabel('X','FontSize',14)
   ylabel('Y','FontSize',14)
   % axes
   x1=[-0.5 0.5];
   y1=[0 0];
   plot(x1,y1,'k--')
   y1=[-0.5 0.5];
   x1=[0 0];
   plot(x1,y1,'k--')
% ajouter le point fixe
figure(2)
x2=0;
y2=0;
plot(x2,y2,'ks','MarkerEdgeColor','k','MarkerFaceColor','k')
```

**sysdyn3.m**

```
%
% Lorenz system as an example of a dynamical system with 3 variables
%
% dx/dt=sigma(y-x)
% dy/dt=rx-y-xz
% dz/dt=xy-bz
%
clear all
close all
%
% use sigma=10, b=8/3 and vary r akin the Rayleigh number.
%
sigma=10;
b=8/3;
% diagnostic
rh=sigma*(sigma+b+3)/(sigma-b-1)
%
% IC
%
% choose r
disp('Rayleigh number r')
% [explore r=0 1 24.74 28 and values in between]
%
%
% windows of periodicity [99.524 100.795]
% r=100
% intermittent chaos
r=166.3
% noisy periodicity
% r=212
%  period doubling [145 166]
% r=166
%
% fixed points C+, C-
%
x_1=sqrt(b*(r-1))
y_1=x_1
z_1=r-1
%
x_2=-sqrt(b*(r-1));
y_2=x_2;
z_2=r-1;
```

```
%
% select initial condition at t=0 near a fixed point
%
epsi=0.5
x0=x_1*(1+epsi);
y0=y_1*(1+epsi);
z0=z_1*(1+epsi);
%
%
% choose time step delt
%
delt=2.e-03;
%
% choose number of steps
%
ntimes=10000
%
% choose with nplot how often you want to plot or store
%
n_plot=1;
%
% store IC
%
i_store=1
%
time=0;
x(1)=x0;
y(1)=y0;
z(1)=z0;
%
% init time scheme
xn=x0;
yn=y0;
zn=z0;
%
figure(1),clf
   plot(xn,yn,'ro')
   hold on
   if r>=1
      plot(x_1,y_1,'g*')
   end
   xlabel('x')
   ylabel('y')
   title('LORENZ SYSTEM')
```

```
figure(2),clf
   plot(xn,zn,'ro')
   hold on
      if r>=1
      plot(x_1,z_1,'g*')
      end
   xlabel('x')
   ylabel('z')
   title('LORENZ SYSTEM')
figure(3),clf
   plot(yn,zn,'ro')
   hold on
      if r>=1
      plot(y_1,z_1,'g*')
      end
   xlabel('y')
   ylabel('z')
   title('LORENZ SYSTEM')
%
% main loop for integration
for i=1:ntimes
   %
   time=time+delt;
   %
   % time scheme is Matsuno scheme
   % 1/ predictor step
   %
   x_p=xn+delt*sigma*(yn-xn);
   y_p=yn+delt*(r*xn-yn-xn*zn);
   z_p=zn+delt*(xn*yn-b*zn);
   %
   % 2/ corrector step
   %
   xn=xn+delt*sigma*(y_p-x_p);
   yn=yn+delt*(r*x_p-y_p-x_p*z_p);
   zn=zn+delt*(x_p*y_p-b*z_p);
%
% store or plot every n_plot time step
%
      if mod(i,n_plot)==0
         %
         i_store=i_store+1;
         x(i_store)=xn;
         y(i_store)=yn;
         z(i_store)=zn;
      end
```

```
end
%
% phase space plots
%
          figure(1)
          plot(x,y,'b-')
          figure(2)
          plot(x,z,'b-')
          figure(3)
          plot(y,z,'b-')
%
% time plots
%
figure(4),clf
%
subplot(3,1,1)
plot(x,'k-')
hold on
xl=xlim;
y_f1=[x_1 x_1];
y_f2=[x_2 x_2];
   if r>=1
      plot(xl,y_f1,'k--')
      plot(xl,y_f2,'k:')
   end
xlabel('time')
ylabel('x')
title('LORENZ SYSTEM')
%
subplot(3,1,2)
plot(y,'b-')
hold on
xl=xlim;
y_f1=[y_1 y_1];
y_f2=[y_2 y_2];

   if r>=1
      plot(xl,y_f1,'b--')
      plot(xl,y_f2,'b:')
   end
xlabel('time')
ylabel('y')
title('LORENZ SYSTEM')
%
subplot(3,1,3)
```

```
plot(z,'r-')
hold on
xl=xlim;
y_f1=[z_1 z_1];
y_f2=[z_2 z_2];
   if r>=1
      plot(xl,y_f1,'r--')
      plot(xl,y_f2,'r:')
   end
xlabel('time')
ylabel('z')
title('LORENZ SYSTEM')
%
figure(5),clf
plot3(x,y,z)
hold on
plot3(x0,y0,z0,'ro','LineWidth',2)
plot3(x_1,y_1,z_1,'c+','LineWidth',2)
plot3(x_2,y_2,z_2,'g+','LineWidth',2)
xlabel('x')
ylabel('y')
zlabel('z')
title('LORENZ SYSTEM')
%
```